



Enterprise Service Bus

Documentation



WSO2 Enterprise Service Bus

Documentation

Version 5.0.0

Table of Contents

1. WSO2 Enterprise Service Bus Documentation	12
1.1 About WSO2 ESB	13
1.1.1 Introducing the ESB	14
1.1.2 Architecture	16
1.1.3 About this Release	19
1.1.4 Key Concepts	21
1.2 Quick Start Guide	22
1.3 Tutorials	28
1.3.1 Sending a Simple Message	29
1.3.2 Routing Requests Based on Message Content	46
1.3.3 Transforming Message Content	55
1.3.4 Exposing Several Services as a Single Service	63
1.3.5 Storing and Forwarding Messages	75
1.3.6 Using the Gmail Connector	82
1.3.7 Using the Analytics Dashboard	91
1.4 Deep Dive	94
1.4.1 Installation Guide	95
1.4.1.1 Installation Prerequisites	95
1.4.1.2 Installing on Linux	97
1.4.1.3 Installing on Solaris	99
1.4.1.4 Installing on Windows	100
1.4.1.5 Installing as a Windows Service	103
1.4.1.6 Installing as a Linux Service	108
1.4.1.7 Running the Product	110
1.4.2 Product Administration	114
1.4.2.1 Upgrading from a Previous Release	117
1.4.2.2 Clustered Deployment	122
1.4.2.2.1 Clustering with JMS	132
1.4.2.3 Performance Tuning	134
1.4.2.3.1 Network and OS Level Performance Tuning	134
1.4.2.3.2 Java Virtual Machine (JVM) Level Tuning	137
1.4.2.3.3 WSO2 Carbon Platform-Level Tuning	138
1.4.2.3.4 Tuning the HTTP Transport	138
1.4.2.3.5 Tuning the JMS Transport	142
1.4.2.3.6 Tuning the VFS Transport	145
1.4.2.3.7 Tuning the RabbitMQ Transport	147
1.4.2.3.8 Tuning Inbound Endpoints	148
1.4.2.3.9 Tuning the Performance based on Use Case	150
1.4.2.3.10 Tuning Analytics	156
1.4.2.4 JMX Monitoring	158
1.4.2.5 SNMP Monitoring	163
1.4.2.6 Working with Proxy Servers	165
1.4.2.7 Viewing the Handlers in Message Flows	168
1.4.2.8 Enabling SSL Tunneling through a Proxy Server	171
1.4.2.9 Governing External References Across Environments	172

1.4.2.9.1 Pass-through Proxy Service with an Endpoint Reference	184
1.4.2.9.2 Secure Proxy Service with a Policy and Endpoint Referenced	187
1.4.2.9.3 Server Profiles and Sample Security Policy	190
1.4.2.9.4 Using the VM Image for Governing External References Across Environments	193
1.4.3 Enterprise Integration Patterns	194
1.4.4 WSO2 ESB Tooling	194
1.4.4.1 Installing WSO2 ESB Tooling	195
1.4.4.2 Working with ESB Artifacts	197
1.4.4.3 Packaging your Artifacts into Composite Applications	208
1.4.4.4 Importing Existing Projects into Workspace	208
1.4.5 Triggering Messages	211
1.4.5.1 Working with APIs	211
1.4.5.1.1 Working with APIs via WSO2 ESB Tooling	216
1.4.5.1.2 Working with APIs via the Management Console	217
1.4.5.1.3 Configuring Specific Use Cases	220
1.4.5.1.4 Securing APIs	240
1.4.5.1.5 Unusual Scenarios for HTTP Methods in REST	252
1.4.5.1.6 Using REST with APIs	255
1.4.5.1.7 Per-API Logs in WSO2 ESB	255
1.4.5.2 Working with Proxy Services	256
1.4.5.2.1 Working with Proxy Services via WSO2 ESB Tooling	261
1.4.5.2.2 Working with Proxy Services via the Management Console	262
1.4.5.2.3 Using REST with a Proxy Service	275
1.4.5.2.4 Applying Security to a Proxy Service	281
1.4.5.3 Working with Inbound Endpoints	290
1.4.5.3.1 WSO2 ESB Inbound Endpoints	291
1.4.5.3.2 Working with Inbound Endpoints via WSO2 ESB Tooling	330
1.4.5.3.3 Working with Inbound Endpoints via the Management Console	333
1.4.5.4 Working with Scheduled Tasks	334
1.4.5.4.1 Working with Scheduled Tasks via WSO2 ESB Tooling	334
1.4.5.4.2 Working with Scheduled Tasks via the Management Console	335
1.4.5.5 Using REST	348
1.4.6 Mediating Messages	348
1.4.6.1 Mediators	350
1.4.6.1.1 Aggregate Mediator	353
1.4.6.1.2 BAM Mediator	356
1.4.6.1.3 Bean Mediator	357
1.4.6.1.4 Cache Mediator	359
1.4.6.1.5 Call Mediator	364
1.4.6.1.6 Call Template Mediator	370
1.4.6.1.7 Callout Mediator	373
1.4.6.1.8 Class Mediator	378
1.4.6.1.9 Clone Mediator	382
1.4.6.1.10 Conditional Router Mediator	385
1.4.6.1.11 Data Mapper Mediator	387
1.4.6.1.12 DBLookup Mediator	440
1.4.6.1.13 DB Report Mediator	452
1.4.6.1.14 Drop Mediator	462

1.4.6.1.15 EJB Mediator	464
1.4.6.1.16 Enqueue Mediator	466
1.4.6.1.17 Enrich Mediator	467
1.4.6.1.18 Entitlement Mediator	471
1.4.6.1.19 Event Mediator	476
1.4.6.1.20 FastXSLT Mediator	478
1.4.6.1.21 Fault Mediator	480
1.4.6.1.22 Filter Mediator	484
1.4.6.1.23 ForEach Mediator	487
1.4.6.1.24 Header Mediator	489
1.4.6.1.25 In and Out Mediators	493
1.4.6.1.26 Iterate Mediator	494
1.4.6.1.27 Log Mediator	498
1.4.6.1.28 Loopback Mediator	501
1.4.6.1.29 OAuth Mediator	502
1.4.6.1.30 PayloadFactory Mediator	504
1.4.6.1.31 POJOCommand Mediator	510
1.4.6.1.32 Property Mediator	513
1.4.6.1.33 Publish Event Mediator	516
1.4.6.1.34 Respond Mediator	519
1.4.6.1.35 Router Mediator	520
1.4.6.1.36 Rule Mediator	520
1.4.6.1.37 Script Mediator	526
1.4.6.1.38 Send Mediator	534
1.4.6.1.39 Sequence Mediator	538
1.4.6.1.40 Smooks Mediator	540
1.4.6.1.41 Spring Mediator	543
1.4.6.1.42 Store Mediator	544
1.4.6.1.43 Switch Mediator	546
1.4.6.1.44 Throttle Mediator	548
1.4.6.1.45 Transaction Mediator	554
1.4.6.1.46 URLRewrite Mediator	560
1.4.6.1.47 Validate Mediator	563
1.4.6.1.48 XQuery Mediator	571
1.4.6.1.49 XSLT Mediator	575
1.4.6.2 Working with Mediators via WSO2 ESB Tooling	580
1.4.6.3 Working with Mediators via the Management Console	581
1.4.6.3.1 Adding a Mediator to a Sequence	581
1.4.6.3.2 Adding a Child Mediator	583
1.4.6.3.3 Editing a Mediator	585
1.4.6.3.4 Deleting a Mediator	588
1.4.6.4 Working with Local Registry Entries	589
1.4.6.5 Creating Custom Mediators	591
1.4.6.6 Best Practices for Mediation	591
1.4.6.7 Mediation Sequences	592
1.4.6.7.1 Working with Sequences via WSO2 ESB Tooling	594
1.4.6.7.2 Working with Sequences via the Management Console	597
1.4.6.8 Working with Message Stores and Message Processors	605

1.4.6.8.1 Message Stores	605
1.4.6.8.2 Message Processors	625
1.4.6.8.3 Guaranteed Delivery with Failover Message Store and Scheduled Failover Message Forwarding Processor	635
1.4.6.9 Working with Message Builders and Formatters	639
1.4.6.9.1 Message Relay	641
1.4.6.10 Prioritizing Messages	644
1.4.6.10.1 Adding a Priority Executor	646
1.4.6.10.2 Editing a Priority Executor	648
1.4.6.10.3 Deleting a Priority Executor	650
1.4.6.11 Transactions	651
1.4.6.12 Debugging Mediation	652
1.4.7 Working with Endpoints	661
1.4.7.1 WSO2 ESB Endpoints	662
1.4.7.1.1 Address Endpoint	663
1.4.7.1.2 Dynamic Load-balance Endpoint	667
1.4.7.1.3 Failover Group	667
1.4.7.1.4 HTTP Endpoint	668
1.4.7.1.5 WSDL Endpoint	671
1.4.7.1.6 Load-balance Group	673
1.4.7.1.7 Indirect and Resolving Endpoints	676
1.4.7.1.8 Default Endpoint	677
1.4.7.1.9 Template Endpoint	679
1.4.7.1.10 Recipient List Endpoint	680
1.4.7.2 Working with Endpoints via WSO2 ESB Tooling	684
1.4.7.3 Working with Endpoints via the Management Console	685
1.4.7.4 Endpoint Error Handling	688
1.4.7.5 Configuring Endpoints using REST APIs	695
1.4.7.6 Using Jconsole to Manage State Transition	695
1.4.8 Working with Web Services	699
1.4.8.1 Accessing Services	699
1.4.8.2 Managing Web Services	701
1.4.8.2.1 Managing Service Groups	703
1.4.8.3 Working with Topics and Events	706
1.4.8.3.1 Adding a New Topic	707
1.4.8.3.2 Adding a New Subtopic	709
1.4.8.3.3 Viewing Topic Details	713
1.4.8.3.4 Subscribing to a Topic	720
1.4.8.3.5 Deleting a Topic	723
1.4.8.4 Per-Service Logs in WSO2 ESB	725
1.4.9 Working with Templates	726
1.4.9.1 Endpoint Template	727
1.4.9.2 Sequence Template	730
1.4.9.3 Working with Templates via WSO2 ESB Tooling	735
1.4.9.4 Working with Templates via the Management Console	736
1.4.9.4.1 Adding a New Endpoint Template	736
1.4.9.4.2 Editing an Endpoint Template	744
1.4.9.4.3 Deleting an Endpoint Template	746

1.4.9.4.4 Adding a New Sequence Template	749
1.4.9.4.5 Editing a Sequence Template	753
1.4.9.4.6 Deleting a Sequence Template	756
1.4.9.4.7 Managing Sequence Templates	759
1.4.10 Working with Transports	761
1.4.10.1 Configuring Transports	761
1.4.10.2 ESB Transports	764
1.4.10.2.1 HTTP PassThrough Transport	765
1.4.10.2.2 HTTP-NIO Transport	765
1.4.10.2.3 HTTPS-NIO Transport	767
1.4.10.2.4 HTTP Servlet Transport	768
1.4.10.2.5 HTTPS Servlet Transport	772
1.4.10.2.6 FIX Transport	773
1.4.10.2.7 JMS Transport	775
1.4.10.2.8 VFS Transport	836
1.4.10.2.9 Local Transport	841
1.4.10.2.10 MailTo Transport	843
1.4.10.2.11 MSMQ Transport	846
1.4.10.2.12 RabbitMQ AMQP Transport	847
1.4.10.2.13 TCP Transport	863
1.4.10.2.14 UDP Transport	863
1.4.10.2.15 HL7 Transport	864
1.4.10.2.16 Multi-HTTPS Transport	875
1.4.10.2.17 MQ Telemetry Transport	880
1.4.10.2.18 WebSocket Transport	882
1.4.11 Working with Modules	883
1.4.11.1 Adding a Module	883
1.4.11.2 Engaging Modules	885
1.4.11.3 Writing an Axis2 Module	886
1.4.12 WSO2 ESB Analytics	892
1.4.12.1 Prerequisites to Publish Statistics	893
1.4.12.2 Analyzing WSO2 ESB with the Analytics Dashboard	896
1.4.12.2.1 Analyzing ESB Statistics Overview	898
1.4.12.2.2 Analyzing Statistics for Proxy Services	902
1.4.12.2.3 Analyzing Statistics for REST APIs	904
1.4.12.2.4 Analyzing Statistics for Sequences	907
1.4.12.2.5 Analyzing Statistics for Endpoints	910
1.4.12.2.6 Analyzing Statistics for Inbound Endpoints	913
1.4.12.2.7 Analyzing Statistics for Mediators	915
1.4.12.2.8 Analyzing Statistics for Messages	918
1.4.12.3 Monitoring WSO2 ESB with WSO2 ESB Analytics	919
1.4.12.4 Extending ESB Analytics	926
1.4.12.4.1 Customizing Statistics Publishing	926
1.4.12.4.2 Monitoring JMX Based Statistics	926
1.4.12.5 Monitoring WSO2 ESB with WSO2 Analytics in a Multi-tenant Environment	928
1.4.13 Cloud Services Gateway (CSG)	935
1.4.13.1 Adding a New CSG Server	937
1.4.13.2 Configuring CSG Properties	939

1.4.13.3 Publishing a Service to CSG Server	939
1.4.13.4 Unpublishing a Service from CSG Server	940
1.4.14 Error Handling	942
1.4.15 Java Message Service (JMS) Support	945
1.4.15.1 JMS UseCases	946
1.4.15.1.1 ESB as a JMS Consumer	946
1.4.15.1.2 ESB as a JMS Producer	949
1.4.15.1.3 ESB as Both a JMS Producer and Consumer	951
1.4.15.1.4 JMS Synchronous Invocations : Dual Channel HTTP-to-JMS	952
1.4.15.1.5 JMS Synchronous Invocations : Quad Channel JMS-to-JMS	956
1.4.15.1.6 Store and Forward Using JMS Message Stores	957
1.4.15.1.7 Publish-Subscribe with JMS	965
1.4.15.1.8 Shared Topic Subscription with WSO2 ESB	969
1.4.15.1.9 Detecting Repeatedly Redelivered Messages via WSO2 ESB Using the JMSXDeliveryCount Property	977
1.4.15.1.10 Using WSO2 ESB as a JMS Producer and Specifying a Delivery Delay on Messages	983
1.4.15.2 JMS Samples	991
1.4.15.3 Advanced Topics	991
1.4.15.3.1 JMS Security Management	991
1.4.15.3.2 JMS Transactions	998
1.4.15.3.3 Working with Multiple Types of Brokers	100
1.4.15.3.4 JMS MapMessage Support	100
1.4.15.4 JMS Troubleshooting Guide	100
1.4.15.5 JMS FAQ	101
1.4.16 JSON Support	101
1.4.17 REST Support	103
1.4.18 WebSocket Support	103
1.4.18.1 Sending a Message from a WebSocket Client to a WebSocket Endpoint	103
1.4.18.2 Sending a Message from a WebSocket Client to an HTTP Endpoint	104
1.4.18.3 Sending a Message from a HTTP Client to a WebSocket Endpoint	104
1.4.19 Integrating with WSO2 BAM, WSO2 DAS and WSO2 CEP	104
1.4.19.1 Configuring a Server Profile	104
1.4.19.2 Working with Event Sinks	104
1.4.20 Integrating with Other Technologies	105
1.4.20.1 AMQP Support	105
1.4.20.2 ebMS Integration	105
1.4.20.3 PayPal WS API Integration	105
1.4.20.4 SAP Integration	105
1.4.21 Extending the ESB	106
1.4.21.1 Working with Connectors	106
1.4.21.1.1 Working with Connectors via WSO2 ESB Tooling	106
1.4.21.1.2 Working with Connectors via the Management Console	107
1.4.21.1.3 Using a Connector	107
1.4.21.2 Writing Tasks	107
1.4.21.2.1 Writing Tasks Sample	108
1.4.21.3 Writing a Synapse Handler	108
1.4.21.4 Writing a Custom Message Builder and Formatter	109

1.4.21.5 Writing a WSO2 ESB Mediator	109
1.4.21.5.1 Writing Custom Configuration Implementations for Mediators	109
1.4.21.6 Places for Putting Custom Mediators	110
1.4.21.7 Writing Custom Mediator Implementations	110
1.4.21.8 Uploading Artifacts	110
1.4.22 Reference Guide	110
1.4.22.1 Configuration Files	110
1.4.22.1.1 XML files	110
1.4.22.1.2 Properties Files	115
1.4.22.2 Calling Admin Services from Apps	116
1.4.22.3 Default Ports of WSO2 Products	116
1.4.22.4 ESB Tools	117
1.4.22.4.1 WSDL2Java	117
1.4.22.4.2 Java2WSDL	117
1.4.22.4.3 Try It	117
1.4.22.4.4 WSDL Validator	118
1.4.22.5 Properties Reference	118
1.4.22.5.1 Generic Properties	118
1.4.22.5.2 HTTP Transport Properties	119
1.4.22.5.3 SOAP Headers	119
1.4.22.5.4 Axis2 Properties	119
1.4.22.5.5 Synapse Message Context Properties	120
1.4.22.5.6 Accessing Properties with XPath	120
1.4.22.6 Synapse Configuration Reference	121
1.4.22.7 Setting Up Host Names and Ports	121
1.5 Samples	121
1.5.1 Setting Up the ESB Samples	121
1.5.2 Using the Sample Clients	123
1.5.3 Message Mediation Samples	123
1.5.3.1 Sample 0: Introduction to ESB	123
1.5.3.2 Sample 1: Simple Content-Based Routing (CBR) of Messages	123
1.5.3.3 Sample 2: CBR with the Switch-Case Mediator Using Message Properties	123
1.5.3.4 Sample 3: Local Registry Entry Definitions, Reusable Endpoints and Sequences	123
1.5.3.5 Sample 4: Specifying a Fault Sequence with a Regular Mediation Sequence	124
1.5.3.6 Sample 5: Creating SOAP Fault Messages and Changing the Direction of a Message	124
1.5.3.7 Sample 6: Manipulating SOAP Headers and Filtering Incoming and Outgoing Messages	1248
1.5.3.8 Sample 7: Using Schema Validation and the Usage of Local Registry for Storing Configuration Metadata	124
1.5.3.9 Sample 8: Introduction to Static and Dynamic Registry Resources and Using XSLT Transformations	125
1.5.3.10 Sample 9: Introduction to Dynamic Sequences with the Registry	125
1.5.3.11 Sample 10: Introduction to Dynamic Endpoints with the Registry	125
1.5.3.12 Sample 11: Using a Full Registry-Based Configuration and Sharing a Configuration Between Multiple Instances	125
1.5.3.13 Sample 12: One-Way Messaging in a Fire-and-Forget Mode through ESB	125
1.5.3.14 Sample 13: Dual Channel Invocation Through Synapse	126
1.5.3.15 Sample 14: Using Sequences and Endpoints as Local Registry Items	126

1.5.3.16 Sample 15: Using the Enrich Mediator for Message Copying and Content Enrichment	126
1.5.3.17 Sample 16: Introduction to Dynamic and Static Registry Keys	126
1.5.3.18 Sample 17: Transforming / Replacing Message Content with PayloadFactory Mediator	1267
1.5.3.19 Sample 18: Transforming a Message Using ForEach Mediator	127
1.5.4 Advanced Mediation with Endpoints	127
1.5.4.1 Sample 50: POX to SOAP conversion	127
1.5.4.2 Sample 51: MTOM and SwA Optimizations and Request/Response Correlation	127
1.5.4.3 Sample 52: Using Load Balancing Endpoints to Handle Peak Loads	127
1.5.4.4 Sample 53: Using Failover Endpoints to Handle Peak Loads	128
1.5.4.5 Sample 54: Session Affinity Load Balancing between Three Endpoints	128
1.5.4.6 Sample 55: Session Affinity Load Balancing between Failover Endpoints	128
1.5.4.7 Sample 56: Using a WSDL Endpoint as the Target Endpoint	129
1.5.4.8 Sample 57: Dynamic Load Balancing between Three Nodes	129
1.5.4.9 Sample 58: Static Load Balancing between Three Nodes	129
1.5.4.10 Sample 59: Weighted load balancing between 3 endpoints	129
1.5.4.11 Sample 60: Routing a Message to a Static List of Recipients	129
1.5.4.12 Sample 61: Routing a Message to a Dynamic List of Recipients	130
1.5.4.13 Sample 62: Routing a Message to a Dynamic List of Recipients and Aggregating Responses	130
1.5.5 Quality of Service Samples in Message Mediation	130
1.5.5.1 Sample 100: Using WS-Security for Outgoing Messages	130
1.5.6 Proxy Service Samples	130
1.5.6.1 Sample 150: Introduction to Proxy Services	130
1.5.6.2 Sample 151: Custom Sequences and Endpoints with Proxy Services	131
1.5.6.3 Sample 152: Switching Transports and Message Format from SOAP to REST POX	131
1.5.6.4 Sample 153: Routing Messages that Arrive to a Proxy Service without Processing Security Headers	131
1.5.6.5 Sample 154: Load Balancing with Proxy Services	131
1.5.6.6 Sample 155: Dual Channel Invocation on Both Client Side and Server Side of Synapse with Proxy Services	131
1.5.6.7 Sample 156: Service Integration with specifying the receiving sequence	131
1.5.6.8 Sample 157: Conditional Router for Routing Messages based on HTTP URL, HTTP Headers and Query Parameters	132
1.5.6.9 Quality of Service Samples in Service Mediation	132
1.5.6.9.1 Sample 200: Using WS-Security with policy attachments for proxy services	132
1.5.7 Transports Samples and Switching Transports	132
1.5.7.1 Sample 250: Introduction to Switching Transports	132
1.5.7.2 Sample 251: Switching from HTTP(S) to JMS	132
1.5.7.3 Sample 252: Pure Text (Binary) and POX Message Support with JMS	132
1.5.7.4 Sample 253: Bridging from JMS to HTTP and Replying with a 202 Accepted Response	133
1.5.7.5 Sample 254: Using the File System as Transport Medium (VFS)	133
1.5.7.6 Sample 255: Switching from FTP Transport Listener to Mail Transport Sender	133
1.5.7.7 Sample 256: Proxy Services with the MailTo Transport	133
1.5.7.8 Sample 257: Proxy Services with the FIX Transport	134
1.5.7.9 Sample 258: Switching from HTTP to FIX	134
1.5.7.10 Sample 259: Switch from FIX to HTTP	134
1.5.7.11 Sample 260: Switch from FIX to AMQP	135

1.5.7.12 Sample 261: Switching between FIX Versions	135
1.5.7.13 Sample 262: CBR of FIX Messages	135
1.5.7.14 Sample 263: Transport Switching - JMS to http/s Using JBoss Messaging(JBM)	136
1.5.7.15 Sample 264: Sending Two-Way Messages Using JMS transport	136
1.5.7.16 Sample 265: Accessing a Windows Share Using the VFS Transport	136
1.5.7.17 Sample 266: Switching from TCP to HTTP/S	136
1.5.7.18 Sample 267: Switching from UDP to HTTP/S	136
1.5.7.19 Sample 268: Proxy Services with the Local Transport	136
1.5.7.20 Sample 270: Transport switching from HTTP to MSMQ and MSMQ to HTTP	137
1.5.7.21 Sample 271: File Processing	137
1.5.7.22 Sample 272: Publishing and Subscribing using WSO2 ESB's MQ Telemetry Transport	138
1.5.8 Introduction to ESB Tasks	138
1.5.8.1 Sample 300: Introduction to Tasks with a Simple Trigger	138
1.5.9 Advanced Mediations with Advanced Mediators	138
1.5.9.1 Using Scripts in Mediation (Script Mediator)	138
1.5.9.1.1 Sample 350: Introduction to the Script Mediator Using JavaScript	138
1.5.9.1.2 Sample 351: Inline script mediation with JavaScript	138
1.5.9.1.3 Sample 352: Accessing Synapse message context API using a scripting language	138
1.5.9.1.4 Sample 353: Using Ruby Scripts for Mediation	138
1.5.9.1.5 Sample 354: Using Inline Ruby Scripts for Mediation	139
1.5.9.2 Database Interactions in Mediation (DBLookup / DBReport)	139
1.5.9.2.1 Sample 360: Introduction to DBLookup Mediator	139
1.5.9.2.2 Sample 361: Introduction to DBReport Mediator	139
1.5.9.2.3 Sample 362: DBReport and DBLookup Mediators Together	139
1.5.9.2.4 Sample 363: Reusable Database Connection Pools	140
1.5.9.2.5 Sample 364: Using Mediators to Execute Database Stored Procedures	140
1.5.9.3 Throttling Messages (Throttle Mediator)	140
1.5.9.3.1 Sample 370: Introduction to Throttle Mediator and Concurrency Throttling	140
1.5.9.3.2 Sample 371: Restricting Requests Based on Policies	140
1.5.9.3.3 Sample 372: Use of Both Concurrency Throttling and Request-Rate-Based Throttling	141
1.5.9.4 Extending the Mediation in Java (Class Mediator)	141
1.5.9.4.1 Sample 380: Writing your own Custom Mediation in Java	141
1.5.9.4.2 Sample 381: Class Mediator to CBR Binary Messages	141
1.5.9.5 Evaluating XQuery for Mediation (XQuery Mediator)	141
1.5.9.5.1 Sample 390: Introduction to XQuery Mediator	141
1.5.9.5.2 Sample 391: Using Data from an External XML Document within XQuery	142
1.5.9.6 Splitting Messages into Parts and Processing in Parallel (Iterate/Aggregate)	142
1.5.9.6.1 Sample 400: Message Splitting and Aggregating the Responses	142
1.5.9.7 Caching Responses Over Requests (Cache Mediator)	142
1.5.9.7.1 Sample 420: Simple Cache Implemented on ESB for the Actual Service	142
1.5.9.8 Mediating JSON Messages	142
1.5.9.8.1 Sample 440: Converting JSON to XML Using XSLT	142
1.5.9.8.2 Sample 441: Converting JSON to XML Using JavaScript	142
1.5.9.9 Rewriting the URL (URL Rewrite Mediator)	142
1.5.9.9.1 Sample 450: Introduction to the URL Rewrite Mediator	142
1.5.9.9.2 Sample 451: Conditional URL Rewriting	143

1.5.9.9.3 Sample 452: Conditional URL Rewriting with Multiple Rules	143
1.5.9.10 Eventing (Event Mediator)	143
1.5.9.10.1 Sample 460: Introduction to Eventing and Event Mediator	143
1.5.9.11 Mediating with Spring	143
1.5.9.11.1 Sample 470: How to Initialize and use a Spring Bean as a Mediator	143
1.5.10 Invoking Web Services	143
1.5.10.1 Sample 430: Callout Mediator for Synchronous Service Invocation	143
1.5.10.2 Sample 500: Call Mediator for Non-Blocking Service Invocation	143
1.5.11 Introduction to Rule Mediator	143
1.5.11.1 Sample 600 : Simple Message Transformation - Rule Mediator for Message Transformation	143
1.5.11.2 Sample 601 : Advance Rule Based Routing - Switching Routing Decision According to the Rules - Rule Mediator as Switch mediator	143
1.5.12 Miscellaneous Samples	144
1.5.12.1 Sample 650: File Hierarchy-Based Configuration Builder	144
1.5.12.2 Sample 651: Using Synapse Observers	144
1.5.12.3 Sample 652: Priority Based Message Mediation	144
1.5.12.4 Sample 653: NHTTP Transport Priority Based Dispatching	144
1.5.12.5 Sample 654: Smooks Mediator	144
1.5.12.6 Sample 655: Message Relay - Basics Sample	145
1.5.12.7 Sample 656: Message Relay - Builder Mediator	145
1.5.12.8 Sample 657: Distributed Transaction Management	145
1.5.12.9 Sample 658: Huge XML Message Processing with Smooks Mediator	145
1.5.12.10 Sample 659: Huge EDI Message Processing with Smooks Mediator	145
1.5.13 Store and Forward Messaging Patterns with Message Stores and Processors	146
1.5.13.1 Sample 700: Introduction to Message Store	146
1.5.13.2 Sample 701: Introduction to Message Sampling Processor	146
1.5.13.3 Sample 702: Introduction to Message Forwarding Processor	146
1.5.13.4 Sample 703: Adding Security to Message Forwarding Processor	146
1.5.13.5 Sample 704: RESTful Invocations with Message Forwarding Processor	146
1.5.13.6 Sample 705: Load Balancing with Message Forwarding Processor	146
1.5.14 Template Samples	147
1.5.14.1 Sample 750: Stereotyping XSLT Transformations with Templates	147
1.5.14.2 Sample 751: Message Split Aggregate Using Templates	147
1.5.14.3 Sample 752: Load Balancing Between 3 Endpoints With Endpoint Templates	147
1.5.15 REST API Management	147
1.5.15.1 Sample 800: Introduction to REST API	147
1.5.16 Inbound Endpoint Samples	148
1.5.16.1 Sample 900: Inbound Endpoint File Protocol Sample (VFS)	148
1.5.16.2 Sample 901: Inbound Endpoint JMS Protocol Sample	148
1.5.16.3 Sample 902: HTTP Inbound Endpoint Sample	148
1.5.16.4 Sample 903: HTTPS Inbound Endpoint Sample	148
1.5.16.5 Sample 904: Inbound Endpoint Kafka Protocol Sample	148
1.5.16.6 Sample 905: Inbound HL7 with Automatic Acknowledgement	149
1.5.16.7 Sample 906: Inbound Endpoint MQTT Protocol Sample	149
1.5.16.8 Sample 907: Inbound Endpoint RabbitMQ Protocol Sample	149
1.6 FAQ	149

WSO2 Enterprise Service Bus Documentation

Welcome to the WSO2 Enterprise Service Bus (WSO2 ESB) 5.0.0 documentation!

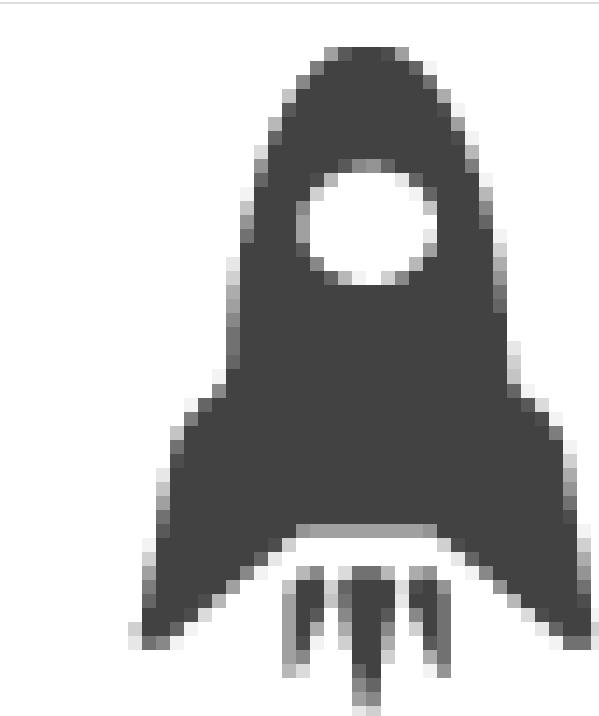
Get started with WSO2 ESB

If you are new to using WSO2 ESB, start here:



Get familiar with WSO2 ESB

Understand the basics of WSO2 ESB and its architecture.



Use the Quick Start Guide

Get up and running quickly while learning the fundamentals.



Try out the Samples
Experiment with common usage scenarios.

Deep dive into WSO2 ESB

	Installation Guide		WSO2 ESB Tooling		WSO2 ESB Analytics	
	Product Extensions		Enterprise Integration Patterns			Reference Guide

To download a PDF of this document or a selected part of it, click [here](#) (only generate one PDF at a time). You can also use this link to export to HTML or XML.

About WSO2 ESB

The topics in this section introduce the WSO2 ESB, including the business cases it solves, its features, and its architecture.

- [Introducing the ESB](#)
- [Architecture](#)
- [About this Release](#)
- [Key Concepts](#)

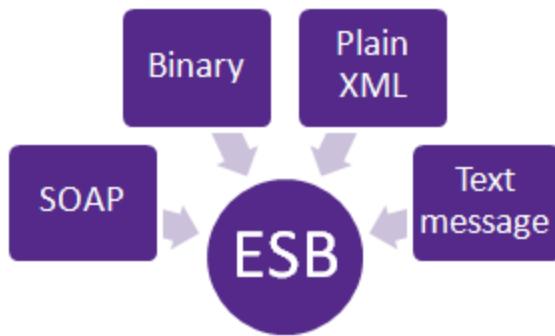
Introducing the ESB

This page introduces WSO2 ESB in the following sections:

- [Overview](#)
- [Distributed Computing Evolution](#)
- [ESB in Context of SOA](#)

Overview

An enterprise service bus (ESB) is a software architecture construct that enables communication among various applications. Instead of having to make each of your applications communicate directly with each other in all their various formats, each application simply communicates with the ESB, which handles transforming and routing the messages to their appropriate destinations.



An ESB provides its fundamental services through an event-driven and standards-based messaging engine (the bus). Thanks to ESB, integration architects can exploit the value of messaging without writing code. Developers typically implement an ESB using technologies found in a category of middleware infrastructure products, usually based on recognized standards. As with a Service-Oriented Architecture (SOA), an ESB is essentially a collection of enterprise architecture design patterns that is now implemented directly by many enterprise software products.

WSO2 ESB is a fast, light-weight, and versatile enterprise service bus. It is 100% open source and is released under [Apache Software License Version 2.0](#), one of the most business-friendly licenses available today. Using WSO2 ESB you can perform a variety of [enterprise integration patterns \(EIPs\)](#), including filtering, transforming, and routing SOAP, binary, plain XML, and text messages that pass through your business systems by HTTP, HTTPS, JMS, mail, etc.

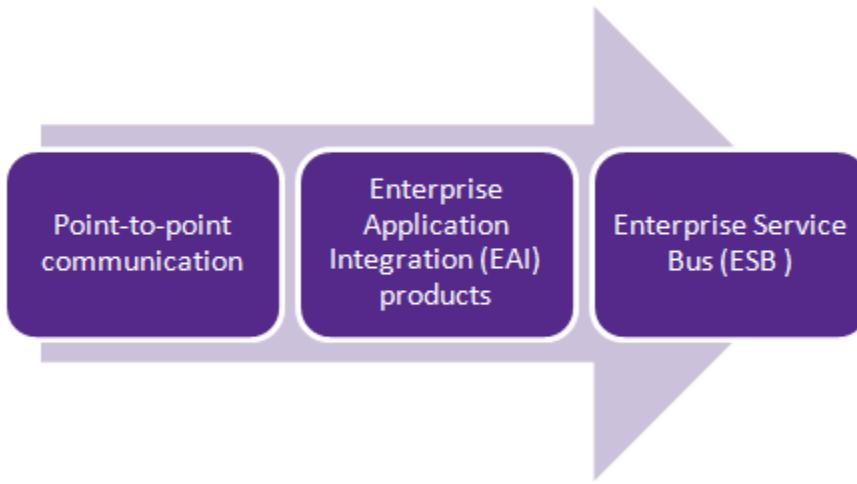
Distributed Computing Evolution

ESBs are part of an evolution of **distributed computing**. Early distributed computing involved pure **point-to-point communication** between systems. This was the simple, obvious way to create complex applications from

distributed components. This is actually a reasonable solution when there are a small number of applications that need to work together. However, the number of point-to-point communications grows proportional to the square of the number of applications in the enterprise. This becomes unmanageable for a large enterprise.

The late 90's saw the rise of **Enterprise Application Integration (EAI)** products. These aimed to allow enterprises to scale beyond the limitations caused by point-to-point integration solutions. The typical solution used a hub-and-spoke architecture. This is a solution still in use by many enterprises today. In a hub-and-spoke EAI, all communication is routed through a central hub. The number of point-to-point communications scales linearly with the number of enterprises, so this is a great improvement over point-to-point architectures. However, the hubs become bottlenecks in the system.

ESBs can be thought of as the next step in the logical progression described above. The ESB acts as a message broker to all applications in the enterprise. It allows for more granular, base functions to be exposed with orchestration provided as needed. This greatly improves the flexibility of the system and allows for more rapid and cheaper adaptation to changes.



ESB in Context of SOA

Here is a list of functionality in an ESB runtime that encourage SOA:

- **Message routing and distribution.** The applications should not have hard-coded destinations for messages or services. The ESB can help in two ways:
 - by supporting **virtualization** (mapping logical destinations to real destinations) and
 - by supporting **event architectures**, where the publisher does not need to know about the subscribers.
- **Management-** The ESB should provide a common set of management capabilities which give a common view of all services and endpoints. This includes:
 - **alerting,**
 - **statistics,**
 - **audit and**
 - **logging.**
- **Excellent support for the Web architecture** - ESBs should encourage good use of HTTP and the Web architecture. Support for HTTP proxying and caching.
- **XML performance-** If an ESB is going to help manage and route XML messages then it has to do it with minimum overhead. It certainly has to be much more scalable and performant than the applications it talks to. Fundamentally this promotes two key technical requirements:
 - **non-blocking IO** (meaning that the ESB must not block while waiting for applications to respond) and
 - **Streaming XML** (meaning that the ESB can send XML through without having to create a large in-memory buffer and fully parse every message).
- **Security Control** - Managing distributed security is one of the hardest problems in SOA and augmenting application security with a set of central security controls is often an essential component in an enterprise infrastructure.

Tip

Automatic schema from WSDLs and other XSD files is available in WSO2 ESB.

Architecture

This page describes the WSO2 ESB architecture in the following sections:

- Overview
- Messaging architecture
- Component architecture

Overview

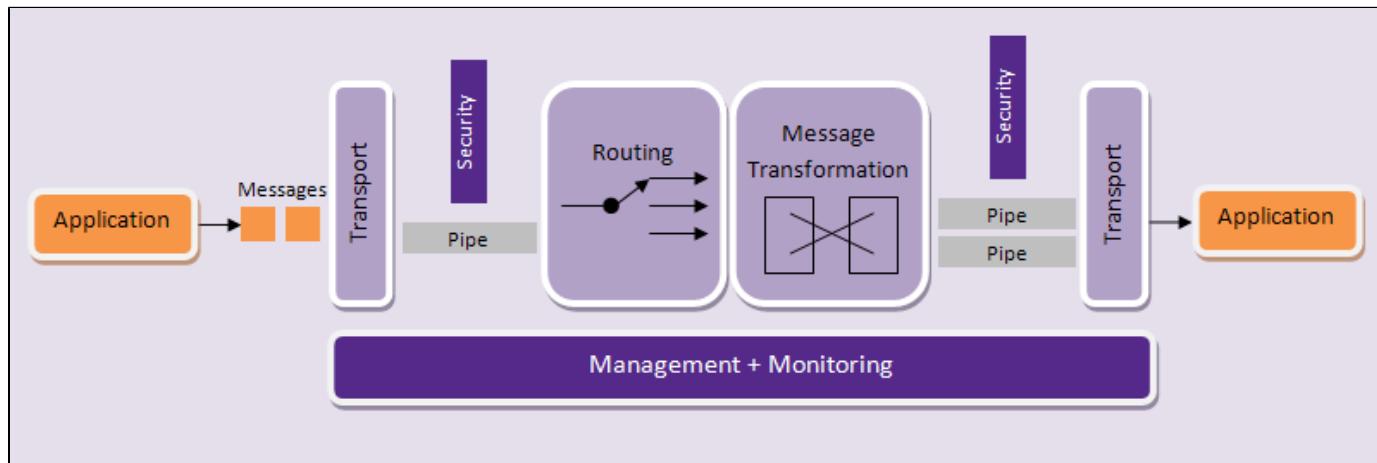
Application infrastructure on the enterprises may be inherently complex, comprising hundreds of applications with completely different semantics. Some of these applications are custom built, some are acquired from third parties, and some can be a combination of both and can be operating in different system environments.

Integration among these heterogeneous applications is vital to the enterprise. Different services may be using different data formats and communication protocols. Physical locations of services can change arbitrarily. All these constraints mean your applications are still tightly coupled together. An ESB can be used to loosen these couplings between different services and service consumers.

WSO2 ESB is a full-fledged, enterprise-ready ESB. It is built on the [Apache Synapse](#) project, which is built using the [Apache Axis2](#) project. All the components are built as [OSGi](#) bundles.

Messaging architecture

The following diagram illustrates the ESB architecture from a messaging perspective (the components of the pipes are not in a specific order):



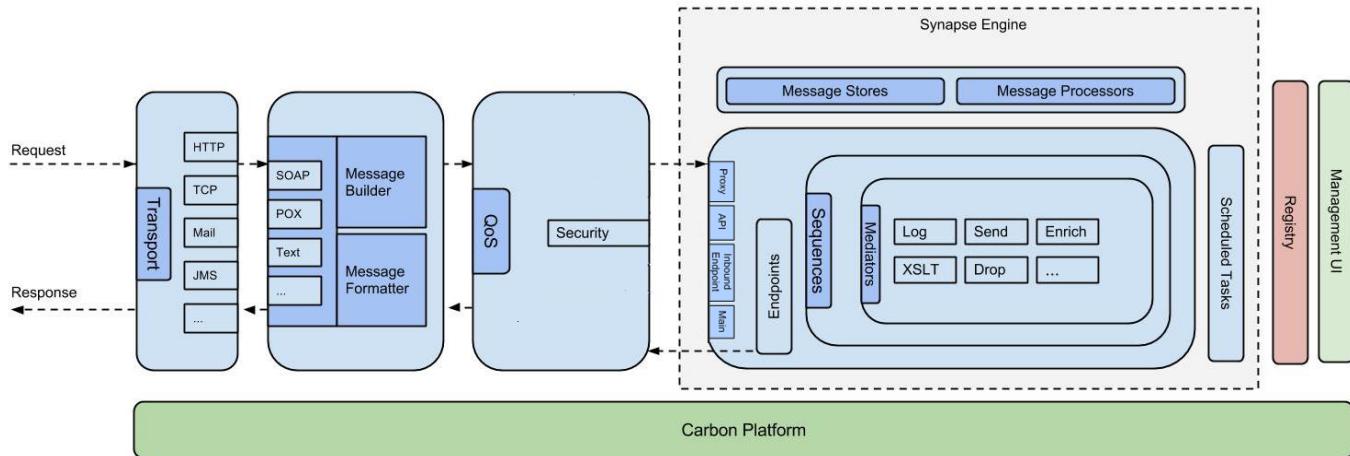
1. An application sends a message to the ESB.
2. The message is picked up by a [transport](#).
3. The transport sends the message through a message pipe, which handles quality of service aspects such as security. Internally, this pipe is the in-flow and out-flow of Axis2. The ESB can operate in two modes:
 - [Mediating Messages](#) - A single pipe is used.
 - [Proxy Services](#) - Separate pipes connecting the transport to different proxy services are used.
4. Both message transformation and routing can be considered as a single unit. As the diagram specifies, there is no clear separation between message transformation components and routing components. In WSO2 ESB, this is known as the mediation framework. Some transformations take place before the routing decision has been made while others take place after the routing decision. This is part of the Synapse implementation.

5. The message is injected to the separate pipes depending on the destinations. Here again, quality of service aspects of the messages are determined.
6. The transport layer takes care of the transport protocol transformations required by the ESB.

The diagram shows how a request propagates to its actual [endpoint](#) through the ESB using its architecture. Response handling is the reverse of this operation. There are other areas like [Working with Scheduled Tasks](#) and [Events](#) that are not shown in the diagram. All these components can be [analyzed](#) and [monitored](#) through WSO2 ESB Analytics.

Component architecture

This section describes the component-based architecture of WSO2 ESB.



Transports

A transport is responsible for carrying messages that are in a specific format. WSO2 ESB supports all the widely used transports including HTTP/s, JMS, and VFS, and domain-specific transports like FIX. You can easily add a new transport using the Axis2 transport framework and plug it into the ESB. Each transport provides a receiver, which the ESB users to receive messages, and a sender, which it uses to send messages. The transport receivers and senders are independent of the ESB core.

See [Working with Transports](#).

Message builders and formatters

When a message comes in to the ESB, the receiving transport selects a **message builder** based on the message's content type. It uses that builder to process the message's raw payload data and convert it into common XML, which the ESB mediation engine can then read and understand. WSO2 ESB includes message builders for text-based and binary content.

Conversely, before a transport sends a message out from the ESB, a **message formatter** is used to build the outgoing stream from the message back into its original format. As with message builders, the message formatter is selected based on the message's content type.

You can implement new message builders and formatters using the Axis2 framework.

See [Working with Message Builders and Formatters](#).

Endpoints

An endpoint defines an external destination for a message. An endpoint can connect to any external service after configuring it with any attributes or semantics needed for communicating with that service. For example, an endpoint could represent a URL, a mailbox, a JMS queue, or a TCP socket, along with the settings needed to connect to it.

You can specify an endpoint as an [address endpoint](#), [WSDL endpoint](#), a [load balancing endpoint](#), and more. An endpoint is defined independently of transports, allowing you to use the same endpoint with multiple transports. When you configure a message mediation sequence or a proxy service to handle the incoming message, you specify which transport to use and the endpoint where the message will be sent.

See [Working with Endpoints](#).

Proxy services

Proxy services are virtual services that receive messages and optionally process them before forwarding them to a service at a given [endpoint](#). This approach allows you to perform necessary transformations and introduce additional functionality without changing your existing service. Any available transport can be used to receive and send messages from the proxy services. A proxy service is externally visible and can be accessed using a URL similar to a normal web service address.

See [Working with Proxy Services](#).

APIs

An API in WSO2 ESB is analogous to a web application deployed in the ESB runtime. Each API is anchored at a user-defined URL context, much like how a web application deployed in a servlet container is anchored at a fixed URL context. An API will only process requests that fall under its URL context. The API defines one or more resources, which is a logical component of an API that can be accessed by making a particular type of HTTP call. This approach allows you to send messages directly into the ESB using REST.

See [Working with APIs](#).

Inbound endpoints

An inbound endpoint is a message source that can be configured dynamically. In the ESB, when it comes to the existing axis2 based transports, only the HTTP transport works in a multi-tenant mode. Inbound endpoints support all transports to work in a multi-tenant mode.

See [Working with Inbound Endpoints](#)

Topics

A topic allows services to receive messages when a specific type of event occurs by subscribing to messages that have been published to a specific topic.

See [Working with Topics and Events](#).

Mediators

Mediators are individual processing units that perform a specific function, such as sending, transforming, or filtering messages. WSO2 ESB includes a comprehensive mediator library that provides functionality for implementing widely used [enterprise integration patterns \(EIPs\)](#). You can also easily write a custom mediator to provide additional functionality using various technologies such as Java, scripting, and Spring.

See [Mediators](#).

Sequences

A sequence is a set of mediators organized into a logical flow, allowing you to implement pipes and filter patterns. You can add sequences to proxy services and APIs.

See [Mediation Sequences](#).

Tasks

A task allows you to run a piece of code triggered by a timer. WSO2 ESB provides a default task implementation, which you can use to inject a message to the ESB at a scheduled interval. You can also write your own custom

tasks by implementing a Java interface.

See [Working with Scheduled Tasks](#).

QoS component

The Quality of Service (QoS) component implements security.

See [Managing Service Groups](#).

Registry

A registry is a content store and metadata repository. WSO2 ESB provides a registry with a built-in repository that stores the configuration and configuration metadata that define your messaging architecture. You can also use an external registry/repository for resources such as WSDLs, schemas, scripts, XSLT and XQuery transformations, etc. You can hide or merge one or more remote registries behind a local registry interface, and you can configure the ESB to poll these registries to update its current configurations.

See [Working with the Registry](#).

Management and configuration GUI

The [Management Console](#) provides a graphical user interface (GUI) that allows you to easily configure the components mentioned above.

Carbon platform

WSO2 Carbon provides the runtime environment for the ESB. It contains all the platform-wide features such as security, logging, clustering, caching, etc. Because of the platform, you can install extra features on WSO2 ESB that don't ship with the default package, which makes WSO2 ESB more flexible and extensible. For complete information, see the [WSO2 Carbon Documentation](#).

Flexible deployment

You can deploy WSO2 ESB in a clustered environment with a load balancer to achieve failover and high availability. For complete information, see the [WSO2 Clustering and Deployment Guide](#).

About this Release

What is new in this release

WSO2 ESB version 5.0.0 is the successor of version 4.9.0, and it comes complete with the runtime, tooling, and analytics in a single release.

WSO2 ESB 5.0.0 contains the following new features and enhancements:

- Tooling support provided via [WSO2 ESB tooling](#) to create and manage ESB artifacts.. For information on how to get started with WSO2 ESB tooling, see [Installing WSO2 ESB Tooling](#).
- An analytics component to analyze WSO2 ESB mediation statistics. For information on WSO2 ESB analytics component, and how to analyze ESB mediation statistics via ESB analytics, see [WSO2 ESB Analytics](#).
- JMS 2.0 specification support for existing JMS messaging features. WSO2 ESB supports the following new messaging features introduced with JMS 2.0:
 - Shared Topic Subscription. For a use case that demonstrates shared topic subscription with WSO2 ESB, see [Shared Topic Subscription with WSO2 ESB](#).
 - JMSX Delivery Count. For a use case that demonstrates how WSO2 ESB can be used to detect repeatedly redelivered messages, see [Detecting Repeatedly Redelivered Messages via WSO2 ESB Using the JMSXDeliveryCount Property](#).
 - JMS Message Delivery Delay. For a use case that demonstrates how you can use WSO2 ESB as a JMS producer and specify a delivery delay on messages, see [Using WSO2 ESB as a JMS Producer](#)

and Specifying a Delivery Delay on Messages.

- WebSocket support via WSO2 ESB WebSocket Transport, WSO2 ESB WebSocket Inbound Protocol, and WSO2 ESB Secure WebSocket Inbound Protocol.
- [Data Mapper Mediator](#), a data transformation tool that can be used to easily convert and transform data.
- Mediation debugger that allows debugging mediation flows via WSO2 ESB tooling. For information on how to debug a message mediation flow, see [Debugging Mediation](#).

What has changed in this release

This release includes a few features and functionalities that are deprecated and might be removed in a future release as well as some features and functionalities that are removed.

Deprecated features and functionalities

- The BAM mediator is deprecated and replaced by the [Publish Event Mediator](#).

Removed features and functionalities

- The Axis2 Quality of Services UI to apply security policies, throttling and caching has been removed from the management console. For information on how to apply security, throttling and caching, see the following:
 - The recommended approach to apply security to a proxy service is via WSO2 ESB tooling. For instructions on the process of applying security to a proxy service, see [Applying Security to a Proxy Service](#).
 - The recommended approach to enable throttling is to use the [throttle mediator](#).
 - The recommended approach to enable caching is to use the [cache mediator](#).
- The EHcache based implementation has been removed from ESB 4.9.0 and above due to distributed caching related issues with EHcache. We have moved onto a hazelcast based caching implementation with ESB 4.9.0 onwards, and officially support the hazelcast based caching implementation.

Note

The EHCache jar that was shipped with ESB 4.8.1 and older versions under the <ESB_HOME>/repository/components/plugins directory is not shipped with ESB 4.9.0 and later. If you do want to use a third party jar like EHCache jar with ESB 4.9.0 and above, you can add the jar to the <ESB_HOME>/repository/components/lib directory.

Compatible WSO2 product versions

WSO2 ESB 5.0.0 is based on WSO2 Carbon 4.4.8 and is expected to be compatible with any WSO2 product that is based on the same Carbon version. If you come across any compatibility issues, [contact team WSO2](#). For information on the third-party software required for running the ESB and its samples, see [Installation Prerequisites](#).

Fixed issues

This release includes several bug fixes.

- For a complete list of issues fixed in the runtime of this release, see [WSO2 ESB 5.0.0 - Fixed Issues](#).
- For a complete list of issues fixed in tooling of this release, see [WSO2 ESB 5.0.0 Tooling - Fixed Issues](#).
- For a complete list of issues fixed in analytics of this release, see [WSO2 ESB 5.0.0 Analytics - Fixed Issues](#).

Known issues

This release includes several known issues.

There is a [known issue](#) related to preserving CDATA blocks when you send CDATA inside a SOAP message

payload. As a temporary workaround, you can add the `XMLInputFactory.properties` file with the following property to the `<ESB_HOME>` directory.

```
javax.xml.stream.isCoalescing=false
```

When you set this property to `false`, it can cause an issue in uploading CAR files from the Management Console. Therefore, if you need to upload a CAR file after adding the `XMLInputFactory.properties` file as specified above, you can either copy the CAR file directly to the `<ESB_HOME>/repository/deployment/server/carbo`napps directory or you can upload it via [WSO2 Developer Studio 3.7.1](#) and later versions.

For a complete list of known issues in this release, see [WSO2 ESB - Open Issues](#).

Key Concepts

Let's take a look at some concepts and terminology that you need to know.

[[Maven Multi Module Project](#)] [[REST API](#)] [[API Resource](#)] [[Endpoints](#)] [[Mediators](#)] [[Sequences](#)] [[Composite Application Project](#)] [[Service Chaining](#)] [[Store and Forward](#)] [[Connectors](#)]

Maven Multi Module Project

WSO2 ESB tooling creates separate projects and a separate Maven `pom.xml` file for most deployable artifacts. In Maven-centric development, however, you have a parent project and some child modules, including a separate distribution module that is a child module of the parent project. To achieve this model, you can create a Maven Multi Module project in your workspace, create your artifact projects nested within it, and then create the Composite Application project for the distribution module.

Building all deployable artifacts within a Maven Multi Module project allows you to build the deployable artifacts using Continuous Integration (CI) tools.

REST API

A REST API allows you receive messages that are sent from the client directly into the ESB and perform specific logic on it based on the instructions in the HTTP call when connecting to a REST backend service. This approach allows you to perform necessary transformations and introduce additional functionality without changing your existing backend service.

A REST API defined in ESB is made up of one or many resources.

API Resource

An API resource is used by the WSO2 ESB mediation engine to mediate incoming requests, forward them to a specified endpoint, mediate the responses from the endpoint, and send the responses back to the client that originally requested them. We can create an API resource to process defined HTTP request method/s that are sent to the backend service. The In sequence handles incoming requests and sends them to the back-end service, and the Out sequence handles the responses from the back-end service and sends them back to the requesting client.

Endpoints

This defines an external destination for the outgoing message from the ESB. Typically an endpoint is based on a service address or a WSDL.

WSO2 ESB has support for a range of different endpoint types, allowing the ESB to connect with advanced types of backends. For detailed information on each endpoint type available with WSO2 ESB, see [WSO2 ESB Endpoints](#).

Mediators

A mediator is a full-powered processing unit in the ESB. At run-time, a mediator has access to all the parts of the

ESB along with the current message and can do virtually anything with the message. Using mediators, you do various message transformations and orchestrate multiple backend services to achieve the design you want.

Sequences

A list of mediators that take action on the request, such as transforming its format and then sending it to a back-end service. By default there are three sequences engaged as `in`, `out` and `fault`.

Composite Application Project

To deploy the artifacts to WSO2 ESB, we must first package the artifact project/s into a Composite Application (C-App) project. A C-App also allows you to easily port your artifacts from one environment to another. For detailed information on C-Apps, see [Introduction to Composite Applications](#) in WSO2 Administration Guide.

Service Chaining

Service chaining is a popular use case in ESB, where several services are exposed as a single service, aggregated service. ESB is used for the integration and sequential calling of these services so that the required response can be provided to the client.

Learn how to implement a simple service chaining scenario in this [tutorial](#).

Store and Forward

Store and forward messaging pattern is used in asynchronous messaging. This can be used when integrating with systems that accept message traffic at a given rate and handling failover scenarios. In this pattern, messages are sent to a [Message Store](#) where they are temporarily stored before they are delivered to their destination by a [Message Processor](#).

Learn how to implement a store and forward pattern using the in-memory store of ESB in this [tutorial](#).

Connectors

A connector is a collection of templates that define operations that can be called from the ESB and is used when connecting the ESB to external third party APIs. WSO2 ESB provides a variety of connectors via the [WSO2 Connector Store](#).

Learn how to use a connector in your ESB configuration in this [tutorial](#).

Quick Start Guide

WSO2 Enterprise Service Bus (ESB) is a lightweight, high performance, comprehensive ESB to enable interoperability among various heterogeneous systems and business applications. WSO2 ESB effectively supports integration standards and patterns.

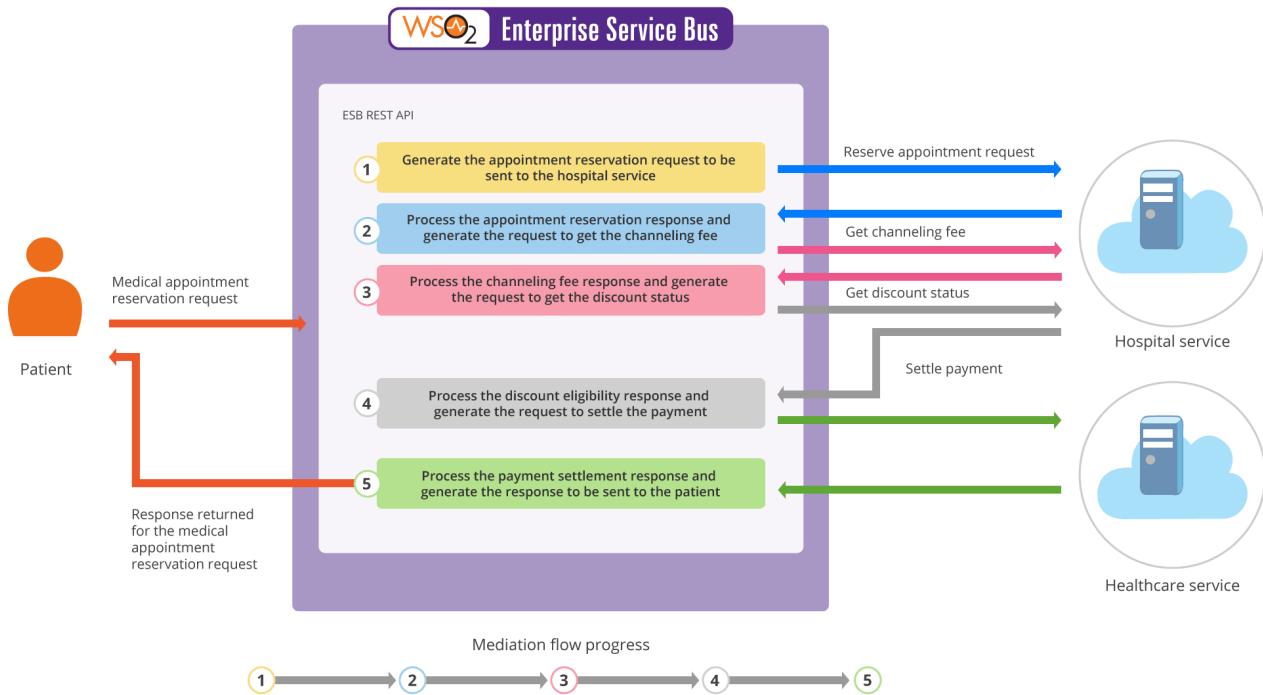
Let's take a look at the basic use cases of the ESB using a sample scenario.

- Introducing the sample
- Deploying the sample to create mediation artifacts
- Sending a request to the ESB
- Analyzing the mediation statistics
- Where to go next

Introducing the sample

This is a healthcare system in which a user reserves a medical appointment by providing his/her personal details, preferred hospital, doctor's name, credit card information etc.

The ESB processes the user's request and returns an appointment confirmation or a refusal. The following diagram shows how the ESB handles all the service calls and payload manipulations involved in a reservation:



Note the following regarding the diagram:

1. When a patient makes a request to reserve an appointment, the request is sent to a REST API that is configured in the ESB. The REST API processes the incoming request and forwards it to the hospital service, which does the appointment reservation.
2. The REST API processes the appointment reservation response and sends a service call to the hospital service to get the doctor's channeling fee and receives the response.
3. The REST API processes the channeling fee response and sends a service call to the hospital service to get the user's eligibility for a discount.
4. When the responses for these service calls are returned, the REST API calculates the actual fee for the appointment and creates a new request with the payment details to be sent to the healthcare service. The

- healthcare service processes the payment settlement and returns the response to the REST API.
- The REST API takes the payment settlement response and generates the response to be sent to the user.

Let's go through the main features of WSO2 ESB using this healthcare service as an example.

Deploying the sample to create mediation artifacts

Instead of creating the ESB artifacts that are required to run this sample scenario from scratch, let's deploy them using a Composite Application Archive (CAR) file. The CAR file consists of mediation artifacts such as the REST API, the **endpoints** (backend services the requests are sent to), and the **sequences** (list of mediators) that allow the ESB to process the requests.

Tip: For information on how to create all the required ESB artifacts in a Composite Application Project, see the tutorial [Sending a Simple Message](#).

Before you begin,

- Install Oracle Java SE Development Kit (JDK) version 1.8.* and set the JAVA_HOME environment variable.
- Go to <http://wso2.com/products/enterprise-service-bus/>, click **DOWNLOAD** to download the ESB runtime ZIP file, and then extract the ZIP file.
The path to this folder will be referred to as <ESB_HOME> throughout the quick start guide.
- Go to <http://wso2.com/products/enterprise-service-bus/>, click **Analytics** to download the ESB analytics ZIP file, and then extract the ZIP file.
The path to this folder will be referred to as <ANALYTICS_HOME> throughout the quick start guide.
- Download `wso2QuickStartGuideCapp_1.0.0.car` from [here](#) and save it in a preferred location in your computer.
- If you are running on Windows, download the `snappy-java_1.1.1.7.jar` from [here](#) and copy the JAR file to <ANALYTICS_HOME>\repository\components\lib directory.

Let's get started!

Follow the steps below to deploy the healthcare sample:

- Set the following properties in the <ESB_HOME>/repository/conf/synapse.properties file to true so that the ESB can publish mediation statistics:

```
...
mediation.flow.statistics.enable=true
mediation.flow.statistics.tracer.collect.payloads=true
mediation.flow.statistics.tracer.collect.properties=true
...
```

- Start the WSO2 ESB Analytics server by going to <ANALYTICS_HOME>/bin using the Command-Line/Terminal and executing one of the following commands:
 - On Linux/Mac OS: `sh wso2server.sh`
 - On Windows: `wso2server.bat --run`
- Start WSO2 ESB server by going to <ESB_HOME>/bin using the Command-Line/Terminal and executing one of the following commands: (**Be sure you have successfully started the Analytics server before taking this step.**)
 - On Linux/Mac OS: `sh wso2server.sh`
 - On Windows: `wso2server.bat --run`
- Open the WSO2 ESB Management Console using <https://localhost:9443/carbon/> and log in using `admin/admin` as the credentials.

5. Deploy `WSO2QuickStartGuideCapp_1.0.0.car` to the ESB as follows:

Tip: The CAR file was created using [WSO2 ESB tooling](#). For information on getting started with WSO2 ESB tooling, see [Working with WSO2 ESB Tooling](#).

- On the **Main** tab of the Management Console, go to **Manage -> Carbon Applications** and click **Add**.
- Click **Choose File**, select the `WSO2QuickStartGuideCapp_1.0.0.car` file you downloaded in the **Before you begin...** section above, and click **Upload**.

Note

After you upload a CAR file, you can confirm that it was successfully deployed by taking the following step:

- On the **Main** tab of the Management Console, go to **Manage -> Carbon Applications** and click **List**. The **Carbon Applications List** screen appears. If successfully deployed, the CAR file will be listed here.

6. Follow the steps below to enable statistics and tracing for the REST API:

- On the **Main** tab of the Management Console, go to **Manage -> Service Bus** and click **APIs**. The **Deployed APIs** screen appears, and you will see the `HospitalServiceApi` listed as follows:

Select	API Name	API Invocation URL	Action
<input type="checkbox"/>	HospitalServiceApi	http://10.100.5.71:8280/hospitalservice	Enable Statistics Enable Tracing

- To enable the collection of mediation statistics for the REST API, click **Enable Statistics**.
- To enable mediation tracing for the REST API, click **Enable Tracing**.

7. Follow the steps below to enable statistics for the endpoints:

- On the **Main** tab of the Management Console, go to **Manage -> Service Bus**, and click **Endpoints**. The **Manage Endpoints** screen appears, and you will see several endpoints listed.

Select	Endpoint Name	Type	Action
<input type="checkbox"/>	channellingFee	HTTP Endpoint	Switch Off Enable Statistics Edit Delete
<input type="checkbox"/>	discountEligibility	HTTP Endpoint	Switch Off Enable Statistics Edit Delete
<input type="checkbox"/>	getPaymentDetails	HTTP Endpoint	Switch Off Enable Statistics Edit Delete
<input type="checkbox"/>	reserveAppointmentEp	HTTP Endpoint	Switch Off Enable Statistics Edit Delete
<input type="checkbox"/>	settlePayment	HTTP Endpoint	Switch Off Enable Statistics Edit Delete

- To enable the collection of mediation statistics for the endpoints, click **Enable Statistics** for each

endpoint.

You have now created mediation artifacts in the ESB and enabled statistics. Let's send a request to the ESB.

Sending a request to the ESB

1. Create a JSON file named `reserve_appointment.json` with the following request:

```
{
  "reserveRequest": {
    "patientDetails": {
      "name": "Mark Smith",
      "dob": "1990-03-19",
      "ssn": "ASJK-asnda-AAA",
      "address": "100 MAIN ST, SEATTLE WA 98104, USA",
      "phone": "0770596754",
      "email": "marksm@mymail.com"
    },
    "doctor": "thomas collins",
    "hospital": "grand oak community hospital",
    "category": "surgery",
    "cardNo": "7844481124110331"
  }
}
```

2. In a new terminal, navigate to the location where you saved the `reserve_appointment.json` request file, and then execute the following command:

```
curl -v -X POST "http://localhost:8280/hospitalservice/reserve" --header
"Content-Type: application/json" -d @reserve_appointment.json -k -v
```

This sends an appointment request to the REST API, which is the configuration inside the ESB that receives and processes the request. Once all the message flows take place as described in the sample introduction, you will see a response similar to the following on the console:

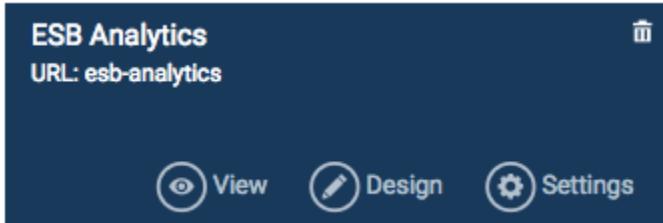
```
{
  "appointmentNumber": 1,
  "Doctor": {
    "fee": 7000,
    "name": "thomas collins",
    "availability": "9.00 a.m - 11.00 a.m",
    "hospital": "grand oak community hospital",
    "category": "surgery"
  },
  "Payment": {
    "patient": "Mark Smith",
    "actualFee": 7000,
    "discount": 0,
    "discounted": 7000,
    "paymentID": "c5eddddf3-d7d7-4756-8a03-5e8e31f2e716",
    "status": "Settled"
  }
}
```

When you receive the response for the request that you sent, you can view the mediation statistics related to the processing that happens within WSO2 ESB via WSO2 ESB Analytics.

Now let's have a look at how you can view the mediation statistics on the Analytics Dashboard of WSO2 ESB Analytics.

Analyzing the mediation statistics

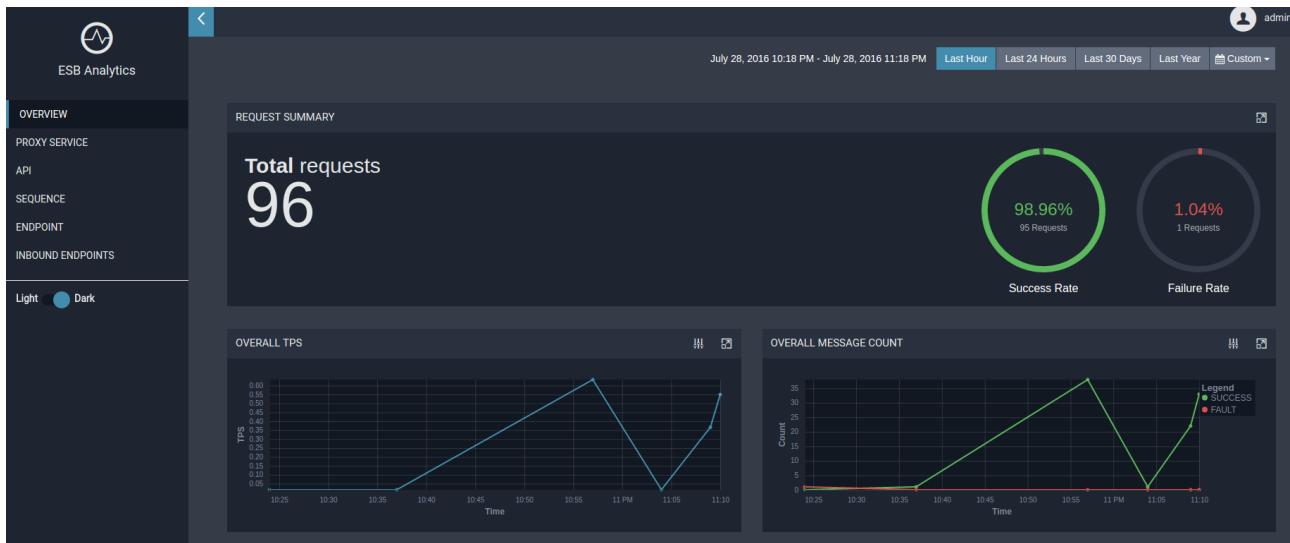
1. In a new browser window or tab, open <https://localhost:9444/carbon/> and log into the Analytics management console using `admin` for both the username and password.
2. On the **Main** tab, click **Analytics Dashboard** and log in using `admin` for both the username and password. You will then see the following:



3. Click **View** to open the ESB Analytics Dashboard. The **OVERVIEW** page is displayed by default. You will see an **OVERVIEW** page similar to the following:

Note

The statistics you see will be for the number of requests that you have sent. If you follow the steps in the [Sending a request to the ESB](#) section, and send just one request, you will see the statistics for just that request.



- To view statistics for a specific date range, select the required date range from the top right menu bar. If you want to view statistics for a specific date, click **Custom** in the menu bar and enter the required date. For more information on analyzing the statistics displayed on this page, see [Analyzing ESB Statistics Overview](#).
4. To view statistics for the REST API, click **API** on the left navigator and search for `HospitalServiceApi . F` or more information on analyzing statistics displayed on this page, see [Analyzing Statistics for REST APIs](#).
 5. To view statistics for an endpoint, click **ENDPOINT** on the left navigator and search for the required endpoint.

You can view statistics for the following endpoints on this page:

- `reserveAppointmentEp`
- `discountEligibility`
- `channellingFee`
- `settlePayment`

For more information on analyzing statistics displayed on this page, see [Analyzing Statistics for Endpoints](#).

Where to go next

You have now explored the basics of using WSO2 ESB and are ready to get started creating your own ESB artifacts. Next, you can have a look at the lessons in the [Tutorials](#) section to understand more about some of the most common usage scenarios of WSO2 ESB, how to use WSO2 ESB Tooling to create your artifacts, and how to use WSO2 ESB Analytics to publish statistics related to the processing carried out in WSO2 ESB to the Analytics Dashboard.

Explore the topics in the table of contents of this guide to learn more about working with the ESB runtime, ESB tooling as well as ESB analytics.

For information on the various integration patterns you can implement, see [Enterprise Integration Patterns with WSO2 ESB](#).

Tutorials

This tutorial gives you a complete introduction to the fundamentals and most common usage scenarios of WSO2 ESB. It walks you through installation of the ESB and supporting products.

In this tutorial we will use a connected healthcare service scenario where we will build on this use case as we progress through the tutorial.

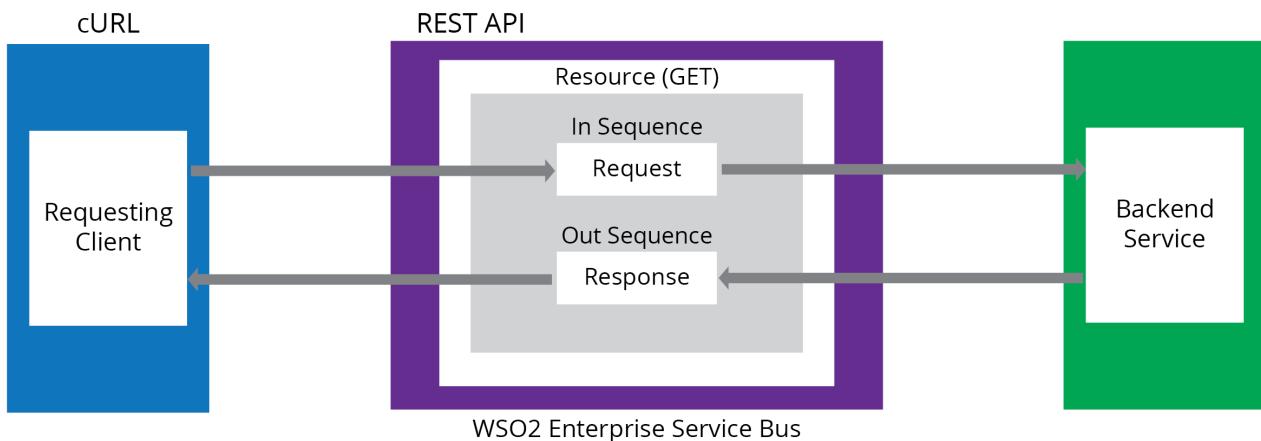
The tutorial comprises of the following sections:

- Sending a Simple Message
- Routing Requests Based on Message Content
- Transforming Message Content
- Exposing Several Services as a Single Service
- Storing and Forwarding Messages
- Using the Gmail Connector
- Using the Analytics Dashboard

Sending a Simple Message

Let's try a simple scenario where a patient makes an inquiry specifying the doctor's specialization(category) to retrieve a list of doctors that match the specialization. We will configure an API resource in ESB that will receive the client request, instead of the client sending messages directly to the back-end service, thereby decoupling the client and the back-end service.

In this tutorial, you create a REST API in WSO2 ESB to connect to a REST back-end service that is defined as a HTTP Endpoint in WSO2 ESB.



Before you begin,

1. Install Oracle Java SE Development Kit (JDK) version 1.8.* and set the JAVA_HOME environment variable.
2. Go to <http://wso2.com/products/enterprise-service-bus/>, click **DOWNLOAD** to download the ESB runtime ZIP file, and then extract the ZIP file.
The path to this folder will be referred to as <ESB_HOME> throughout the tutorials.
3. Go to <http://wso2.com/products/enterprise-service-bus/>, click **Tooling** to select and download the relevant ESB tooling ZIP file, and then extract the ZIP file.
The path to this folder will be referred to as <TOOLING_HOME> throughout the tutorials.

For more detailed installation instructions, see the [Installing WSO2 ESB Tooling](#).

To start Eclipse on a Mac for the first time, open a terminal and execute the following commands:

```
cd <DevStudio_Home>/Eclipse.app/Contents/MacOS
chmod +x eclipse
./eclipse
```

Thereafter, you can start Eclipse by double-clicking the Eclipse icon in `<DevStudio_Home>`.

Let's get started!

This tutorial includes the following sections:

- Creating the message mediation artifacts
 - Creating the deployable artifacts project
 - Connecting to the back-end service
 - Mediating requests to the back-end service
- Deploying the artifacts to WSO2 ESB
- Sending requests to WSO2 ESB

Creating the message mediation artifacts

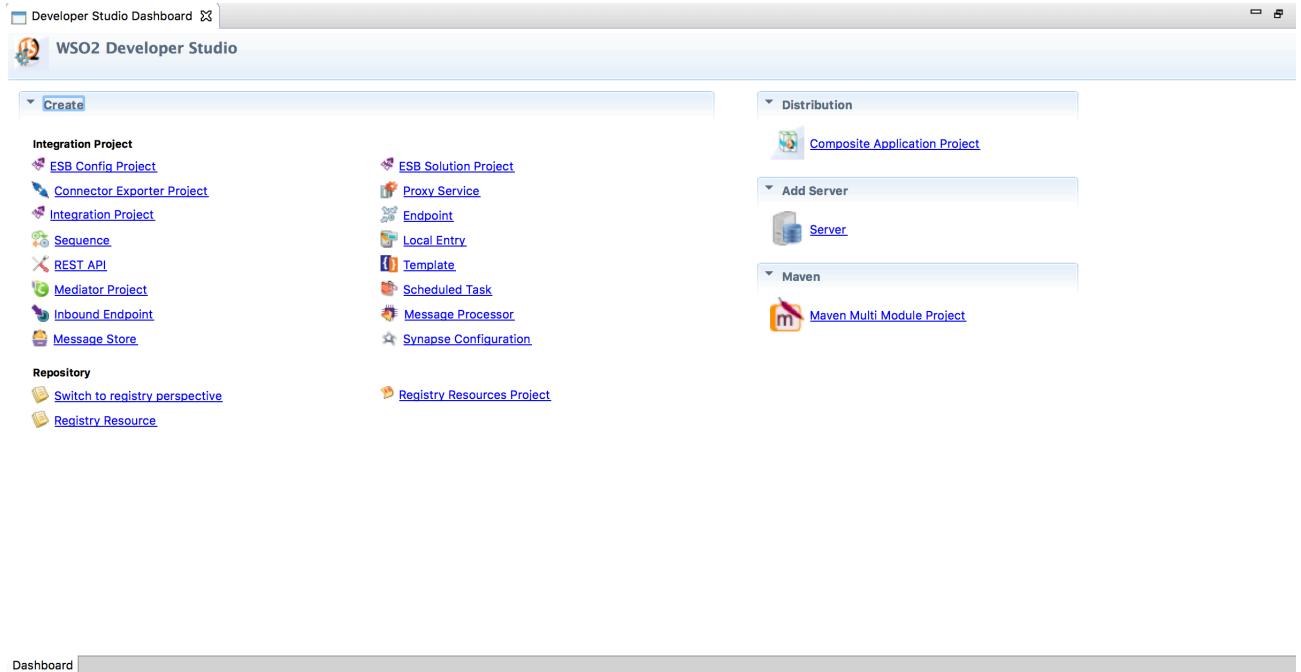
Requests going through the ESB are called messages, and [message mediation](#) is a fundamental part of any ESB. In this section, we will configure message mediation for requests received by the ESB that need to be sent to the HealthcareService back-end service. We will use Eclipse based WSO2 ESB Tooling to create the message mediation artifacts and then deploy them to WSO2 ESB.

See the following topics for a description of the **concepts** that you need to know when creating ESB artifacts:

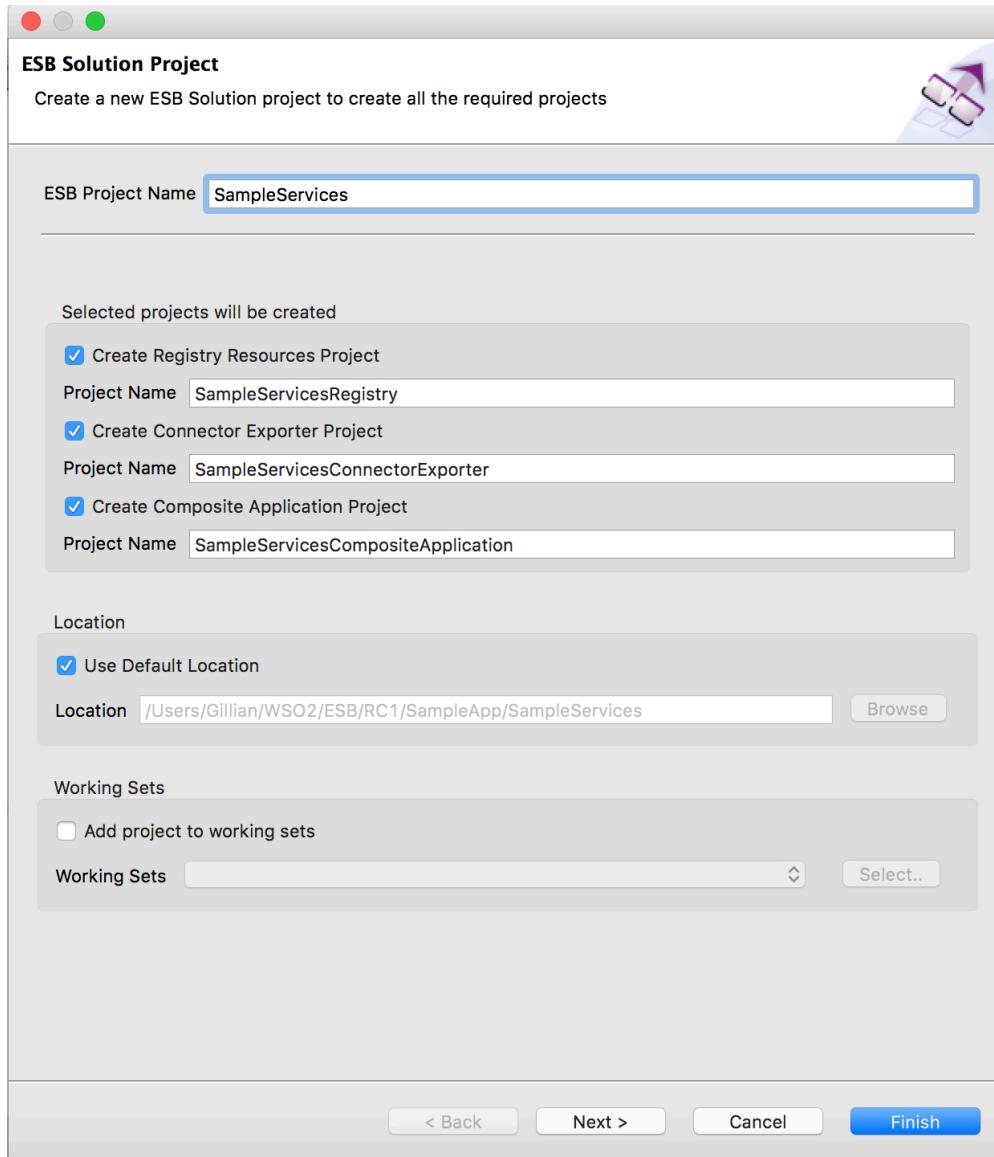
- [REST API](#)
- [Endpoints](#)
- [Sequences](#)
- [API Resource](#)
- [Composite Application Project \(C-App\)](#)

Creating the deployable artifacts project

1. In Eclipse, open the Developer Studio dashboard by clicking the **Developer Studio** menu and choosing **Open Dashboard**.

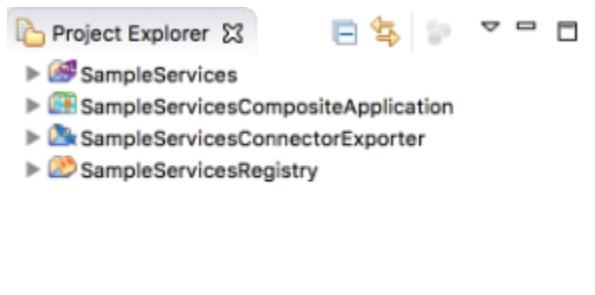


2. Click **ESB Solution Project** and create a project named SampleServices and select the following check boxes so that the relevant projects will be created.
 - Create Registry Resources Project
 - Create Connector Exporter Project
 - Create Composite Application Project



Click **Finish**.

You have now created the following ESB related projects as shown in the Project Explorer:



Next, inside the SampleServices ESB Config project, we will create an endpoint that will allow the ESB to connect to the back-end service.

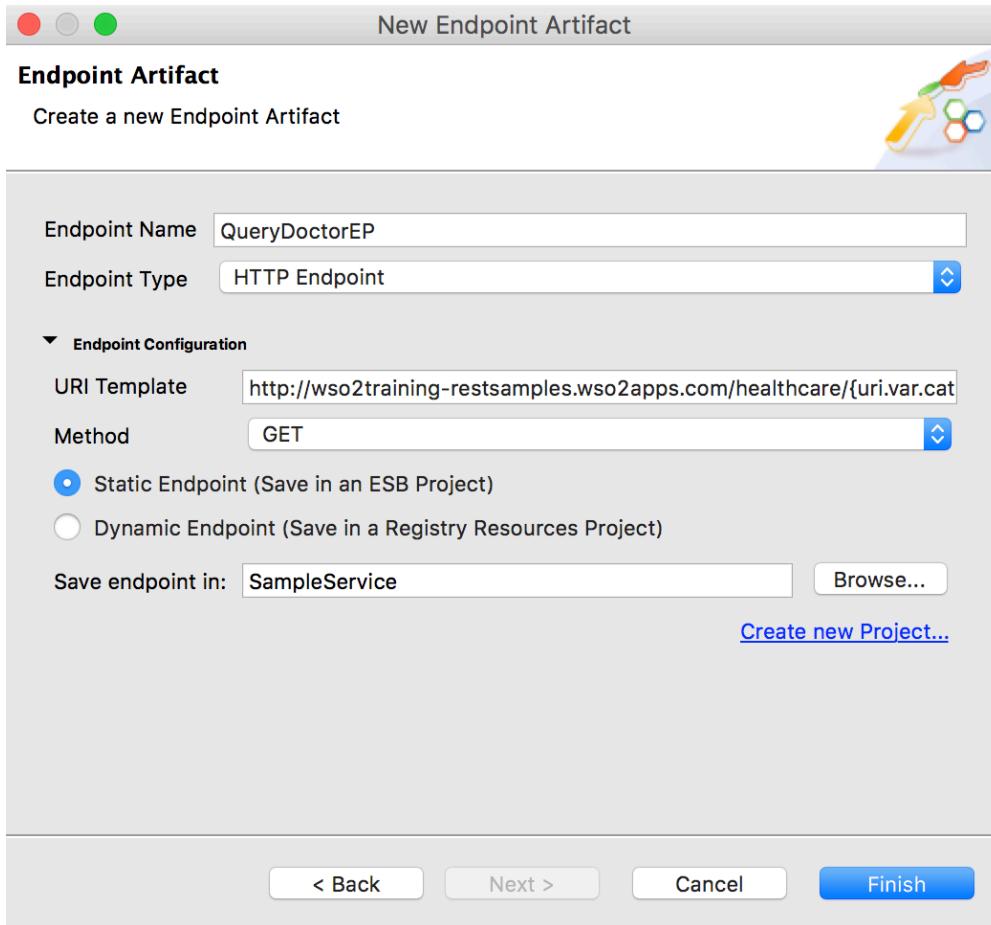
Connecting to the back-end service

To connect to the back-end service (i.e. HealthcareService hosted in WSO2 App Cloud) we must expose a URL that can be used to connect to the service. To do this, we create an endpoint for this service.

The sample back-end service we are using in this tutorial is hosted and available at <http://wso2training-restsamples.wso2app.s.com>.

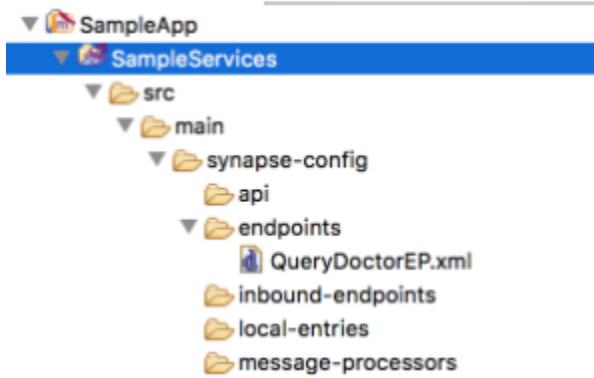
1. Right-click **SampleServices** in the Project Explorer and navigate to **New -> Endpoint**.
2. Ensure **Create a New Endpoint** is selected and click **Next**. Give the information as in the table below:

Field	Value	Description
Endpoint Name	QueryDoctorEP	The name of the endpoint defined.
Endpoint Type	HTTP Endpoint	Indicates that we are connecting to REST back-end service.
URI Template	http://wso2training-restsamples.wso2apps.com/healthcare/{uri.var.category}	The template for the request URL expected by Healthcare back-end service. In this case, the variable 'category' that needs to be included in the request for querying doctors, is represented as {uri.var.category} in the template.
Method	GET	Indicates that we are creating this endpoint for GET requests that are sent to the back-end service.
Static Endpoint		Select this option because we are going to use this endpoint in this ESB project only and will not re-use it in other projects. If you need to create a reusable endpoint, you create it as a Dynamic Endpoint and save the endpoint in either the Configuration or Governance Registry. For more information, see the documentation on registries .
Save Endpoint in	SampleServices	This is the ESB Config project we created in the last section



Click **Finish**.

The QueryDoctorEP endpoint you created is saved in the `endpoints` folder within the ESB Config Project you created.



Now that you have created the endpoint for the back-end service, it's time to create the REST API and the relevant API resource that will receive requests from client applications, mediate them and send them to the endpoint, and return the results to the client.

Mediating requests to the back-end service

We will use WSO2 ESB Tooling to create a REST API named `HealthcareAPI`. We will then create a resource within this API for the GET HTTP method that is used to send requests to the `HealthcareService` back-end service and retrieve available doctor information.

1. In the Project Explorer, right-click **SampleServices** and navigate to **New -> REST API**.

2. Ensure **Create A New API Artifact** is selected and click **Next**.

3. Fill in the information as in the table below:

Field	Value	Description
Name	HealthcareAPI	The name of the REST API in WSO2 ESB
Context	/healthcare	Here we are anchoring the API at "/healthcare" context. This will become part of the name of the generated URL used by the client when sending requests to HealthcareService. For example, setting the context to /healthcare defines that the API will only handle HTTP requests whose URL path starts with <code>http://<host>:<port>/healthcare</code> .
Save location	SampleServices	This is the ESB Config project we have already created previously.

New Synapse API

Synapse API Artifact

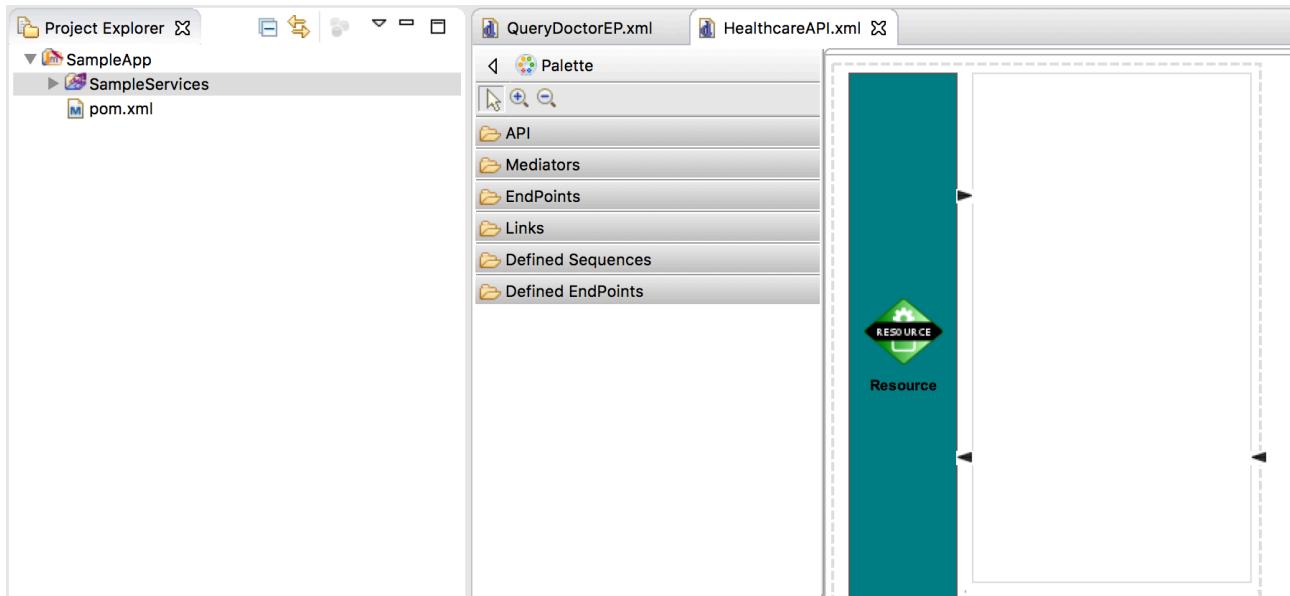
Create a new Synapse API Artifact

Name*	HealthcareAPI
Context*	/healthcare
Hostname	
Port	

Save location:

[Create a new ESB project...](#)

Click **Finish**. Once the API resource is created, the design view of the `HealthcareAPI.xml` file appears so that you can start configuring the API resource.



The top part of the canvas is the **In sequence**, which controls how incoming messages are mediated.

The middle part of the canvas is the **Out sequence**, which controls how responses are handled. In this case, a Send mediator is already in place to send responses back to the requesting client.

The bottom part of the canvas is the **Fault sequence**, which allows you to configure how to handle messages when an error occurs (for more information, see [Error Handling](#)).

4. Click the Resource icon on the left side of the canvas. The properties for the API resource will appear on the Properties tab at the bottom of the window. (If they do not appear, you can right-click the proxy icon and click **Show Properties View**.)
5. On the Properties tab provide the following in the Basic property:
 - Url Style: Click in the Value field, click the down arrow, and then select **URI_TEMPLATE** from the list
 - URI-Template: /querydoctor/{category}
 This defines request URL format. In this case, the full request URL format is `http://<host>:<port>/querydoctor/{category}` where {category} is a variable.

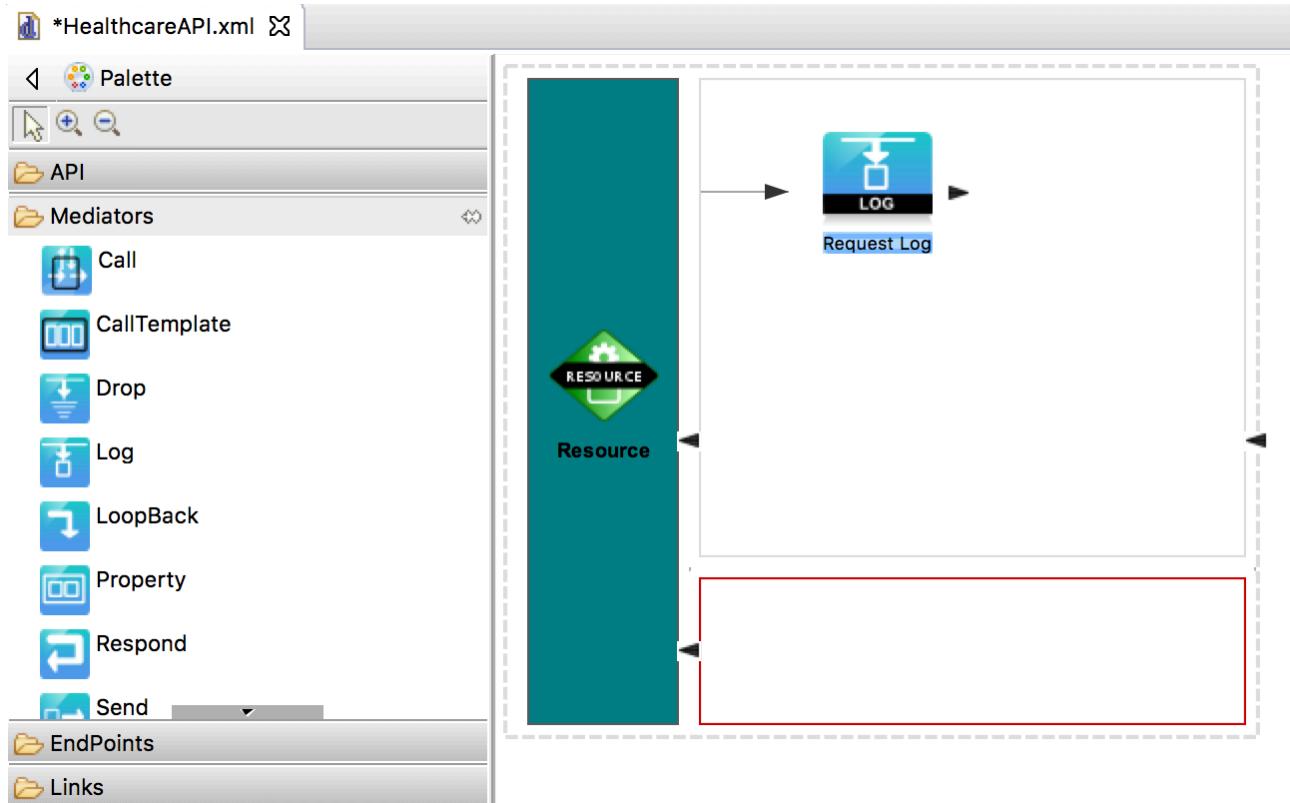
Property	Value
Basic	<ul style="list-style-type: none"> URI_STYLE: URI_TEMPLATE URI-Template: /querydoctor/{category} Protocol: http,https
Fault Sequence	Fault Sequence Type: Anonymous
In Sequence	In Sequence Type: Anonymous

6. In the Methods section of the properties tab, set the value of **Get** to true. This defines that the API resource created handles only requests where the HTTP method is GET.

We are now ready to configure In sequence to handle requests from the client.

You can use the **Log mediator** to log a message when the request is received by the In sequence of the API resource. In this scenario, we will configure the Log mediator display the message "Welcome to the HealthcareService".

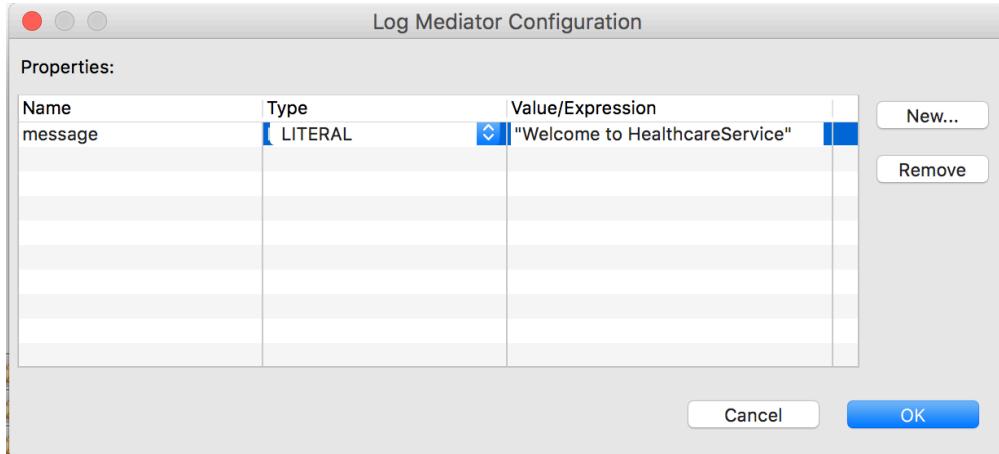
7. From the Mediators palette, click and drag a Log mediator to the In sequence (the top of the canvas).



8. With the Log mediator selected, access the Properties tab and fill in the information in the table below:

Field	Value	Description
Log Category	INFO	Indicates that the log contains an informational message.
Log Level	Custom	When 'Custom' is selected, only specified properties will be logged by this mediator.
Log Separator	(blank)	Since there is only one property that is being logged, we do not require a separator, so this field can be left blank.
Properties		We will add the property in the next step, so skip this for now.
Description	Request Log	The Description field provides the name that appears for the Log mediator icon in the design view.

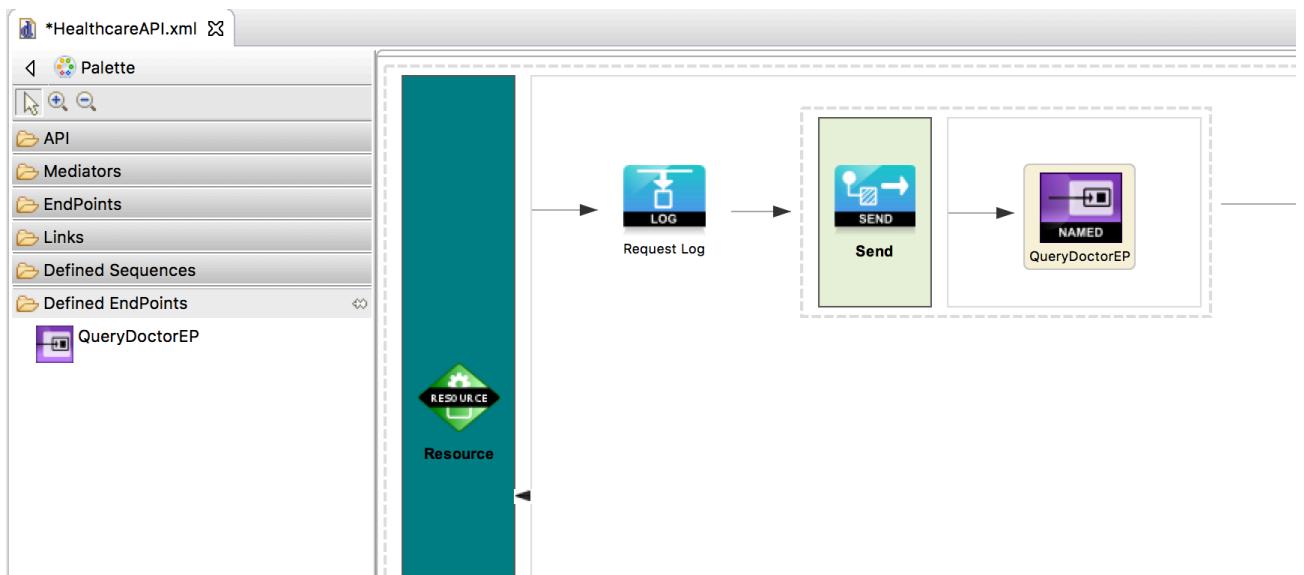
9. We will now add the property that will extract the stock symbol from the request and print a welcome message in the log. Click the Value field of the Properties property, and then click the browse (...) icon that appears.
10. In the Log Mediator Configuration dialog, click **New**, and then add a property called "message" as follows:
- Name: message
 - Type: LITERAL
- We select LITERAL because the required the log message is a static value.
- Value/Expression: "Welcome to HealthcareService"



Click **OK** to save the Log mediator configuration.

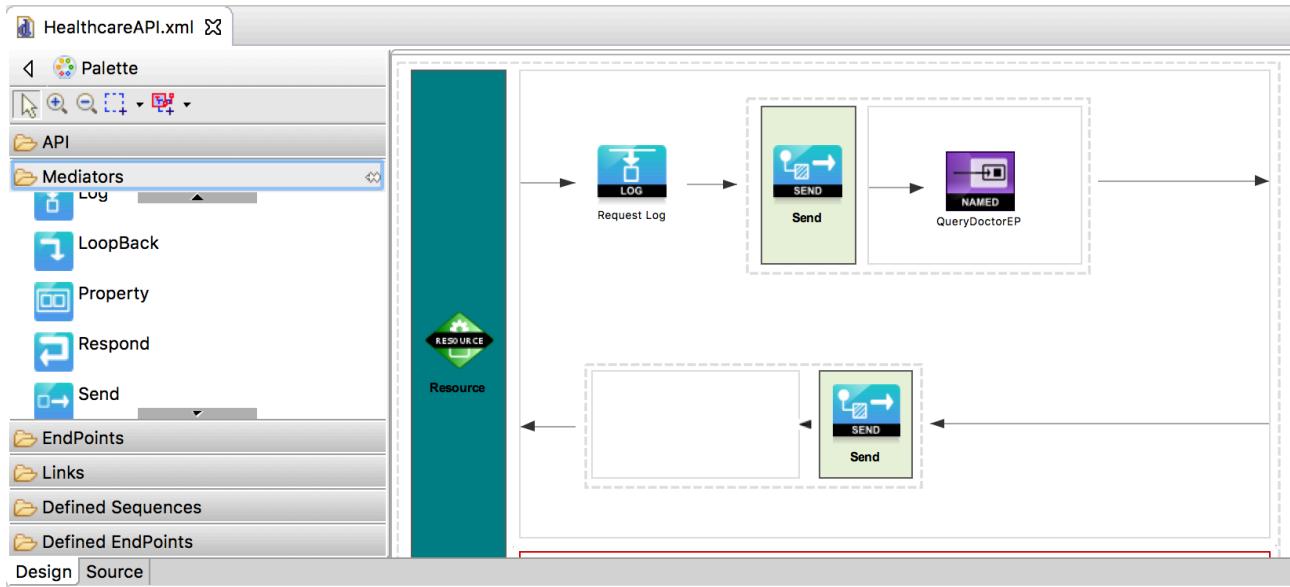
We will now configure the **Send Mediator** to send the request message to the **HealthcareService** endpoint.

11. From the **Mediators palette**, click and drag a **Send mediator** to the **In sequence** adjoining the **Log mediator** you added above. Adjoining this, click and drag the **QueryDoctorEP** we created above the **Defined EndPoints palette**.



The **In Sequence** is now complete. Next, we need to ensure that we send the response from the **HealthcareService** endpoint back to the client. For this, we use a **Send mediator** with no output endpoint defined, which defaults to sending the response back to the requesting client.

12. From the **Mediators palette**, click and drag a **Send mediator** to the **Out Sequence** (the bottom part of the canvas).



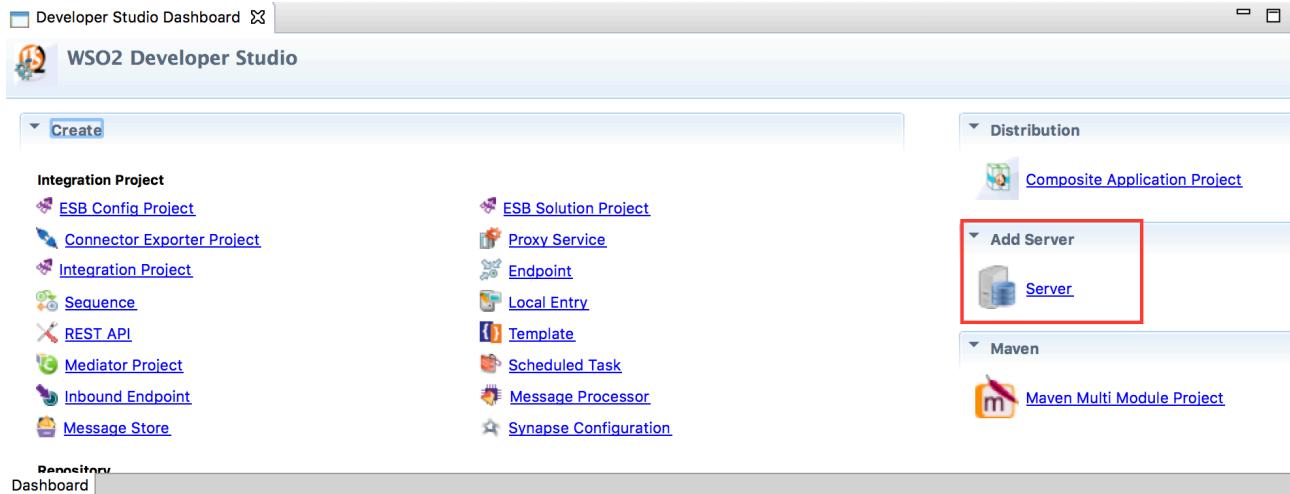
This completes the steps for creating the artifacts required for sending a request through WSO2 ESB to the HealthcareService back-end service. We will now package these artifacts and deploy them to WSO2 ESB.

Deploying the artifacts to WSO2 ESB

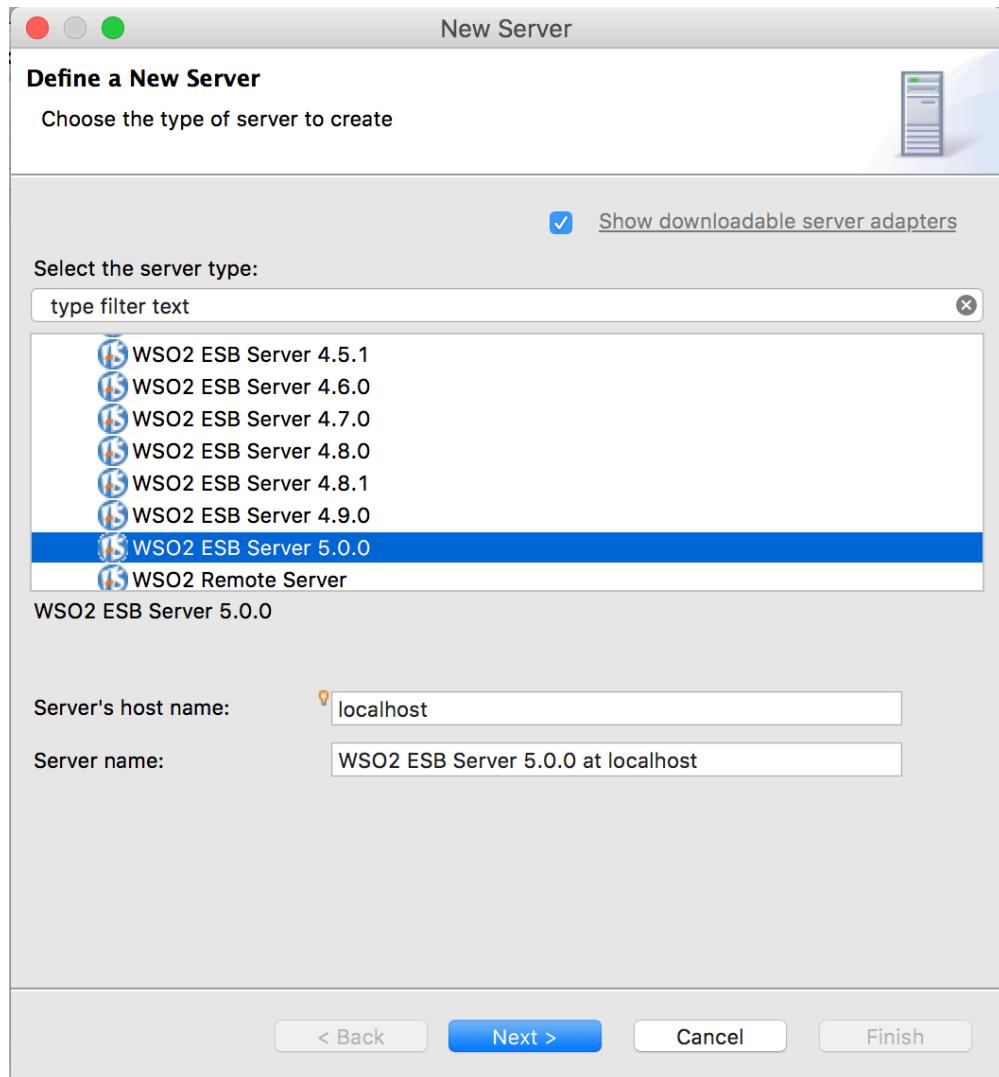
1. Package the QueryDoctorEP endpoint and HealthcareAPI resource into the **Composite Application (C-App)** project named SampleServicesCompositeApplication. Save all changes.

The SampleCApp Composite Application project is generated and is listed under SampleApp project in the Project Explorer. The **Composite Application Project POM Editor** can be accessed by selecting the `pom.xml` file listed under SampleServicesCompositeApplication project.

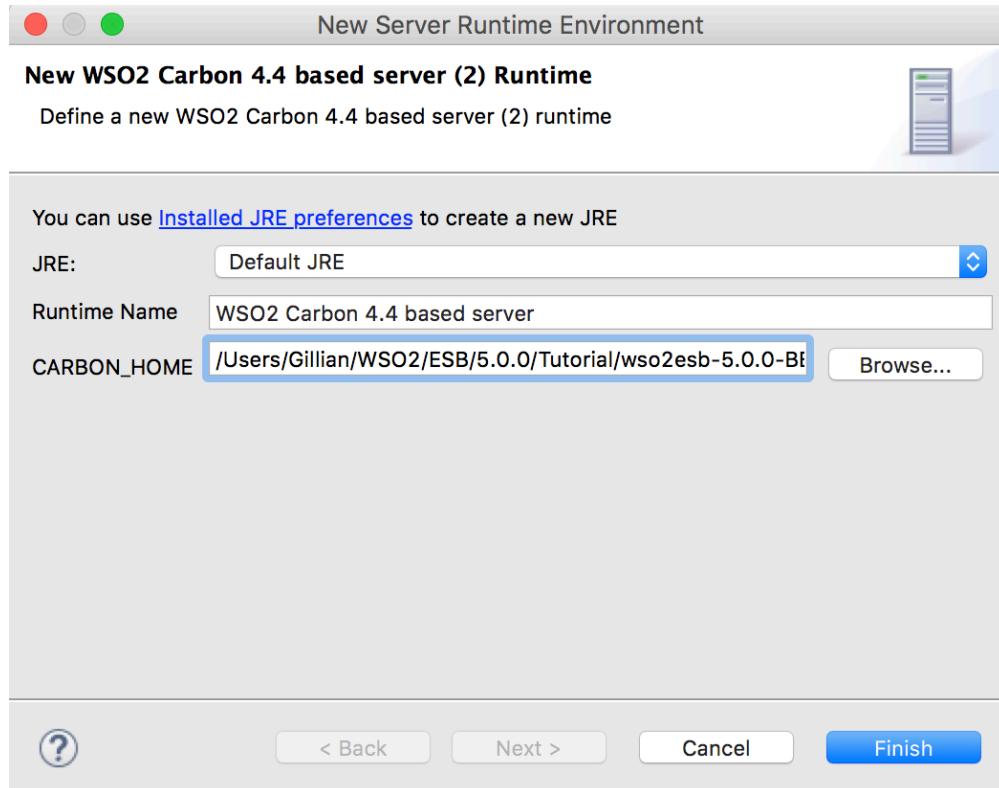
2. In ESB Tooling, navigate to Developer Studio Dashboard and click **Server** under **Add Server**.



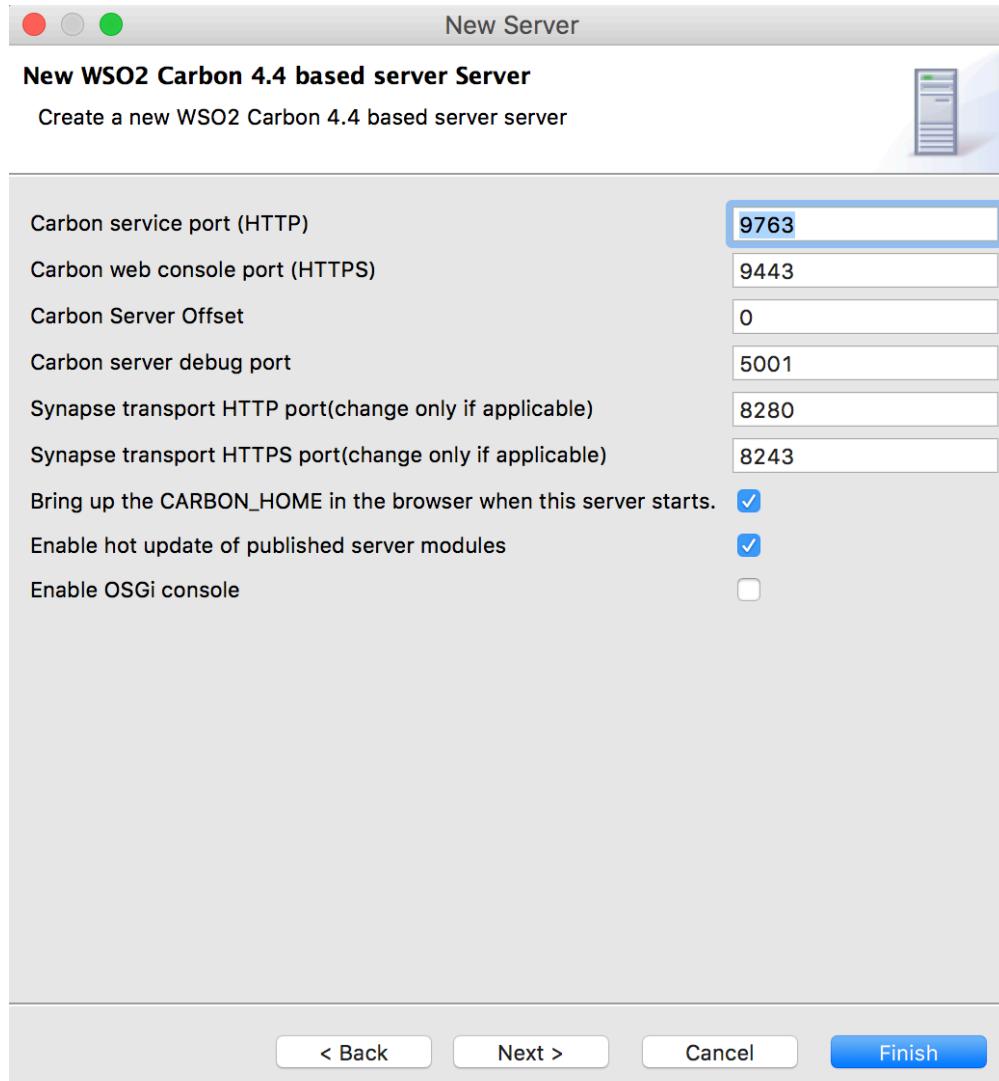
3. In the Define a New Server dialog box, expand the WSO2 folder, and select the version the WSO2 ESB server. Select **WSO2 ESB Server 5.0.0**.



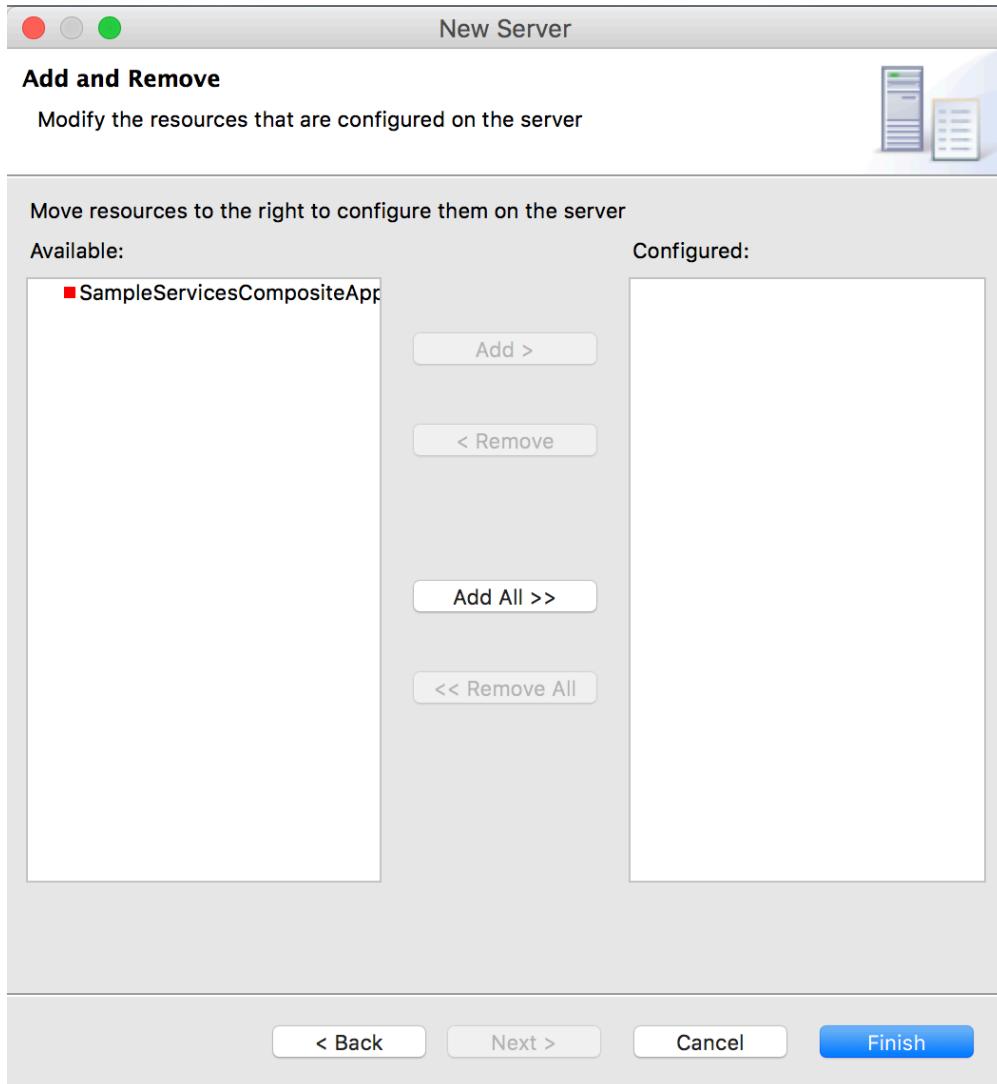
4. Click **Next**, provide the path to your ESB home (the directory where you installed WSO2 ESB) in the CARBON_HOME field, and then click **Next** again.



5. Review the default port details for WSO2 ESB. Typically, you can leave these unchanged, but if you are already running another server on these ports, specify unused ports here. (See [Default Ports of WSO2 Products](#) for more information.) Click **Next**.

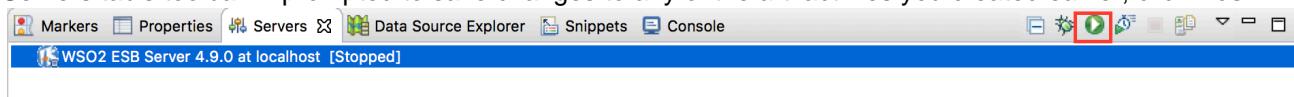


6. To deploy the CApp project to the WSO2 ESB server we just added, select **SampleServicesCompositeApplication** from the list, click **Add** to move it into the Configured list, and then click **Finish**.



The WSO2 ESB server is now added inside Eclipse ESB tooling.

On the Servers tab, you can see that the server is currently stopped. Click the "start the server" icon on the Servers tab's toolbar. If prompted to save changes to any of the artifact files you created earlier, click **Yes**.



As the server starts, the Console tab will appear. You should see messages indicating that SampleCApp_1.0.0 was successfully deployed. The C-App will now be available in the ESB management console in **Manage > Carbon Applications > List**.

[Home](#) > [Manage](#) > [Carbon Applications](#) > [List](#)

Carbon Applications List

1 Running Carbon Applications.

Carbon Applications	Version	Actions
SampleCApp	1.0.0	Delete Download

You can also deploy the artifacts to the ESB server using a [Composite Application Archive \(CAR\)](#) file.

Sending requests to WSO2 ESB

Let's send a request to our REST API, HealthcareAPI, which is now deployed in WSO2 ESB. You will need a REST client like curl for this.

1. In your Web browser, navigate to the WSO2 ESB management console using the following URL: <https://localhost:9443/carbon/>. (If you changed the ports, replace "9443" with the correct HTTPS port.)
2. Log into the management console using the following credentials:
 - Username: admin
 - Password: admin
3. In the left navigation pane, click **APIs** under **Service Bus**. Here you can see that the REST API we created earlier, HealthcareAPI, is available in the ESB and is ready to receive requests and send them to the back-end service. Here you can also view the API Invocation URL that is used in to send the request to the service.

Home > Manage > Service Bus > APIs Help

Deployed APIs

[+ Add API](#)

Search API

Available defined APIs in the Synapse Configuration : 1

Select all in this page | Select none

Select	API Name	API Invocation URL	Action
<input type="checkbox"/>	HealthcareAPI	http://192.168.1.122:8280/healthcare	Enable Statistics Enable Tracing Edit Delete

Select all in this page | Select none

4. Open a command line terminal and enter the following request:

```
curl -v http://localhost:8280/healthcare/querydoctor/surgery
```

This is derived from the [URI-Template defined](#) when creating the API resource.

```
http://<host>:<port>/querydoctor/{category}
```

Other categories you can try sending in the request are:

- cardiology
- gynaecology
- ent
- paediatric

5. You will see the response message from HealthcareService with a list of all available doctors and relevant details.

```
[ {"name": "thomas collins",
  "hospital": "grand oak community hospital",
  "category": "surgery",
  "availability": "9.00 a.m - 11.00 a.m",
  "fee": 7000.0},
 {"name": "anne clement",
  "hospital": "clemency medical center",
  "category": "surgery",
  "availability": "8.00 a.m - 10.00 a.m",
  "fee": 12000.0},
 {"name": "seth mears",
  "hospital": "pine valley community hospital",
  "category": "surgery",
  "availability": "3.00 p.m - 5.00 p.m",
  "fee": 8000.0}]
```

6. Now, check the ESB server Console in Eclipse, and you will see the following message:

```
INFO - LogMediator message = "Welcome to HealthcareService"
```

This is the message printed by the Log mediator when the message from the client is received in the In sequence of the API resource.

You have now created and deployed an API resource in WSO2 ESB that receives requests, logs a message using the Log mediator, sends the request to a back-end service using the Send mediator, and returns a response to the requesting client.

Routing Requests Based on Message Content

The message payload sent by the client to WSO2 ESB for making an appointment reservation will contain the hospital name where the appointment needs to be confirmed. The HTTP request method used will be POST. Based on the hospital name sent in the request message, WSO2 ESB will then route the appointment reservation to the relevant hospital's backend service.

In this tutorial, you use a [Switch mediator](#) to route messages based on the message content to the relevant HTTP Endpoint defined in WSO2 ESB.

For more details on how routing of messages within WSO2 ESB is done based on the message content, refer to [Content-Based Router Enterprise Integration Pattern](#).

See the following topics for a description of the **concepts** that you need to know when creating ESB artifacts:

- [REST API](#)
- [Endpoints](#)
- [API Resource](#)
- [Composite Application Project \(C-App\)](#)

Before you begin,

1. Install Oracle Java SE Development Kit (JDK) version 1.8.* and set the JAVA_HOME environment variable.
2. Go to <http://wso2.com/products/enterprise-service-bus/>, click **DOWNLOAD** to download the ESB

runtime ZIP file, and then extract the ZIP file.

The path to this folder will be referred to as <ESB_HOME> throughout the tutorials.

3. Go to <http://wso2.com/products/enterprise-service-bus/>, click **Tooling** to select and download the relevant ESB tooling ZIP file, and then extract the ZIP file.

The path to this folder will be referred to as <TOOLING_HOME> throughout the tutorials.

For more detailed installation instructions, see the [Installing WSO2 ESB Tooling](#).

4. Open ESB Tooling and [create the deployable artifacts project](#) from the previous tutorial.

Let's get started!

This tutorial contains the following sections:

- [Connecting to the back-end service](#)
- [Mediating requests to the back-end service](#)
- [Deploying the artifacts to WSO2 ESB](#)
- [Sending requests to WSO2 ESB](#)

Connecting to the back-end service

In this tutorial we have three hospital backend services that are hosted for three different hospitals as follows:

- Grand Oak Community Hospital : <http://wso2training-restsamples.wso2apps.com/grandoaks/>
- Clemency Medical Center : <http://wso2training-restsamples.wso2apps.com/clemency/>
- Pine Valley Community Hospital : <http://wso2training-restsamples.wso2apps.com/pinvalley/>

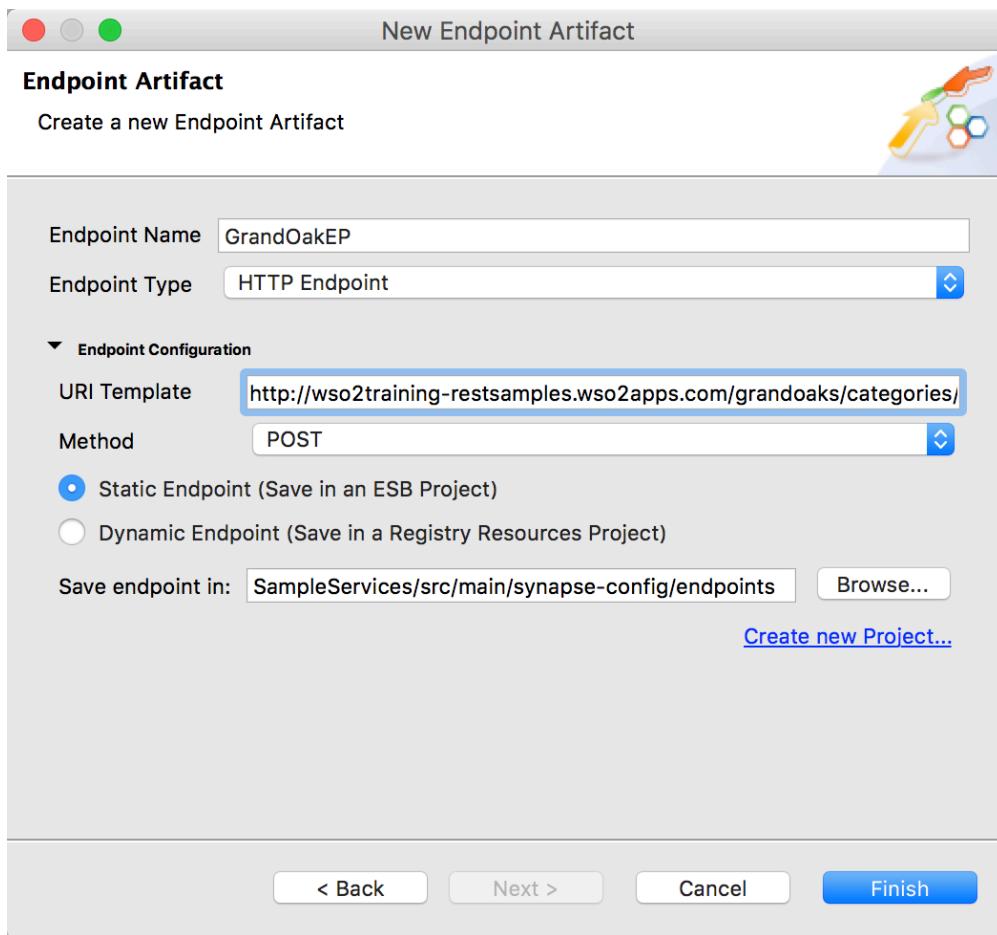
The request method is POST and a sample request URL expected by the backend services is:

<http://wso2training-restsamples.wso2apps.com/grandoaks/categories/{category}/reserve>

Let's now create three different HTTP endpoints for the above services.

1. Right-click **SampleServices** in the Project Explorer and navigate to **New -> Endpoint**. Ensure **Create a New Endpoint** is selected and click **Next**.
2. Fill in the information as in the following table:

Field	Value
Endpoint Name	GrandOakEP
Endpoint Type	HTTP Endpoint
URI Template	http://wso2training-restsamples.wso2apps.com/grandoaks/categories/{uri.var.category}/reserve
Method	POST
Static Endpoint	(Select this option because we are going to use this endpoint in this ESB project only and will not re-use it in other projects.)
Save Endpoint in	SampleServices



Click **Finish**.

3. Create similar HTTP endpoints for the other two hospital services using the relevant URI Template as follows:
 - ClemencyEP : http://wso2training-restsamples.wso2apps.com/clemency/categories/{uri.var.category}/reserve
 - PineValleyEP : http://wso2training-restsamples.wso2apps.com/pinevalley/categories/{uri.var.category}/reserve

You have now created three endpoints for the three hospital backend services that will be used to make appointment reservations.

You can also create a single endpoint where the differentiation of the hospital name can be handled using a variable in the URI template. See the tutorial, [Exposing Several Services as a Single Service](#).

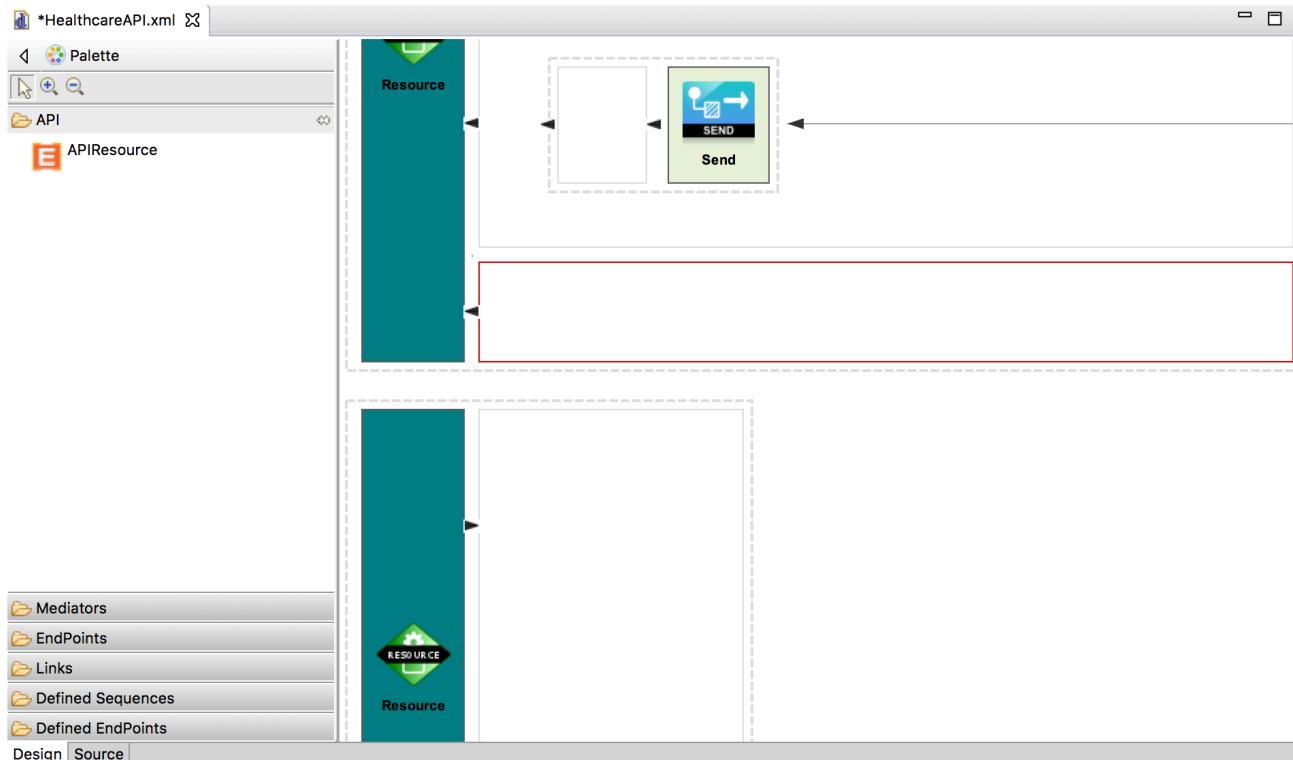
Using three different endpoints is advantageous when the back-end services are very different from one another and/or when there is a requirement to configure error handling differently for each of them.

Mediating requests to the back-end service

To implement the routing scenario, we will update the REST API we created in the previous section by adding a new API resource. We will then use a Switch mediator to route the message to the relevant backend service based on the hospital name passed in the payload of the request message.

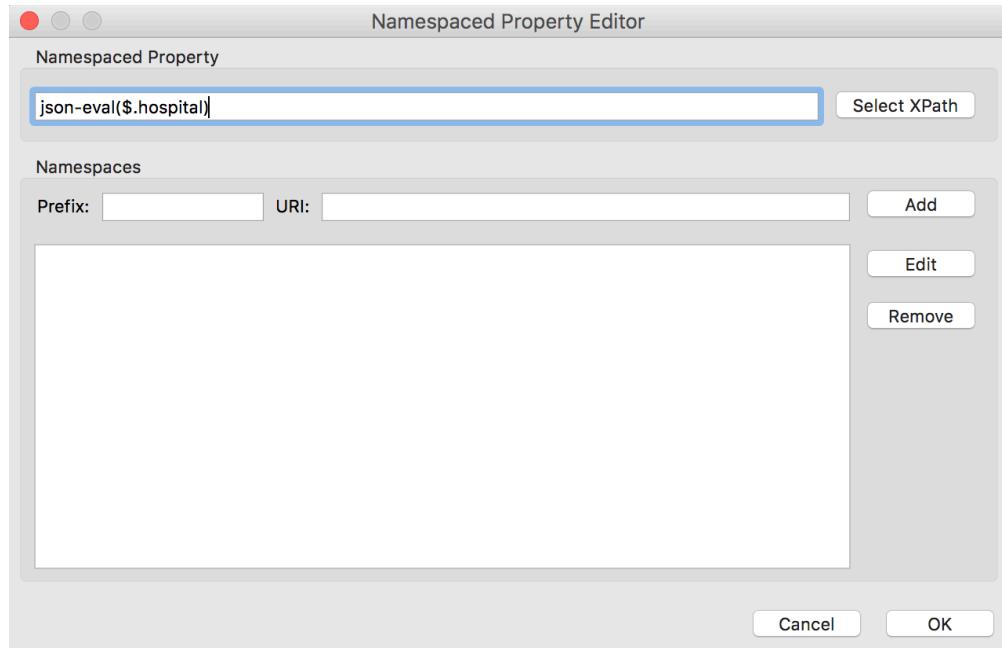
Let's update the REST API we created in the [previous tutorial](#) using WSO2 ESB Tooling.

1. In the REST API configuration, select API Resource in the API palette and drag it onto the canvas just below the previous API resource that was created.

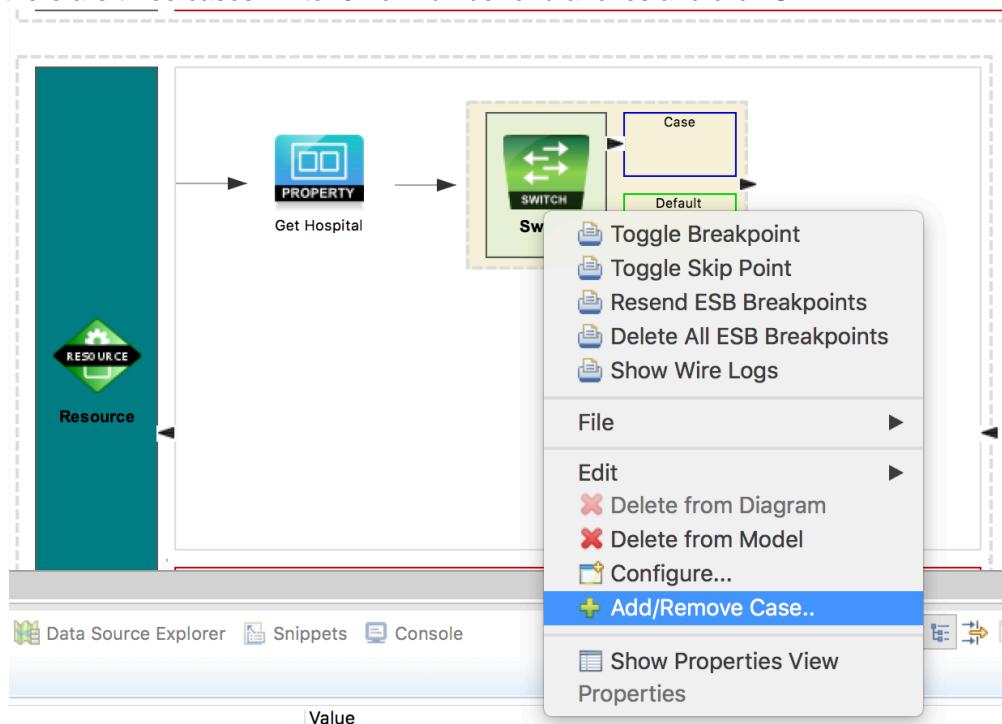


2. With the API Resource you added in the previous step selected, access the Properties tab and fill in the following details:
 - Url Style: Click in the Value field, click the down arrow, and then select **URI_TEMPLATE** from the list
 - URI-Template: /categories/{category}/reserve
3. Set the value of **Post** to true in the Methods section of the properties tab.
4. Drag a **Property Mediator** from the Mediators palette to the In Sequence of the API resource and name it **Get Hospital**. This will be used to extract the hospital name that is sent in the request payload.
5. With the Property mediator selected, access the Properties tab and fill in the following details:
 - Property Name: New Property...
 - New Property Name: Hospital
 - Property Action: set
 - Value Type: Expression
6. We will now add the **JSONPath expression** that will extract the hospital from the request payload. Click value field of Value Expression in the Properties tab and add the following expression:


```
json-eval($.hospital)
```



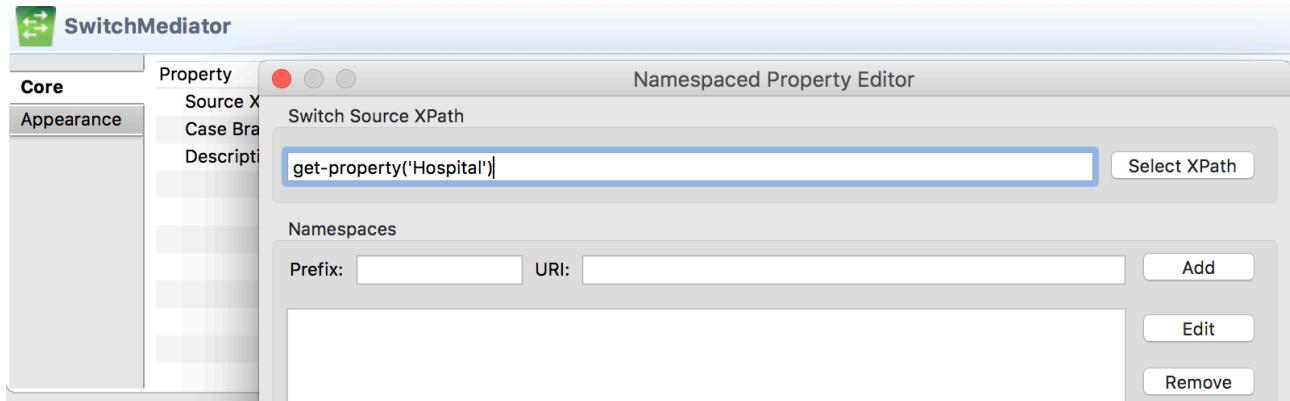
7. Add a Switch mediator from the Mediator palette just after the Property Mediator you added above.
8. Right-click the Switch mediator you just added and select **Add/Remove Case** to add the number of cases you want to specify. In this scenario, we are assuming there are three different hospitals, hence there are three cases. Enter 3 for Number of branches and click **OK**.



The screenshot shows the 'Add Case Branches' dialog box. It has a title bar 'Add Case Branches.', a text input field 'Number of branches:' containing the value '3', and two buttons at the bottom: 'Cancel' and 'OK'.

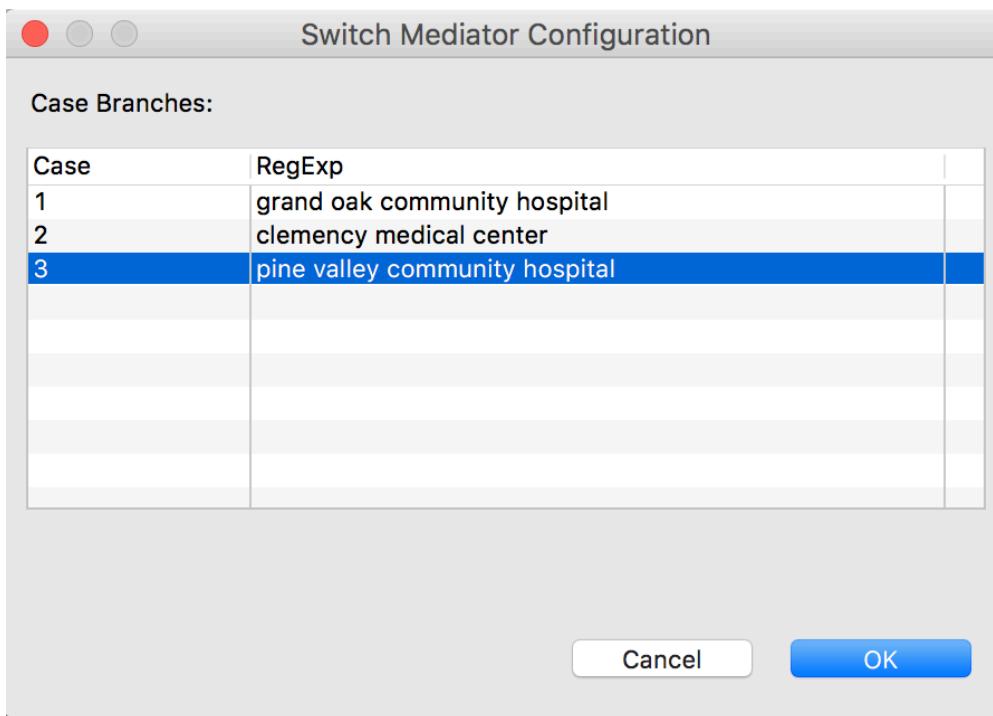
9. With the Switch mediator selected, go to the Properties tab.
10. The Source XPath field is where we will specify the XPath expression that obtains the value of Hospital that we stored in the Property mediator. To specify the expression, click in the Value field of the Source XPath property, click the browse (...) button, and then overwrite the default expression with the following and click **OK**:

```
get-property('Hospital')
```



For more information on `get-property()`, see [XPath Extension Functions](#).

11. Click in the Value field of the Case Branches property, click the browse (...) button, and then change the RegExp value as follows:
 - Case 1: grand oak community hospital
 - Case 2: clemency medical center
 - Case 3: pine valley community hospital

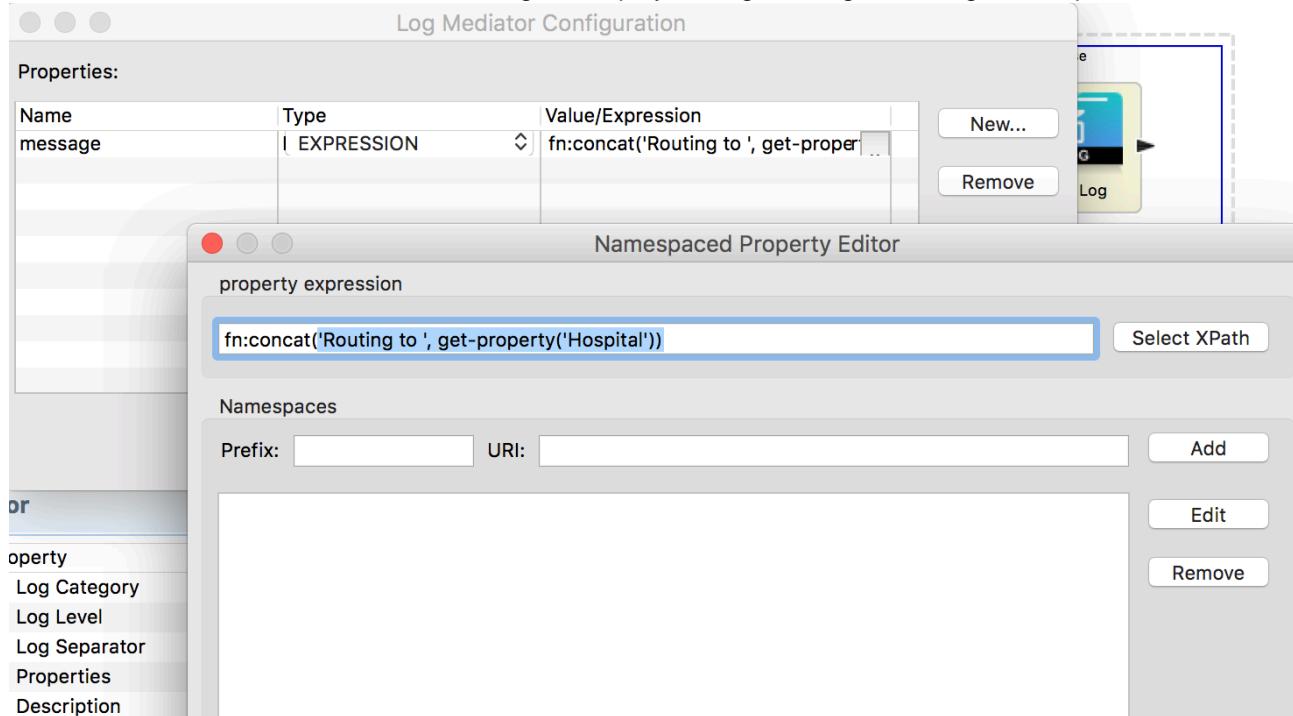


Click **OK**.

12. Let's add a Log mediator to print a message indicating to which hospital the request message is being routed. Drag a Log mediator to the first Case box of the Switch mediator, name it 'Grand Oak Log', access the Properties tab and enter the following:
 - Log Category: INFO

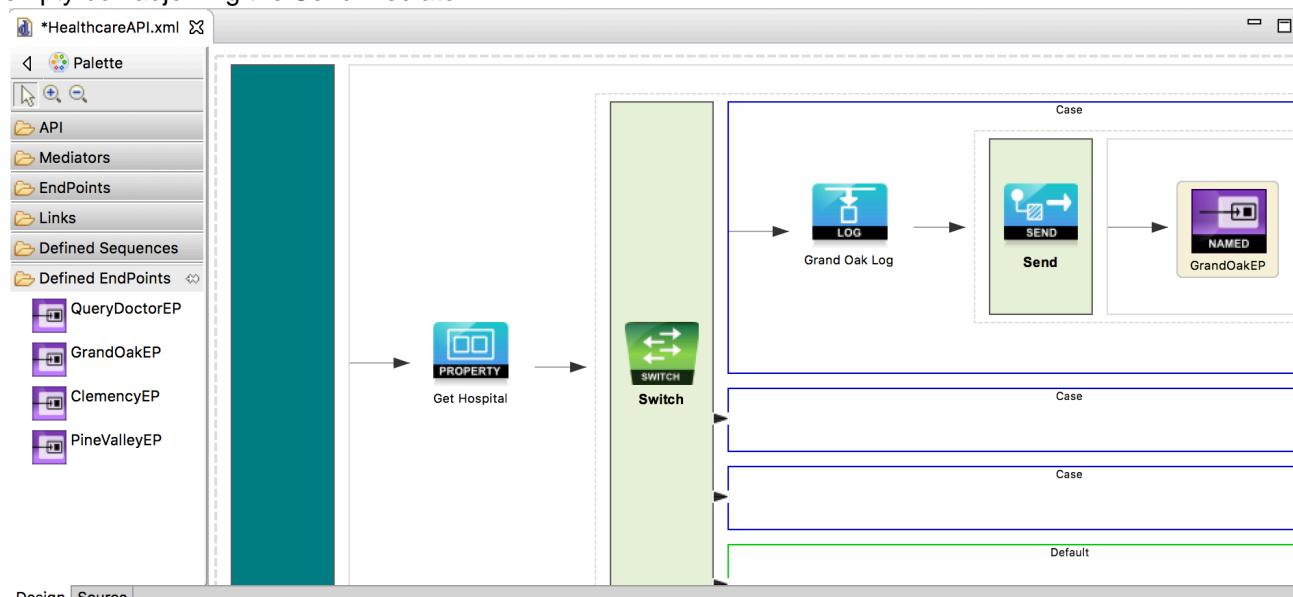
- Log Level: CUSTOM
13. Click the Value field of the Properties property, and then click the browse (...) icon that appears. In the Log Mediator Configuration dialog box, click **New**, and then add a property called 'message' as follows:
- Name: message
 - Type: EXPRESSION
- We select EXPRESSION because the required properties for the log message must be extracted from the request, which we can do using an XPath expression.
- Value/Expression: Click the browse (...) icon in the Value/Expression field and enter the following:
 - Property expression: `fn:concat('Routing to ', get-property('Hospital'))`

This is an XPath expression value that uses the value stored in the Property mediator and will then concatenate the two strings to display the log message "Routing to <hospital name>".



Click **OK**.

14. Add a Send mediator adjoining the Log mediator and add GrandOakEP from Defined Endpoints palette to the empty box adjoining the Send mediator.



15. Add Log mediators in the other two Case boxes in Switch mediator and then enter the same properties as in

Step 13. Name the two Log mediators as 'Clemency Log' and 'Pine Valley Log'.

Add Send mediators adjoining these log mediators and respectively add endpoints ClemencyEP and PineValleyEP from the Defined Endpoints palette.

You have now configured the Switch mediator so that when a request is sent to this API resource, the message "Routing to <Hospital Name>" will be logged. The request message will then be routed to the relevant hospital backend service based on the hospital that is sent in the request payload.

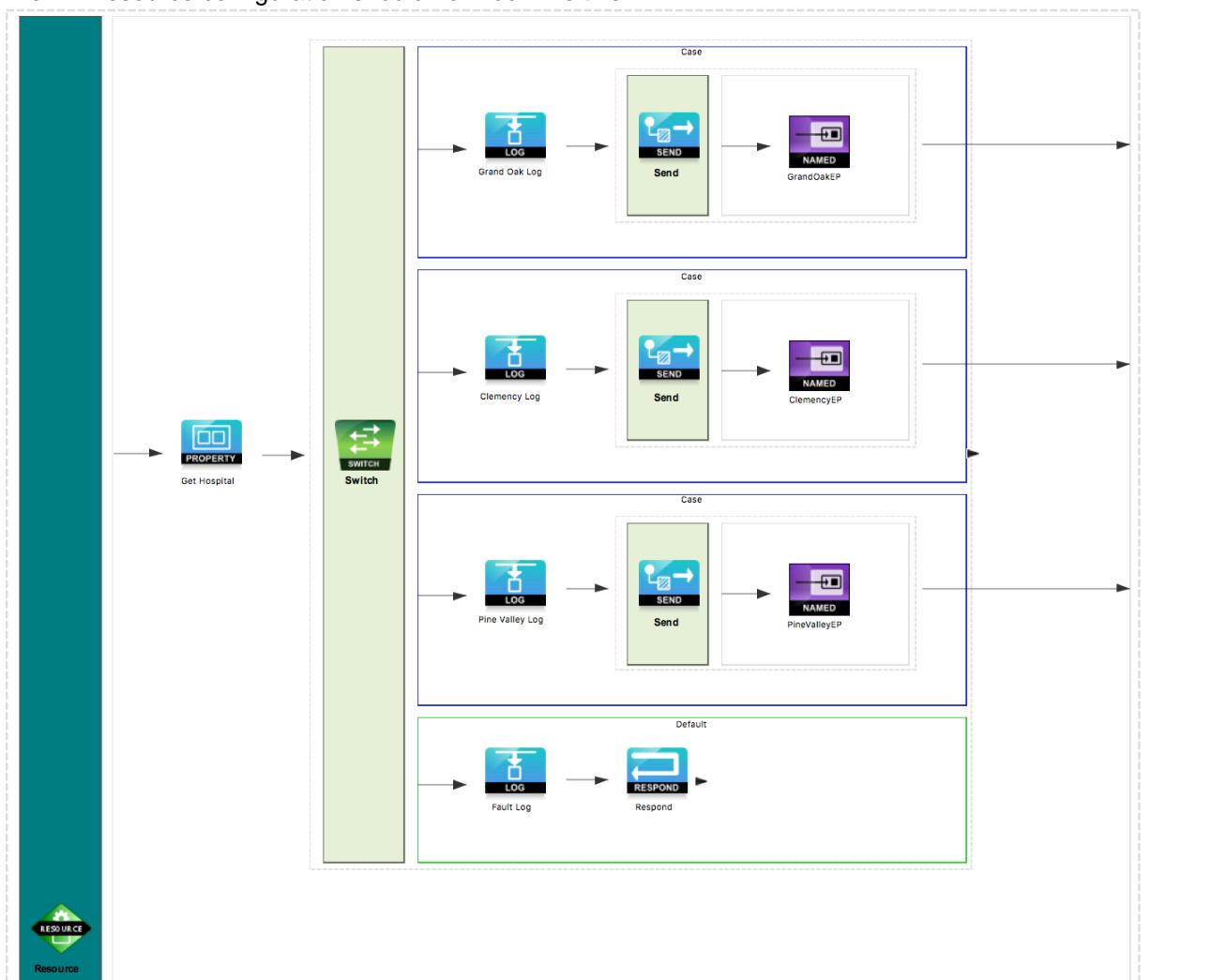
- Let's now configure the default case in the switch mediator. This will handle requests where an invalid hospital is sent in the request payload. Add a Log mediator to the Default (the bottom box) of the Switch mediator and configure it the same way you did for the Log mediator above, this time naming it **Fault Log** and changing its Value/Expression as follows:

```
fn:concat('Invalid hospital - ', get-property('Hospital'))
```

This results in the message "Invalid hospital - <Hospital Name>" to be logged for requests where the request payload contains an invalid hospital.

- Drag a Respond mediator adjoining the Log mediator you added in the above step. This ensures that there is no further processing of the current message and returns the request message back to the client.

The API resource configuration should now look like this:



- Drag a Send mediator to the Outsequence of the API resource to send the response back to the client.
- Save the updated REST API configuration.

Deploying the artifacts to WSO2 ESB

- Package the C-App names SampleServicesCompositeApplication project with the artifacts created.

Ensure the following artifact check boxes are selected in the **Composite Application Project POM Editor**.

- HealthcareAPI
- ClemencyEP
- GrandOakEP
- PineValleyEP

2. Assuming you have already added a server in Eclipse, on the Servers tab, expand the WSO2 Carbon server, right-click **SampleServicesCompositeApplication**, and choose **Redeploy**. The Console window will indicate that the CApp has been deployed successfully.

If you do not have a server added in Eclipse, refer [this tutorial](#).

You can also deploy the artifacts to the ESB server using a Composite Application Archive (CAR) file.

Sending requests to WSO2 ESB

Let's send a request to the API resource to make a reservation.

1. Create a JSON file names `request.json` with the following request payload.

```
{
  "patient": {
    "name": "John Doe",
    "dob": "1940-03-19",
    "ssn": "234-23-525",
    "address": "California",
    "phone": "8770586755",
    "email": "johndoe@gmail.com"
  },
  "doctor": "thomas collins",
  "hospital": "grand oak community hospital"
}
```

You can also try using any of the following paramters in your request payload.

For hospital:

- clemency medical center
- pine valley community hospital

Doctor Names:

- thomas collins
- henry parker
- abner jones
- anne clement
- thomas kirk
- cailen cooper
- seth mears

- emeline fulton
- jared morris
- henry foster

2. Open a command line terminal and execute the following command from the location where `request.json` file you created is saved:

```
curl -v -X POST --data @request.json http://localhost:8280/healthcare/categories/surgery/reserve --header "Content-Type:application/json"
```

This is derived from the [URI-Template defined](#) when creating the API resource.

`http://<host>:<port>/categories/{category}/reserve`

You will see the following response:

```
{
  "appointmentNumber":1,
  "doctor":
    {
      "name":"thomas collins",
      "hospital":"grand oak community hospital",
      "category":"surgery","availability":"9.00 a.m - 11.00 a.m",
      "fee":7000.0
    },
  "patient":
    {
      "name":"John Doe",
      "dob":"1990-03-19",
      "ssn":"234-23-525",
      "address":"California",
      "phone":"8770586755",
      "email":"johndoe@gmail.com"
    },
  "fee":7000.0,
  "confirmed":false}
```

Now check the Console tab in Eclipse and you will see the following message:

```
INFO - LogMediator message = Routing to grand oak community hospital
```

This is the message printed by the Log mediator when the message from the client is routed to the relevant endpoint in the Switch mediator.

You have seen how requests received by WSO2 ESB can be routed to the relevant endpoint using the Switch mediator.

Transforming Message Content

Message transformation is necessary when the message format sent by the client is different from the message format expected by the back-end service. The [Message Translator](#) architectural pattern in WSO2 ESB describes how to translate from one data format to another.

In this tutorial, you send a request message to a back-end service where the payload is in a different format when compared to the request payload expected by the back-end service. [Data Mapper mediator](#) in WSO2 ESB is used to transform the request message payload to the format expected by the back-end service.

See the following topics for a description of the **concepts** that you need to know when creating ESB artifacts:

- REST API
- Endpoints
- Sequences
- API Resource
- [Composite Application Project \(C-App\)](#)

Before you begin,

1. Install Oracle Java SE Development Kit (JDK) version 1.8.* and set the JAVA_HOME environment variable.
 2. Go to <http://wso2.com/products/enterprise-service-bus/>, click **DOWNLOAD** to download the ESB runtime ZIP file, and then extract the ZIP file.
The path to this folder will be referred to as <ESB_HOME> throughout the tutorials.
 3. Go to <http://wso2.com/products/enterprise-service-bus/>, click **Tooling** to select and download the relevant ESB tooling ZIP file, and then extract the ZIP file.
The path to this folder will be referred to as <TOOLING_HOME> throughout the tutorials.
- For more detailed installation instructions, see the [Installing WSO2 ESB Tooling](#).
4. Open the ESB Tooling environment and click **File -> Import**. Then, select **Existing WSO2 Projects into workspace** under the WSO2 category, click **Next** and upload the [pre-packaged C-App project](#). This C-App contains the configurations of the [previous tutorial](#) so that you do not have to repeat those steps.

Let's assume this is the format of the request sent by the client:

```
{
  "name": "John Doe",
  "dob": "1940-03-19",
  "ssn": "234-23-525",
  "address": "California",
  "phone": "8770586755",
  "email": "johndoe@gmail.com",
  "doctor": "thomas collins",
  "hospital": "grand oak community hospital"
}
```

However, the format of the message must be as follows to be compatible with the backend service.

```
{
  "patient": {
    "name": "John Doe",
    "dob": "1990-03-19",
    "ssn": "234-23-525",
    "address": "California",
    "phone": "8770586755",
    "email": "johndoe@gmail.com"
  },
  "doctor": "thomas collins",
  "hospital": "grand oak community hospital"
}
```

The client message format must be transformed to the back-end service message format within the In sequence.

Let's get started!

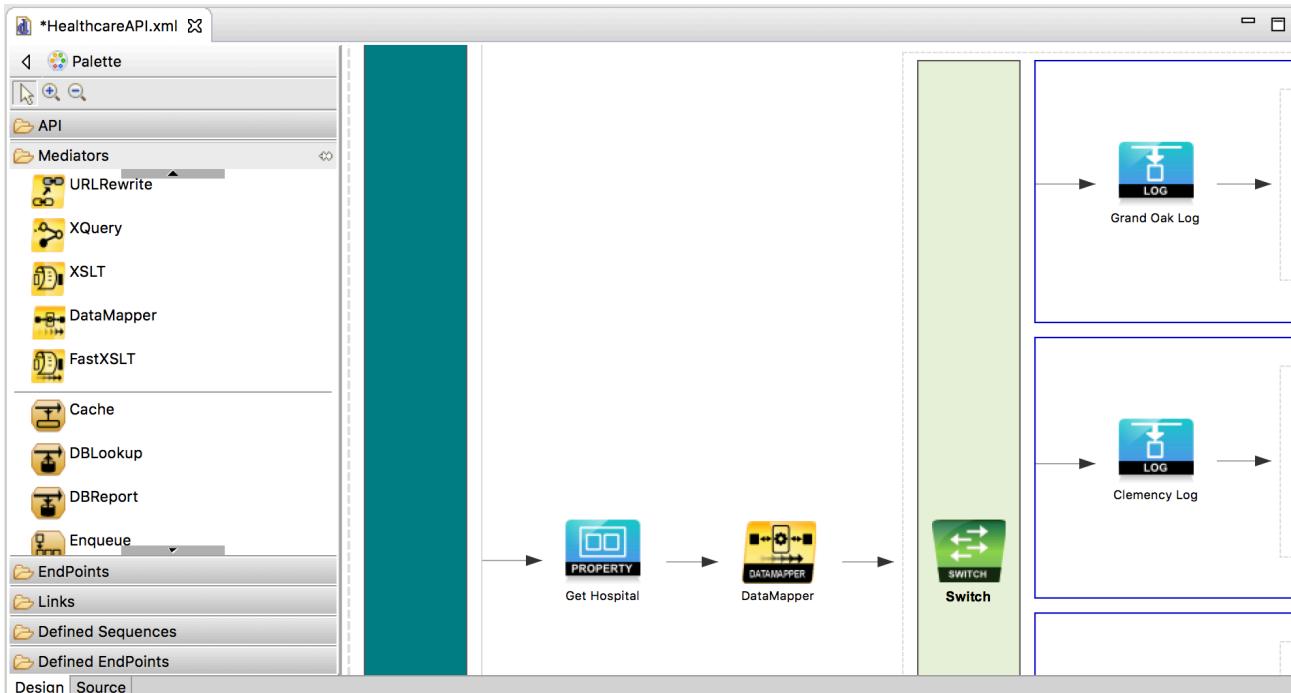
This tutorial contains the following sections:

- Creating the deployable artifacts
- Deploying the Artifacts to WSO2 ESB
- Sending requests to WSO2 ESB

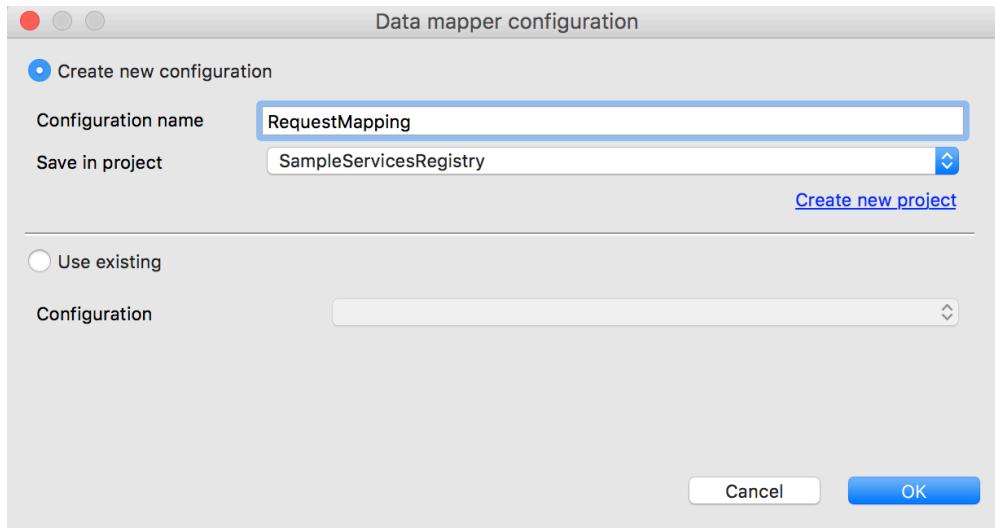
Creating the deployable artifacts

The Data Mapper mediator transforms the message within the In sequences. The Data Mapper mediator is a data mapping solution that can be integrated into a mediation sequence. It converts and transforms one data format to another, or changes the structure of the data in a message.

1. In WSO2 ESB Tooling, add a Data Mapper mediator just after the Property mediator in the In sequence of the API resource.

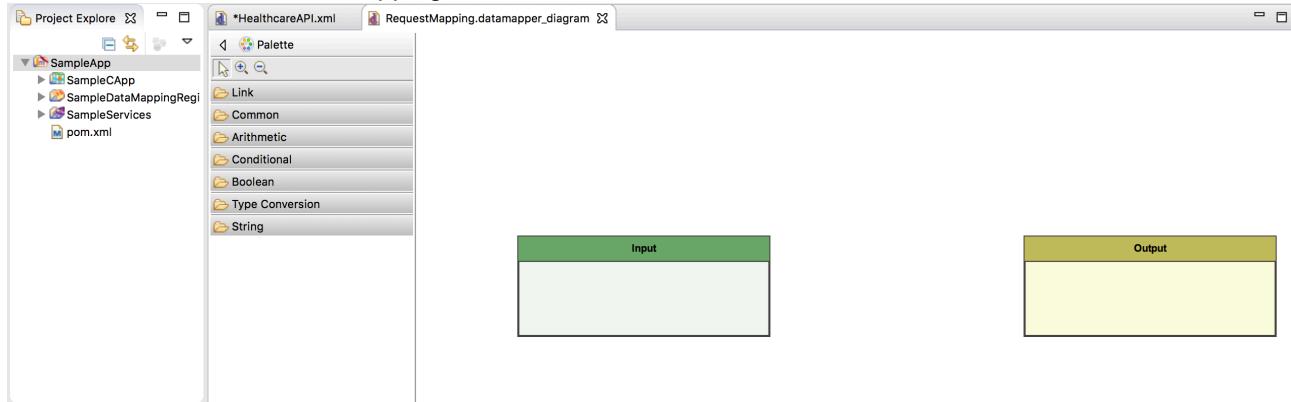


2. Double click the Data Mapper mediator icon and provide the following name for the data mapping configuration file that will be created.
 - Configuration Name: RequestMapping



The **SampleServicesRegistry** project is created at the time of creating the ESB Solution project and will get auto picked here.

Click **OK**. You view the data mapping editor.

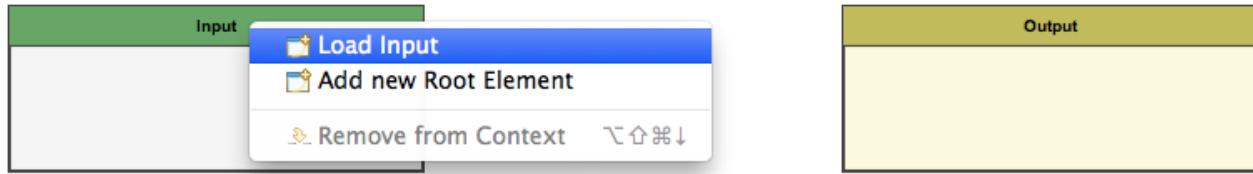


3. Create a JSON file by copying the following sample content of the request message sent to the API Resource, and save it in your local file system.

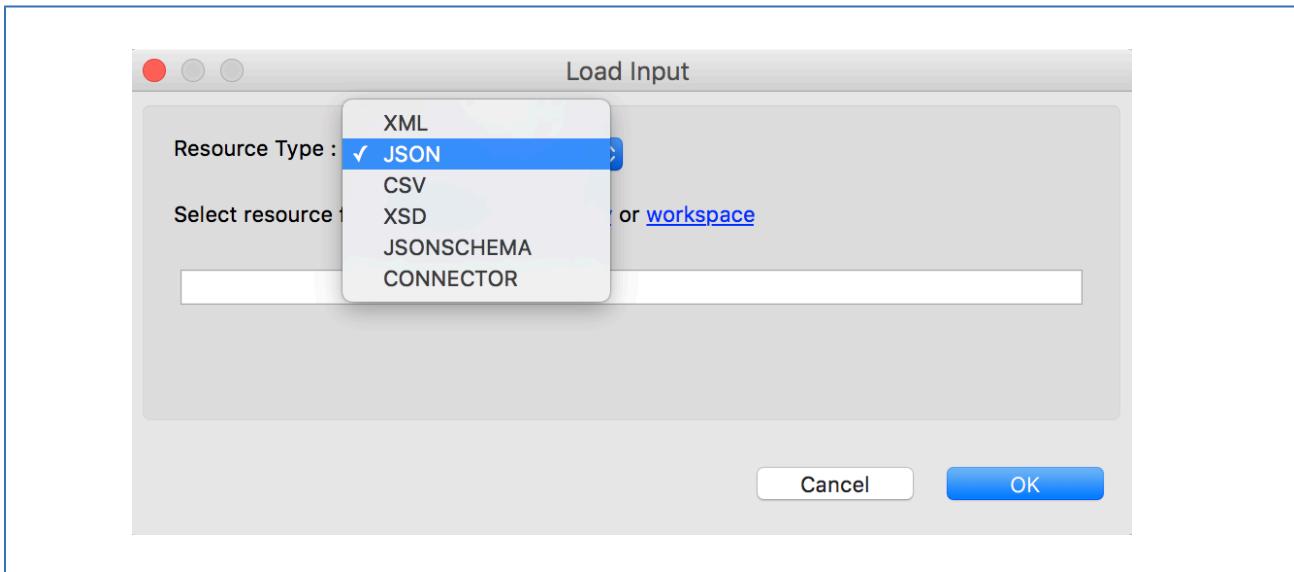
```
{
  "name": "John Doe",
  "dob": "1990-03-19",
  "ssn": "234-23-525",
  "address": "California",
  "phone": "8770586755",
  "email": "johndoe@gmail.com",
  "doctor": "thomas collins",
  "hospital": "grand oak community hospital"
}
```

You can [create a JSON schema manually](#) for input and output using the Data Mapper Diagram editor.

4. Right-click on the top title bar of the **Input** box and click **Load Input** as shown below .



5. Select **JSON** as the **Resource Type** as shown below.



6. Click the **file system** link in **Select resource from**, select the JSON file you saved in your local file system in step 5, and click **Open**.

You view the input format loaded in the **Input** box in the editor as shown below.

```

Input

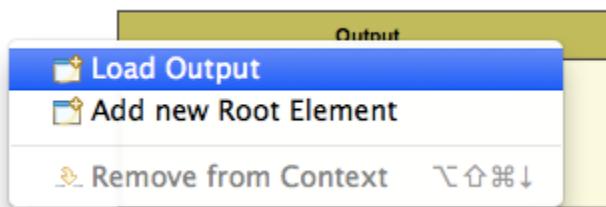
{} root
<> name : [STRING]
<> dob : [STRING]
<> ssn : [STRING]
<> address : [STRING]
<> phone : [STRING]
<> email : [STRING]
<> doctor : [STRING]
<> hospital : [STRING]

```

7. Create another JSON file by copying the following sample content of the request message expected by the backend service, and save it in your local file system.

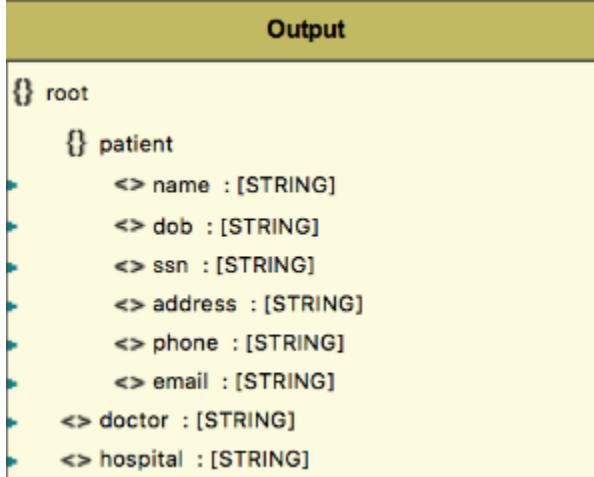
```
{
  "patient": {
    "name": "John Doe",
    "dob": "1990-03-19",
    "ssn": "234-23-525",
    "address": "California",
    "phone": "8770586755",
    "email": "johndoe@gmail.com"
  },
  "doctor": "thomas collins",
  "hospital": "grand oak community hospital"
}
```

8. Right-click on the top title bar of the **Output** box and click **Load Output** as shown below.



9. Select **JSON** as the resource type
 10. Click the **file system** link in **Select resource from**, select the JSON file you saved in your local file system in **step 7**, and click **Open**.

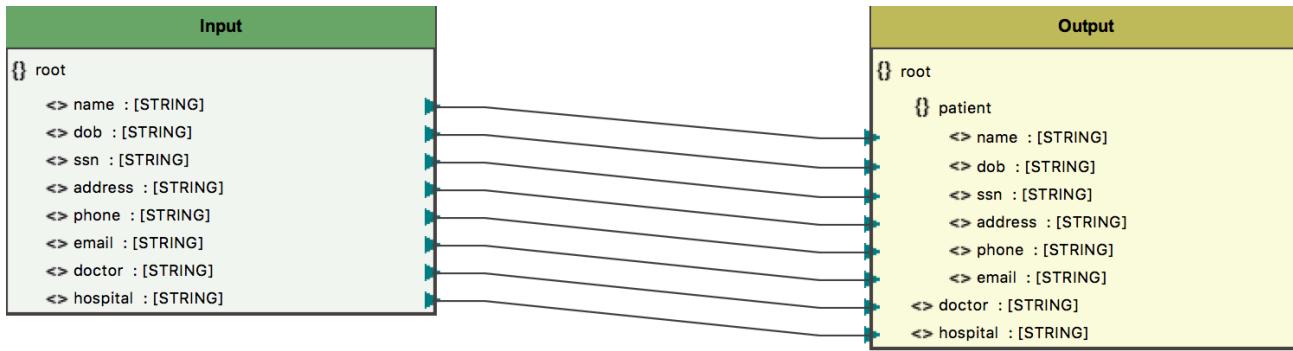
You view the input format loaded in the **Output** box in the editor as shown below.



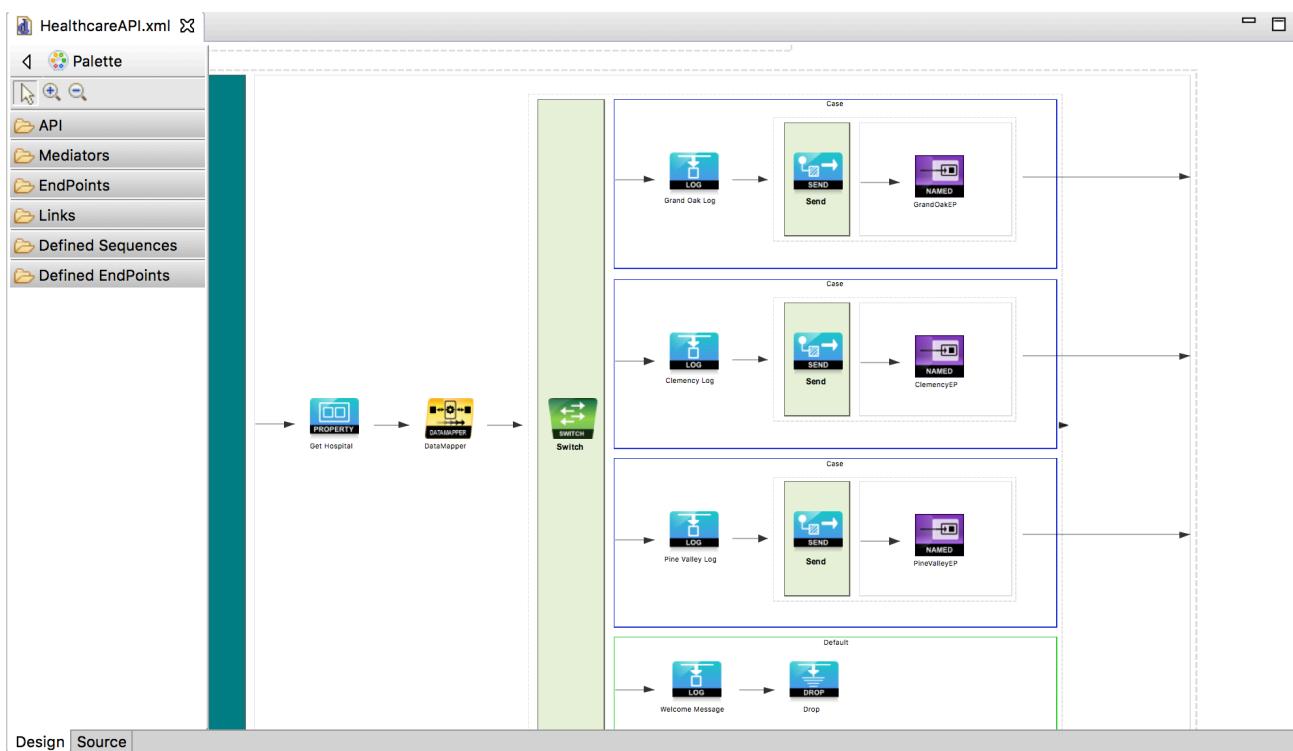
Check the **Input** and **Output** boxes with the sample messages, to see if the element types (i.e. (Arrays, Objects and Primitive values) are correctly identified or not. Following signs will help you to identify them correctly.

- {} - represents object elements
- [] - represents array elements
- <> - represents primitive field values
- A - represents XML attribute value

11. Do the mapping by dragging arrows from field values in the input box to the relevant field values in the output box. The final mapping is as follows:



12. Save and close the configuration.
13. Go back to the **Design View** of the API Resource and select the Data Mapper Mediator and edit the following in the **Properties** tab:
 - Input Type: JSON
 - Output Type: JSON
14. Save the REST API configuration.



We are now ready to package and deploy the C-App and send the request.

Deploying the Artifacts to WSO2 ESB

Since we created a new Registry Resource project, this will need to be packaged into our existing C-App.

1. Package the C-App names **SampleServicesCompositeApplication** project with the artifacts created.

Ensure the following artifact check boxes are selected in the **Composite Application Project POM Editor**.

- SampleServices
 - HealthcareAPI
 - ClemencyEP
 - GrandOakEP
 - PineValleyEP

- SampleServicesRegistry
- On the Servers tab, expand the WSO2 Carbon server, right-click **SampleServicesCompositeApplication**, and choose **Redeploy**. The Console window will indicate that the CApp has been deployed successfully.

If you do not have a server added in Eclipse, refer [this tutorial](#).

You can also deploy the artifacts to the ESB server using a Composite Application Archive (CAR) file.

Sending requests to WSO2 ESB

- Create a JSON file names `request.json` with the following request payload.

```
{
  "name": "John Doe",
  "dob": "1990-03-19",
  "ssn": "234-23-525",
  "address": "California",
  "phone": "8770586755",
  "email": "johndoe@gmail.com",
  "doctor": "thomas collins",
  "hospital": "grand oak community hospital"
}
```

- Open a command line terminal and execute the following command from the location where `request.json` file you created is saved:

```
curl -v -X POST --data @request.json http://localhost:8280/healthcare/categories/surgery/reserve --header "Content-Type:application/json"
```

This is derived from the [URI-Template defined](#) when creating the API resource.

`http://<host>:<port>/categories/{category}/reserve`

You will see the response as follows:

```
{"appointmentNumber":2,
"doctor":
  {"name":"thomas collins",
   "hospital":"grand oak community hospital",
   "category":"surgery","availability":"9.00 a.m - 11.00 a.m",
   "fee":7000.0},
"patient":
  {"name":"John Doe",
   "dob":"1990-03-19",
   "ssn":"234-23-525",
   "address":"California",
   "phone":"8770586755",
   "email":"johndoe@gmail.com"},
"fee":7000.0,
"confirmed":false}
```

You have now explored how WSO2 ESB can receive a message in one format and transform it into the format expected by the backend service using the Data Mapper mediator.

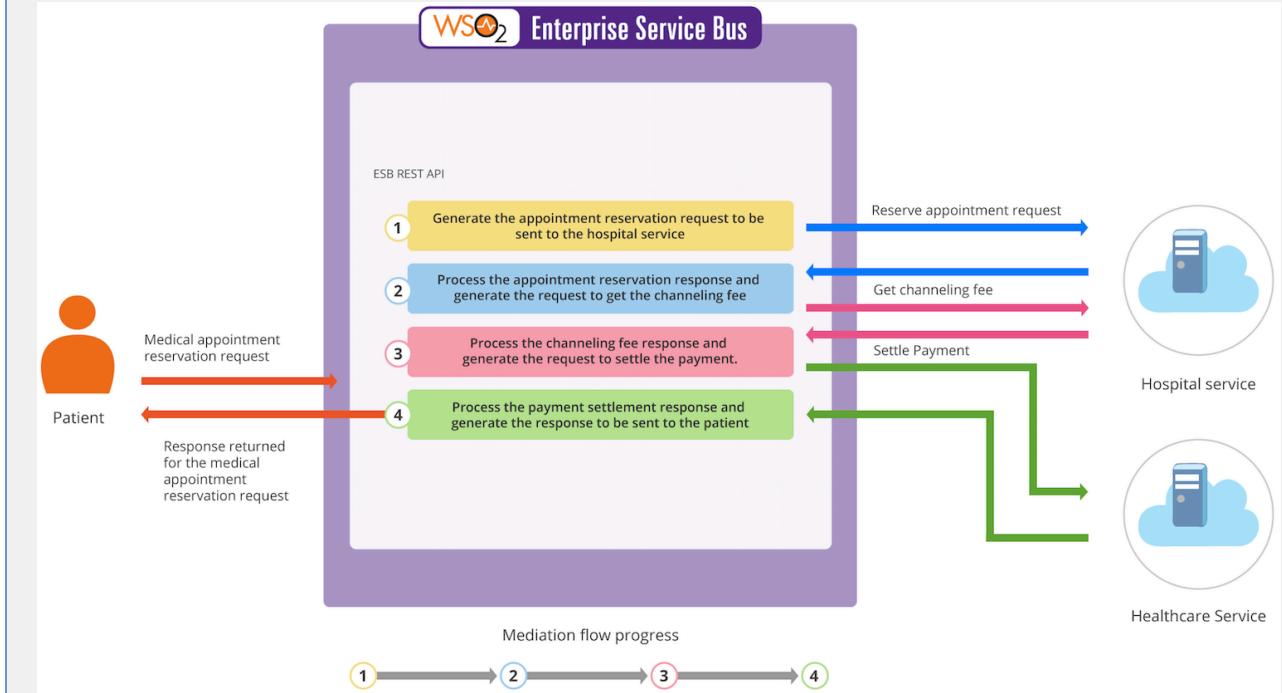
Exposing Several Services as a Single Service

In WSO2 ESB this is commonly referred to as Service Chaining where several services are integrated based on some business logic and exposed as a single, aggregated service.

In this tutorial, you send a message through the ESB to the back-end service using the Call mediator, instead of the Send mediator. Using the Call mediator, you can build a service chaining scenario as it allows you to specify all service invocations one after the other within a single sequence.

You then use the [PayloadFactory mediator](#) to take the response from one back-end service and change it to the format that is accepted by the other back-end service.

The diagram below depicts the sample scenario:



See the following topics for a description of the **concepts** that you need to know when creating ESB artifacts:

- [REST API](#)
- [Endpoints](#)
- [Sequences](#)
- [API Resource](#)
- [Service Chaining](#)
- [Composite Application Project \(C-App\)](#)

Before you begin,

1. Install Oracle Java SE Development Kit (JDK) version 1.8.* and set the JAVA_HOME environment variable.
2. Go to <http://wso2.com/products/enterprise-service-bus/>, click **DOWNLOAD** to download the ESB runtime ZIP file, and then extract the ZIP file.
The path to this folder will be referred to as <ESB_HOME> throughout the tutorials.
3. Go to <http://wso2.com/products/enterprise-service-bus/>, click **Tooling** to select and download the relevant ESB tooling ZIP file, and then extract the ZIP file.

The path to this folder will be referred to as <TOOLING_HOME> throughout the tutorials.

For more detailed installation instructions, see the [Installing WSO2 ESB Tooling](#).

4. Open the ESB Tooling environment and click **File -> Import**. Then, select **Existing WSO2 Projects into workspace** under the WSO2 category, click **Next** and upload the [pre-packaged C-App project](#). This C-App contains the configurations of the [previous tutorial](#) so that you do not have to repeat those steps.

Let's get started!

- Connecting to the back-end services
- Creating the deployable artifacts
- Deploying the Artifacts to WSO2 ESB
- Sending requests to WSO2 ESB

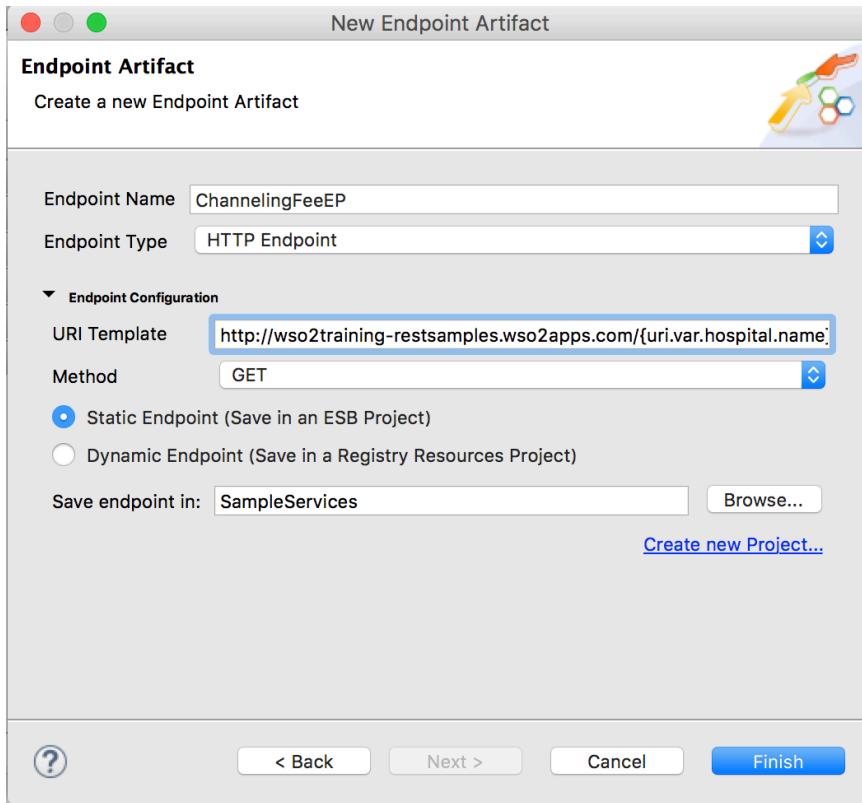
Connecting to the back-end services

Let's create HTTP endpoints to the back-end services that you need to connect to check channeling fee and settle payment.

1. Right click **SampleServices** in the Project Explorer and navigate to **New -> Endpoint**. Ensure **Create a New Endpoint** is selected and click **Next**.
2. Fill in the information as in the following table:

Field	Value
Endpoint Name	ChannelingFeeEP
Endpoint Type	HTTP Endpoint
URI Template	http://wso2training-restsamples.wso2apps.com/{uri.var.hospital}/categories/appointments/{uri.var.appointmentId}
Method	GET
Static Endpoint	(Select this option because we are going to use this endpoint in this ESB project only and will not share it with other projects.)
Save Endpoint in	SampleServices

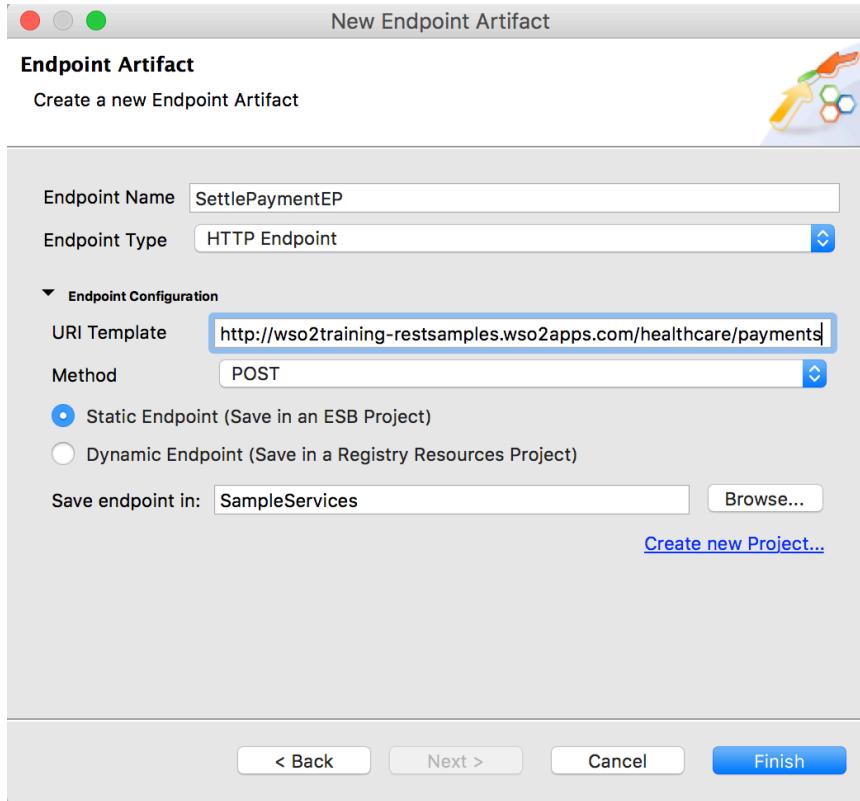
Click **Finish**.



3. Create another endpoint for Settle Payment and fill in the information as in the following table:

Field	Value
Endpoint Name	SettlePaymentEP
Endpoint Type	HTTP Endpoint
URI Template	http://wso2training-restsamples.wso2apps.com/healthcare/payments
Method	POST
Static Endpoint	(Select this option because we are going to use this endpoint in this ESB project only and will not re-use it in other projects.)
Save Endpoint in	SampleServices

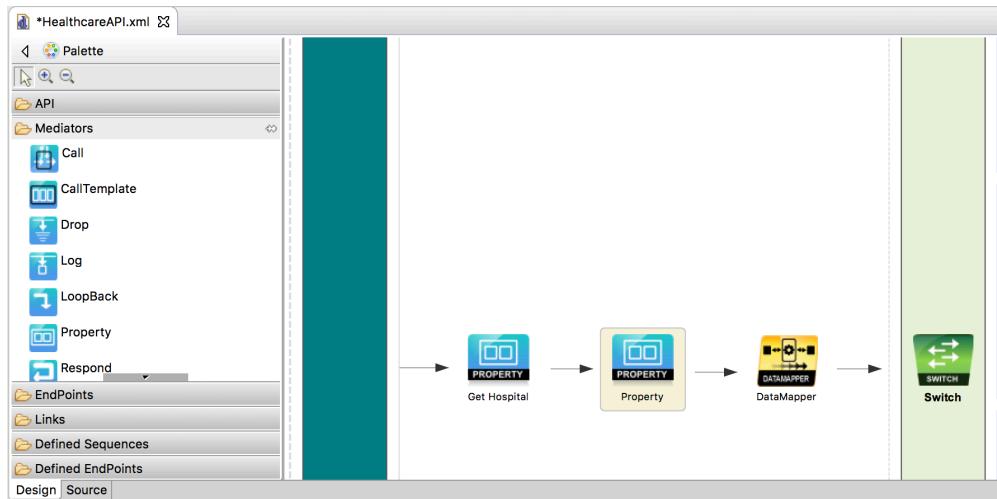
Click **Finish**.



You have now created the additional endpoints that are required for this tutorial.

Creating the deployable artifacts

1. In WSO2 ESB Tooling, add a Property mediator just after the **Get Hospital** Property mediator in the In sequence of the API resource to retrieve and store the card number that is sent in the request payload.



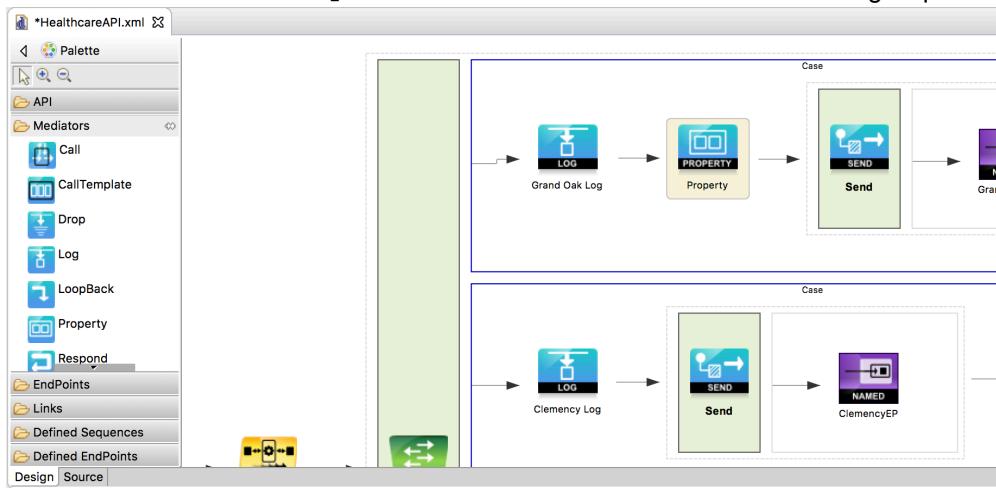
With the Property mediator selected, access the Properties tab and fill in the information as in the following table:

Field	Value
Property Name	Select New Property
New Property Name	card_number

URI Template	Select set
Value Type	Select Expression
Value Expression	json-eval(\$.cardNo)
Description	Get Card Number

For detailed instructions on adding a Propertymediator, see [Mediating requests to the back-end service](#).

2. Go to the first case box of the Switch mediator. Add a Property mediator just after the Log mediator to store the value for `uri.var.hospital` variable that will be used when sending requests to ChannelingFeeEP.

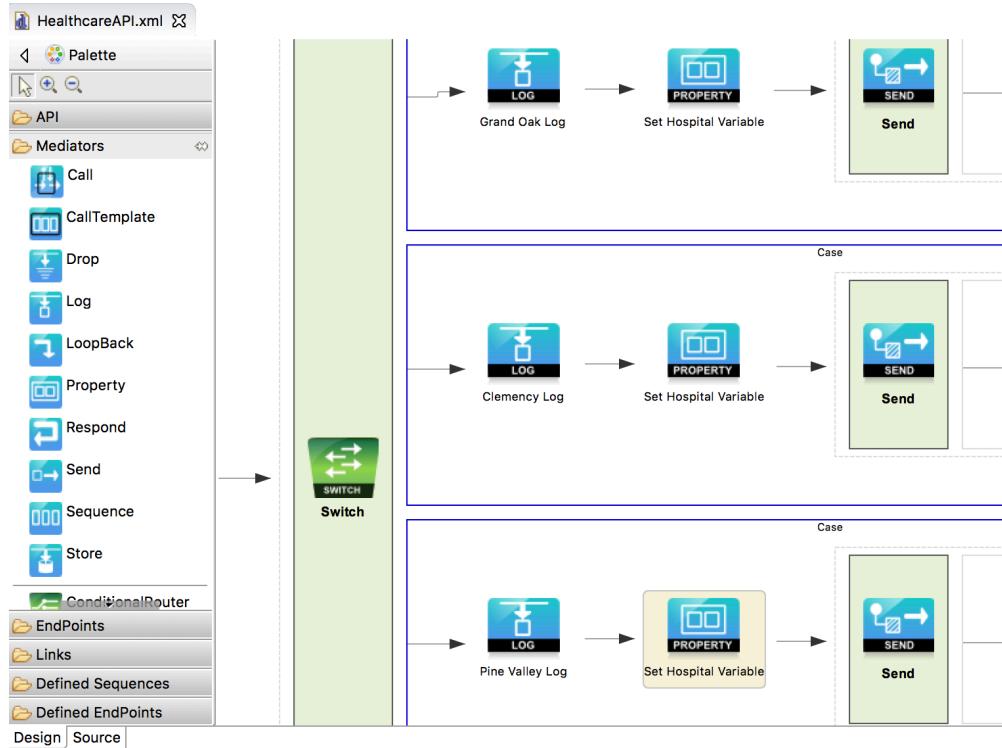


With the Property mediator selected, access the Properties tab and fill in the information as in the following table:

Field	Value
Property Name	Select New Property
New Property Name	uri.var.hospital
URI Template	Select set
Value Type	Select LITERAL
Property Data Type	Select STRING
Value	grandoaks
Description	Set Hospital Variable

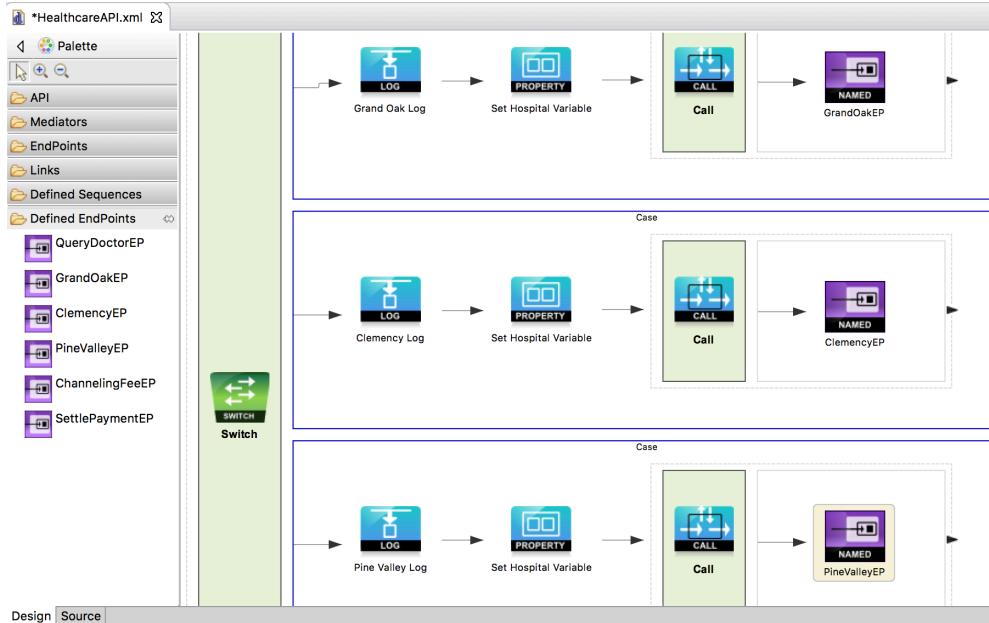
3. Similarly, add property mediators in the other two case boxes in the Switch mediator. Change only the **Value** field as follows:

- Case 2 : clemency
- Case 3: pinevalley



4. Delete the Send mediator by right clicking on the mediator and selecting **Delete from Model**. Replace this with a Call mediator from the Mediators palette and add GrandOakEP from Defined Endpoints palette to the empty box adjoining the Call mediator.

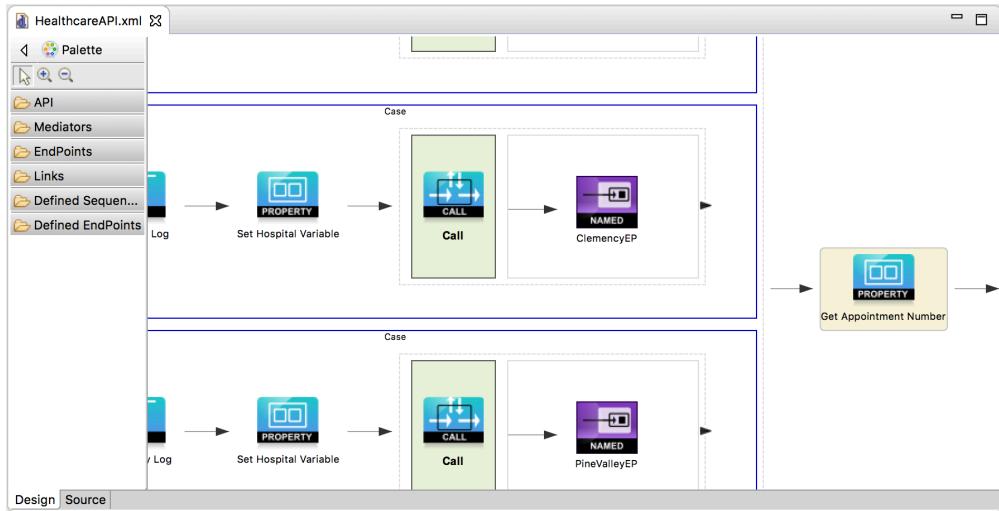
Replace the Send mediators in the following two case boxes as well and add ClemencyEP and PineValleyEP to the respective boxes adjoining the Call mediators.



Replacing with a Call mediator allows us to define other service invocations following this mediator.

Let's use Property mediators to retrieve and store the values that you get from the response you receive from GrandOakEP, ClemencyEP or PineValleyEP.

5. Next to the Call mediator box, add a Property mediator to retrieve and store the value sent as appointment Number .



With the Property mediator selected, access the Properties tab and fill in the information as in the following table:

Field	Value
Property Name	Select New Property
New Property Name	uri.var.appointment_id (This value is used when invoking ChannelingFeeEP)
URI Template	Select set
Value Type	Select EXPRESSION
Value Expression	json-eval(\$.appointmentNumber)
Description	Get Appointment Number

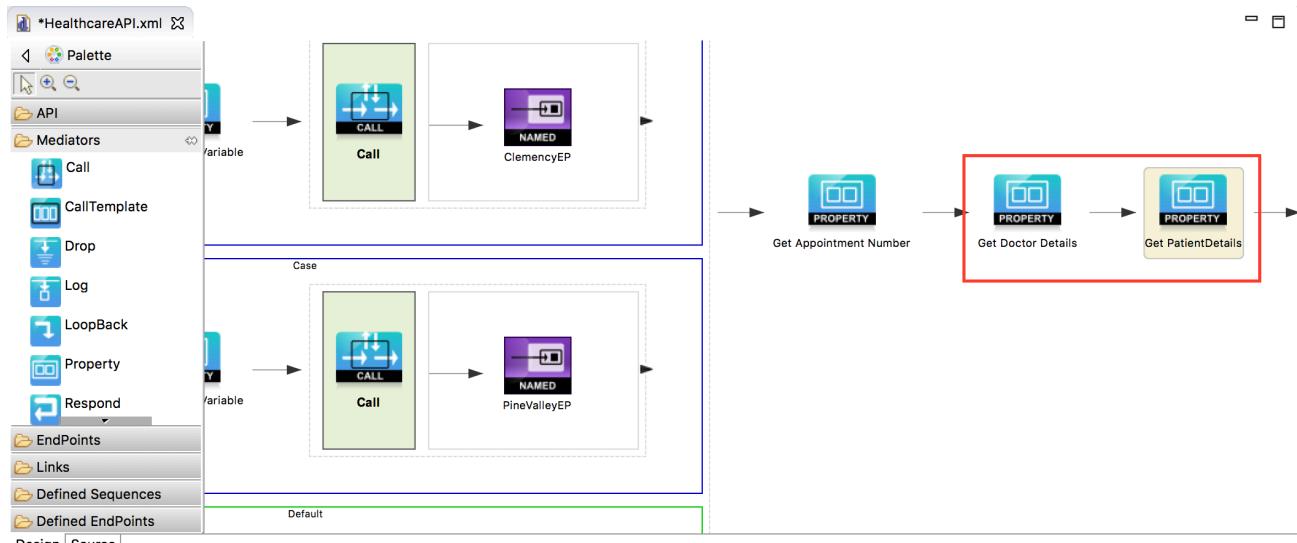
You derive the **Value Expression** in the above table from the following response that is received from GrandOakEP, ClemencyEP or PineValleyEP:

```
{
  "appointmentNumber":1,    "doctor": {
    "name":"thomas collins",
    "hospital":"grand oak community hospital",
    "category":"surgery","availability":"9.00 a.m - 11.00 a.m",
    "fee":7000.0},
  "patient": {
    "name":"John Doe",
    "dob":"1990-03-19",
    "ssn":"234-23-525",
    "address":"California",
    "phone":"8770586755",
    "email":"johndoe@gmail.com"} ,
  "fee":7000.0,
  "confirmed":false}
```

- Similarly, add two more Property mediators as follows. They retrieve and store the doctor details and patient details respectively, from the response that is received from GrandOakEP, ClemencyEP or

PineValleyEP.

Field	Value
Property Name	Select New Property
New Property Name	doctor_details
URI Template	Select set
Value Type	Select EXPRESSION
Value Expression	json-eval(\$.doctor)
Description	Get Doctor Details
Property Name	Select New Property
New Property Name	patient_details
URI Template	Select set
Value Type	Select EXPRESSION
Value Expression	json-eval(\$.patient)
Description	Get Patient Details



7. Add a Call mediator and add ChannelingFeeEP from Defined Endpoints palette to the empty box adjoining the Call mediator.
8. Add a Property mediator adjoining the Call mediator box to retrieve and store the value sent as `actualFee`. Access the Properties tab of the mediator and fill in the information as in the following table:

Field	Value
Property Name	Select New Property
New Property Name	actual_fee (This value is used when invoking SettlePaymentEP)

URI Template	Select set
Value Type	Select EXPRESSION
Value Expression	json-eval(\$.actualFee)
Description	Get Actual Fee

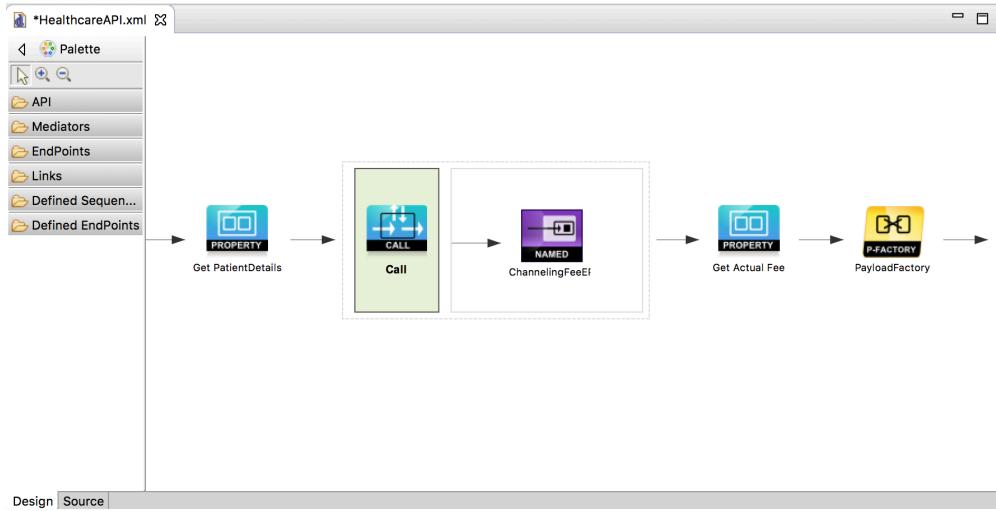
You derive the **Value Expression** in the above table from the following response that is received from ChannelingFeeEP:

```
{"patientName": " John Doe ",  
 "doctorName": "thomas collins",  
 "actualFee": "7000.0"}
```

Let's use the [PayloadFactory mediator](#) to construct the following message payload for the request sent to SettlePaymentEP.

```
{"appointmentNumber":2,  
 "Doctor":{  
   "name": "thomas collins",  
   "hospital": "grand oak community hospital",  
   "category": "surgery",  
   "availability": "9.00 a.m - 11.00 a.m",  
   "Fee":7000.0  
 },  
 "Patient":{  
   "name": "John Doe",  
   "Dob": "1990-03-19",  
   "ssn": "234-23-525",  
   "address": "California",  
   "phone": "8770586755",  
   "email": "johndoe@gmail.com"  
 },  
 "Fee":7000.0,  
 "Confirmed":false,  
   "card_number": "1234567890"  
 }
```

9. Next to the Property mediator, add a PayloadFactory mediator from the mediators palette to construct the above message payload.



With the Payloadfactory mediator selected, access the properties tab of the mediator and fill in the information as in the following table:

Field	Value	Description
Payload Format	Select Inline	-
Payload	{ "appointmentNumber":\$1, "doctor":\$2, "patient":\$3, "fee":\$4, "confirmed":"false", "card_number":"\$5" }	The message payload to send with the request to SettlePaymentEP. In this payload, \$1, \$2, \$3, \$4 and \$5 indicate variables.
Media Type	Select json	-

We will look at adding the value for the field **Args** in the following steps.

To avoid getting an error message, first select the **Media Type** before providing the **Payload**.

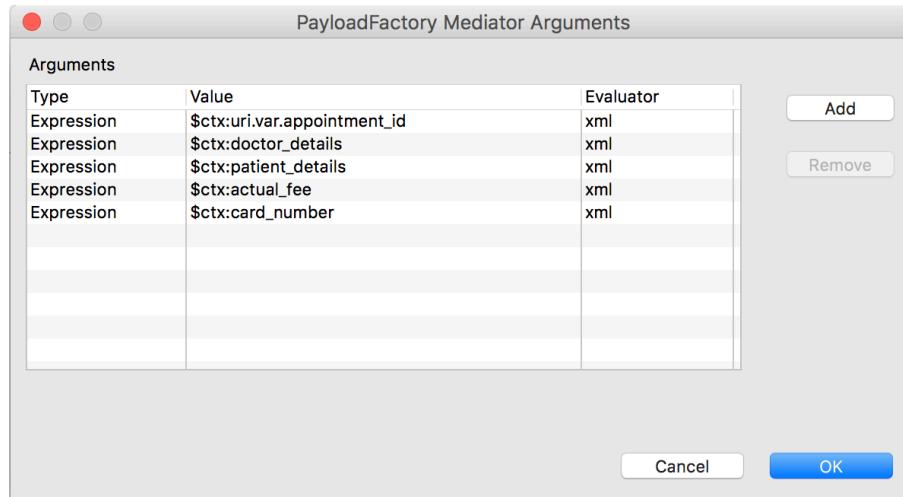
10. To add the **Args** field for the PayloadFactory mediator, click the Browse (...) icon in the Args field. Click on the **Add** button and enter the following information as in the table below. It provides the argument that defines the actual value of the first variable used in the format definition in the previous step.

Field	Value	Description
Type	Select Expression	-
Value	\$ctx:uri.var.appointment_id	The value for the first variable (\$1) in the message payload format.
Evaluator	Select xml	Indicates that the expression provided is in XML.

The `$ctx` method is similar to using the `get-property` method. This method checks in the message context. For more details on using this method, refer the [documentation](#).

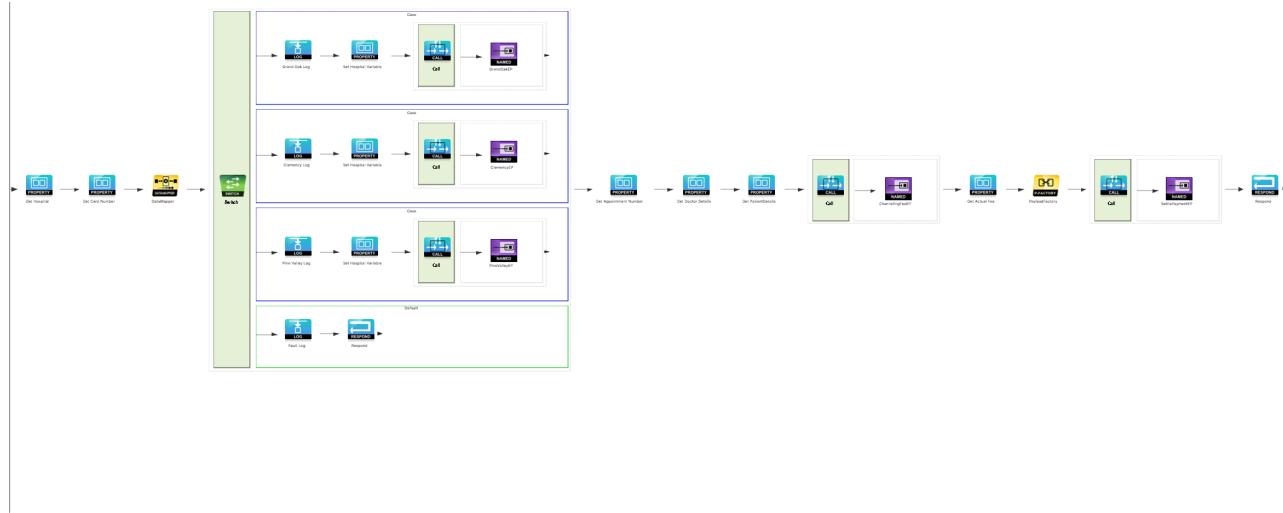
11. Similarly, click **Add** and add more arguments to define the other variables that are used in the message payload format definition. Use the following as the **Value** for each of them:

```
$ctx:doctor_details  
$ctx:patient_details  
$ctx:actual_fee  
$ctx:card_number
```



12. Add a Call mediator and add SettlePaymentEP from Defined Endpoints palette to the empty box adjoining the Call mediator.
 13. Add a Respond mediator to send the response to the client.

You should now have a completed configuration that looks like this:



14. Save the updated REST API configuration.

Deploying the Artifacts to WSO2 ESB

Since you created new endpoints, these will need to be packaged into our existing CApp.

1. Package the C-App names SampleServicesCompositeApplication project with the artifacts created.

Ensure the following artifact check boxes are selected in the **Composite Application Project POM Editor**.

- SampleServices
 - HealthcareAPI
 - ClemencyEP

- GrandOakEP
- PineValleyEP
- ChannelingFeeEP
- SettlePaymentEP
- SampleServicesRegistry

2. On the Servers tab, expand the WSO2 Carbon server, right-click **SampleServicesCompositeApplication**, and choose **Redeploy**. The Console window will indicate that the CApp has been deployed successfully.

If you do not have a server added in Eclipse, refer [this tutorial](#).

You can also deploy the artifacts to the ESB server using a [Composite Application Archive \(CAR\)](#) file

Sending requests to WSO2 ESB

1. Create a JSON file names `request.json` with the following request payload.

```
{
  "name": "John Doe",
  "dob": "1940-03-19",
  "ssn": "234-23-525",
  "address": "California",
  "phone": "8770586755",
  "email": "johndoe@gmail.com",
  "doctor": "thomas collins",
  "hospital": "grand oak community hospital",
  "cardNo": "7844481124110331"
}
```

2. Open a command line terminal and execute the following command from the location where `request.json` file you created is saved:

```
curl -v -X POST --data @request.json http://localhost:8280/healthcare/categories/surgery/reserve --header "Content-Type:application/json"
```

This is derived from the [URI-Template defined](#) when creating the API resource.

`http://<host>:<port>/categories/{category}/reserve`

You will see the response as follows:

```
{"patient": "John Doe",
 "actualFee": 7000.0,
 "discount": 20,
 "discounted": 5600.0,
 "paymentID": "e2781025-5332-4a78-950b-3be83c99fa76",
 "status": "Settled"}
```

You have now explored how WSO2 ESB can do service chaining using the Call mediator and transform message payloads from one format to another using the PayloadFactory mediator.

Storing and Forwarding Messages

Store and forward messaging is used for serving traffic to back-end services that can accept request messages only at a given rate. This is also used for guaranteed delivery to ensure that request received never gets lost since they are stored in the message store and also available for future reference.

In this tutorial, instead of sending the request directly to the back-end service, you store the request message into an [In Memory Message Store](#). You then use a [Message Processor](#) to retrieve the message from the store and then deliver the message to the back-end service.

See the following topics for a description of the **concepts** that you need to know when creating ESB artifacts:

- REST API
- Endpoints
- Sequences
- API Resource
- [Composite Application Project \(C-App\)](#)
- Store and Forward

Before you begin,

1. Install Oracle Java SE Development Kit (JDK) version 1.8.* and set the JAVA_HOME environment variable.
 2. Go to <http://wso2.com/products/enterprise-service-bus/>, click **DOWNLOAD** to download the ESB runtime ZIP file, and then extract the ZIP file.
The path to this folder will be referred to as <ESB_HOME> throughout the tutorials.
 3. Go to <http://wso2.com/products/enterprise-service-bus/>, click **Tooling** to select and download the relevant ESB tooling ZIP file, and then extract the ZIP file.
The path to this folder will be referred to as <TOOLING_HOME> throughout the tutorials.
- For more detailed installation instructions, see the [Installing WSO2 ESB Tooling](#).
4. Open the ESB Tooling environment and click **File -> Import**. Then, select **Existing WSO2 Projects into workspace** under the WSO2 category, click **Next** and upload the extracted [pre-packaged C-App](#) project. This C-App contains the configurations of the [previous tutorial](#) so that you do not have to repeat those steps.

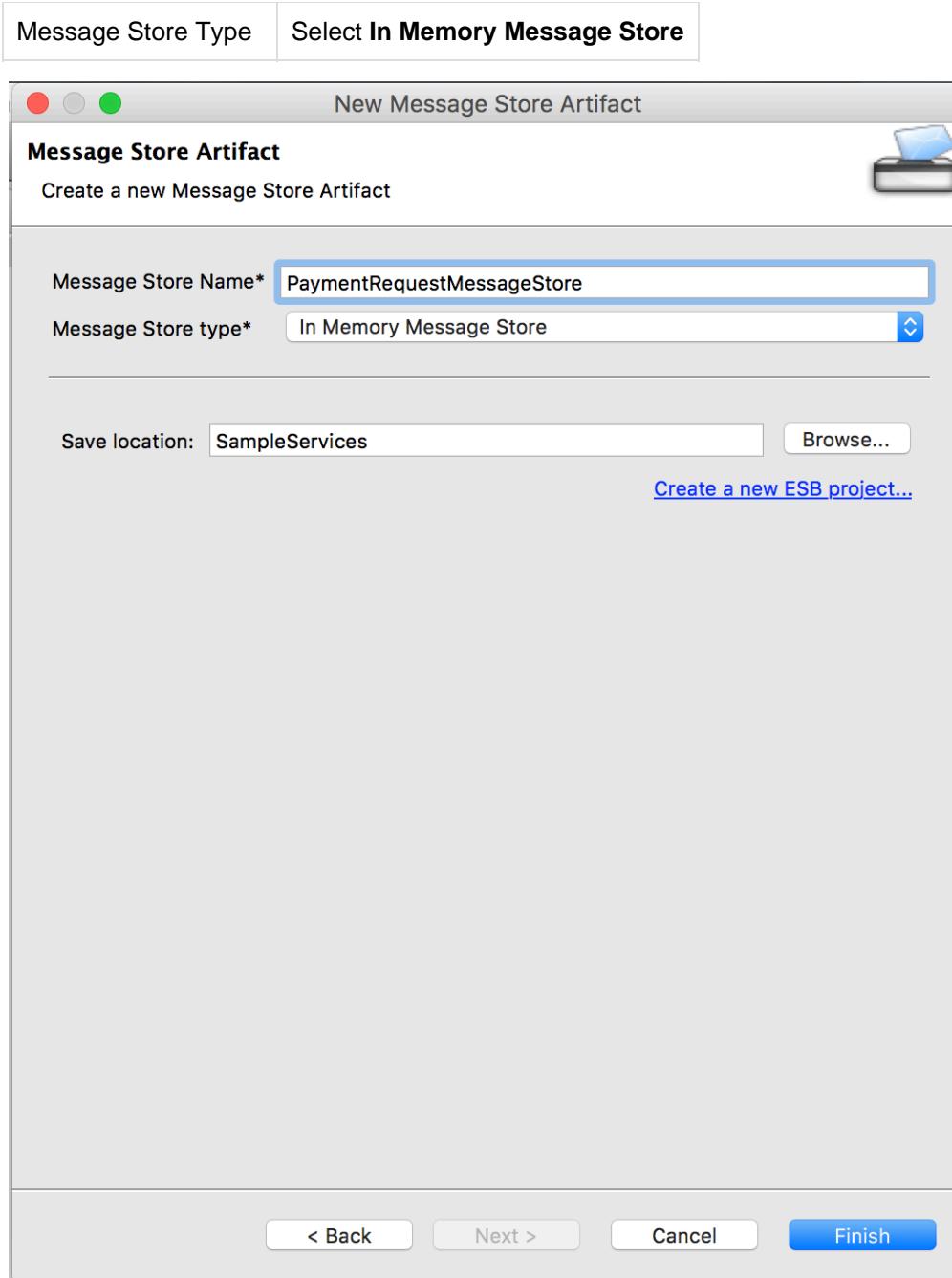
Let's get started!

- [Creating the Message Store](#)
- [Creating the Deployable Artifacts](#)
- [Creating the Response Sequence](#)
- [Creating the Message Processor](#)
- [Deploying the Artifacts to WSO2 ESB](#)
- [Sending requests to WSO2 ESB](#)

Creating the Message Store

1. Right click on **SampleServices** in the Project Explorer and navigate to **New->Message Store**.
2. Select **Create a new message-store artifact** and fill in the information in the following table and click **Finish**.

Field	Value
Message Store Name	PaymentRequestMessageStore

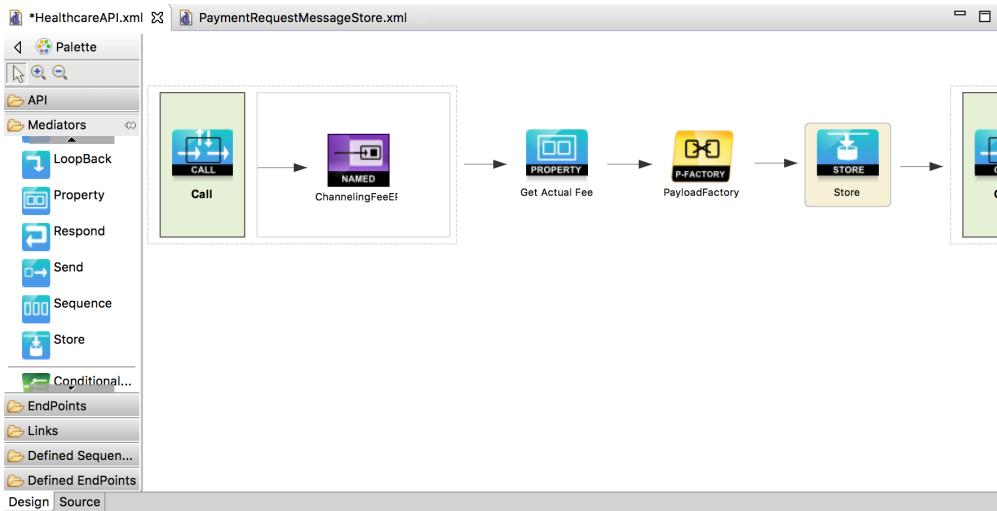


Let's look at how we update the REST API with the Store mediator.

Creating the Deployable Artifacts

Let's now update the REST API so that the messages sent to SettlePaymentEP is forwarded to the message store we created above.

1. Drag and add a Store Mediator from the mediators palette just after the PayloadFactory mediator.



2. With the Store mediator selected, access the Properties tab and fill in the information as in the following table:

Field	Value	Description
Available Message Store	Select PaymentRequestMessageStore	After selecting this, when you click Message Store field the value will be populated.
Description	Payment Store	

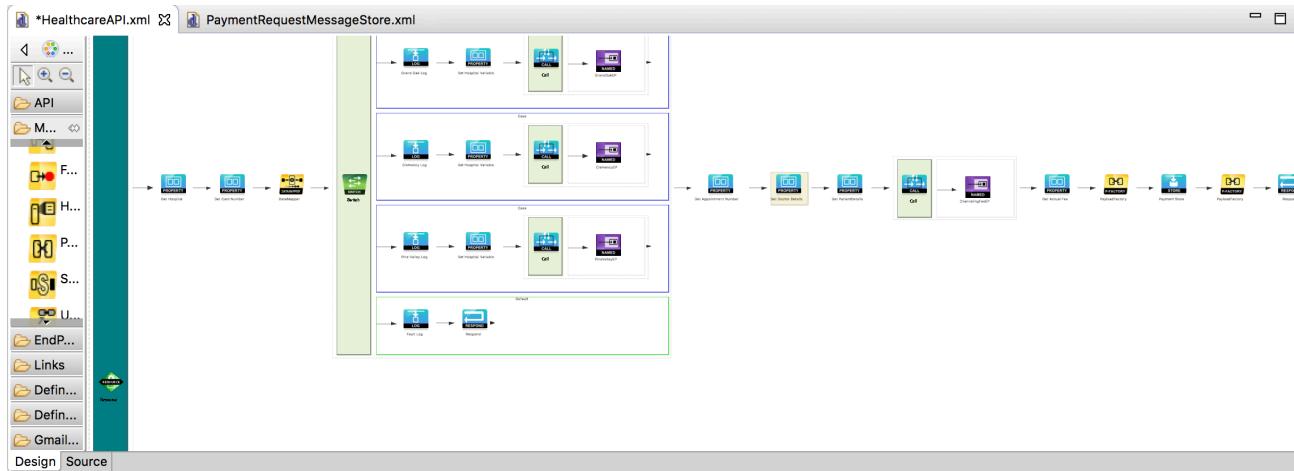
Let's use a PayloadFactory mediator to send a customized response message to the client.

3. Delete the Call mediator by right clicking on the mediator and selecting **Delete from Model**. Replace this with a PayloadFactory mediator from the Mediators palette to configure the response to be sent to the client. With the PayloadFactory mediator selected, access the Properties tab and fill in the information in the following table to define a customized message to be returned to the client.

Field	Value
Payload Format	Select Inline
Payload	{"message": " Payment request successfully submitted. Payment confirmation will be sent via email . "}
Media Type	Select json

To avoid getting an error message, first select **Media Type** before selecting **Payload**.

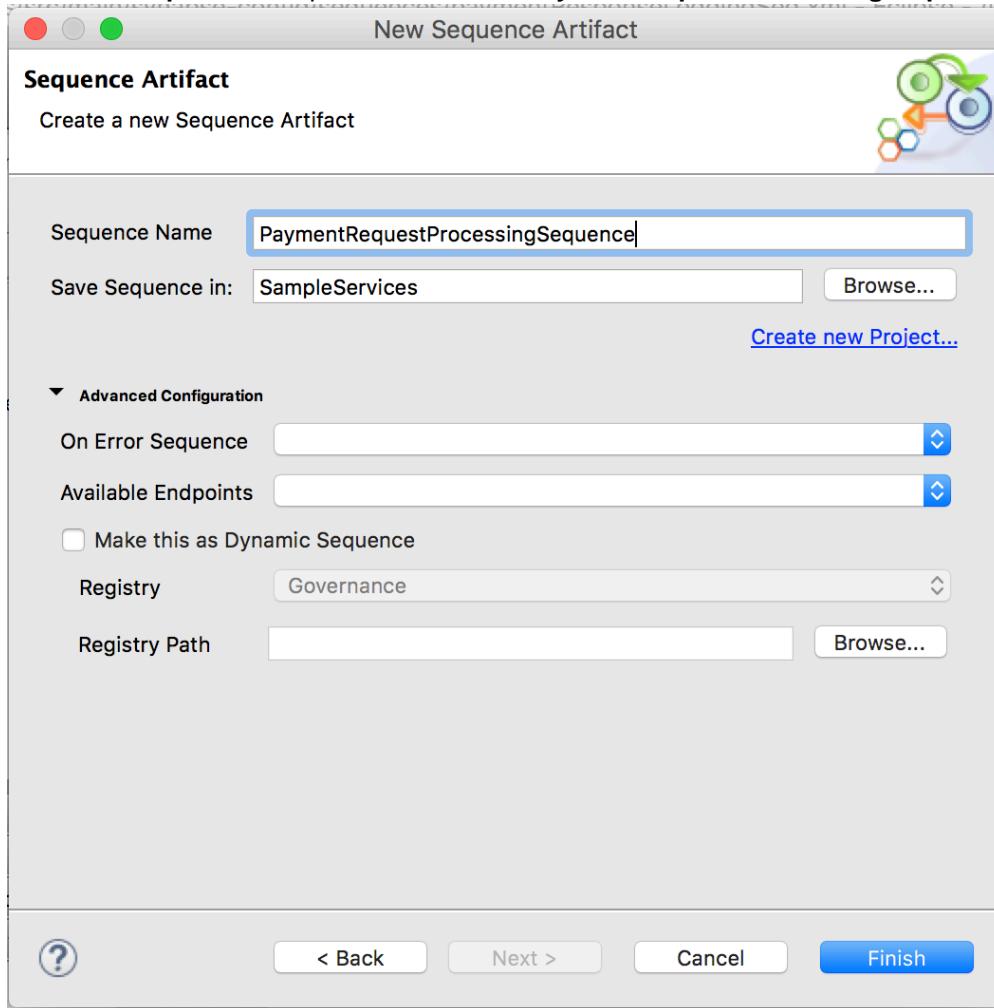
You should now have a completed configuration that looks like this:



Creating the Response Sequence

Let's create a Sequence that uses the message in the message store to send the request to SettlePaymentEP.

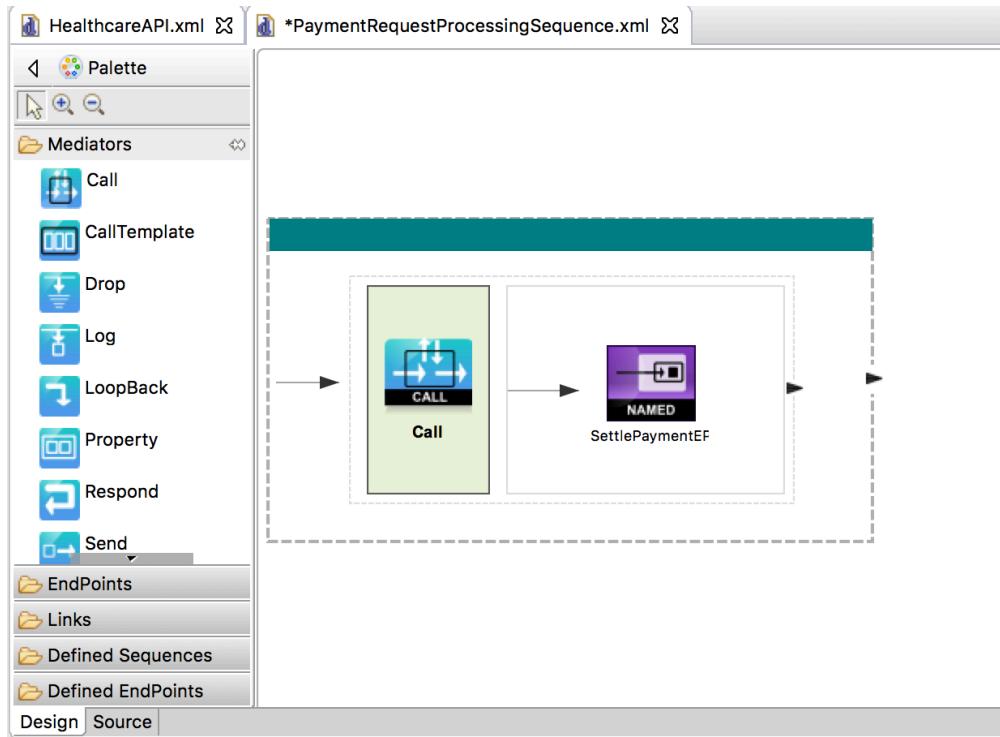
1. Right click the **SampleServices** project in the Project Explorer and navigate to **New -> Sequence**. Select **Create New Sequence** and provide the name **PaymentRequestProcessingSequence**.



Click **Finish**.

2. Drag and drop a Call mediator from the Mediators palette and add SettlePaymentEP from Defined Endpoints palette to the empty box adjoining the Call mediator. This sends the request message from the store to

SettlePaymentEP.

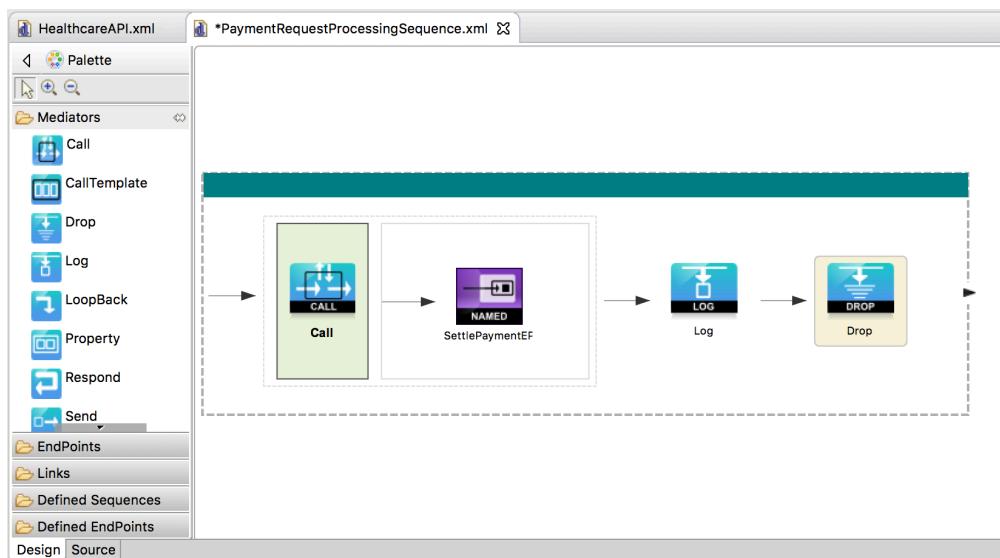


- Drag and drop a Log mediator from the Mediators palette to log the response from SettlePaymentEP. Access the Properties tab and fill in the following information:

Field	Value
Log Category	Select INFO
Log Level	Select FULL

- Add a Drop mediator from the Mediators palette.

You should now have a completed sequence configuration that looks like this:



Creating the Message Processor

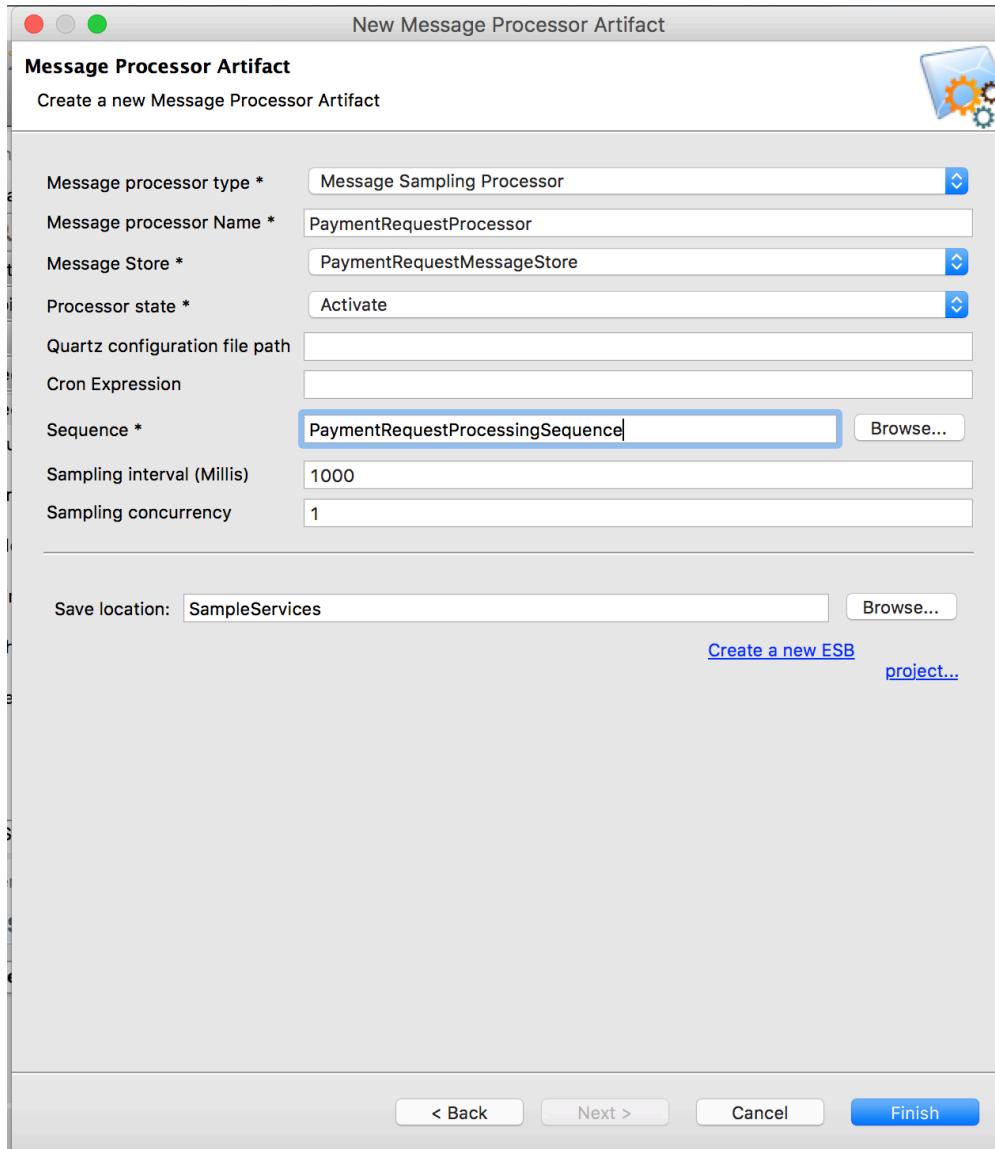
Let's create a [Message Sampling Processor](#) to dispatch the request message from the [message store](#) to the [PaymentRequestProcessingSequence](#).

You can also use the [Scheduled Message Forwarding Processor](#) here and define the endpoint within the processor.

The Message Sampling Processor is used because you need to perform mediation on the request message in the next tutorial.

Right click the **SampleServices** project in the Project Explorer and navigate to **New -> Message Processor**. Select **create a new message-processor artifact** and fill in the details as in the following table:

Field	Value	Description
Message Processor Type	Select Message Sampling Processor	This processor takes the message from the store and puts it into a sequence.
Message Processor Name	PaymentRequestProcessor	The name of the scheduled message forwarding processor.
Message Store	Select PaymentRequestMessageStore	The message store from which the scheduled message forwarding processor consumes messages.
Processor State	Activate	Whether the processor needs to be activated or deactivated.
Sequence	Select PaymentRequestProcessingSequence	The name of the sequence the message from the store needs to be sent to.



Click **Finish**.

We have now finished creating all the required artifacts.

Deploying the Artifacts to WSO2 ESB

Since you created a message store, sequence and a message processor, these will need to be packaged into our existing CApp.

1. Package the C-App names SampleServicesCompositeApplication project with the artifacts created.

Ensure the following artifact check boxes are selected in the **Composite Application Project POM Editor**.

- SampleServices
 - HealthcareAPI
 - ClemencyEP
 - GrandOakEP
 - PineValleyEP
 - ChannelingFeeEP

- SettlePaymentEP
- PaymentRequestMessageStore
- PaymentRequestProcessingSequence
- PaymentRequestProcessor
- SampleServicesRegistry

2. On the Servers tab, expand the WSO2 Carbon server, right-click **SampleServicesCompositeApplication**, and choose **Redeploy**. The Console window will indicate that the CApp has been deployed successfully.

If you do not have a server added in Eclipse, refer [this tutorial](#).

You can also deploy the artifacts to the ESB server using a [Composite Application Archive \(CAR\)](#) file

Sending requests to WSO2 ESB

1. Create a JSON file names `request.json` with the following request payload.

```
{
  "name": "John Doe",
  "dob": "1940-03-19",
  "ssn": "234-23-525",
  "address": "California",
  "phone": "8770586755",
  "email": "johndoe@gmail.com",
  "doctor": "thomas collins",
  "hospital": "grand oak community hospital",
  "cardNo": "7844481124110331"
}
```

2. Open a command line terminal and execute the following command from the location where `request.json` file you created is saved:

```
curl -v -X POST --data @request.json http://localhost:8280/healthcare/categories/surgery/reserve --header "Content-Type:application/json"
```

This is from the [URI-Template defined](#) when creating the API resource QueryDoctorAPI.

`http://<host>:<port>/categories/{category}/reserve`

You will see the response as follows:

```
{"message": "Payment request successfully submitted. Payment confirmation will be sent via email."}
```

3. Now check the ESB server Console in Eclipse and you will see that the response from SettlePaymentEP is logged.

You have now explored how WSO2 ESB can be used to implement store and forward messaging using Message Stores, Message Processors and the Store mediator.

Using the Gmail Connector

In this tutorial, you use the Gmail connector to send an email containing the response received from SettlePaymentEP in the [previous tutorial](#).

See the following topics for a description of the **concepts** that you need to know when creating ESB artifacts:

- REST API
- Endpoints
- Sequences
- API Resource
- Connectors
- [Composite Application Project \(C-App\)](#)
- Store and Forward

Before you begin,

1. Install Oracle Java SE Development Kit (JDK) version 1.8.* and set the JAVA_HOME environment variable.
 2. Go to <http://wso2.com/products/enterprise-service-bus/>, click **DOWNLOAD** to download the ESB runtime ZIP file, and then extract the ZIP file.
The path to this folder will be referred to as <ESB_HOME> throughout the tutorials.
 3. Go to <http://wso2.com/products/enterprise-service-bus/>, click **Tooling** to select and download the relevant ESB tooling ZIP file, and then extract the ZIP file.
The path to this folder will be referred to as <TOOLING_HOME> throughout the tutorials.
- For more detailed installation instructions, see the [Installing WSO2 ESB Tooling](#).
4. Open the ESB Tooling environment and click **File -> Import**. Then, select **Existing WSO2 Projects into workspace** under the WSO2 category, click **Next** and upload the extracted [pre-packaged C-App](#) project. This C-App contains the configurations of the [previous tutorial](#) so that you do not have to repeat those steps.

Let's get started!

- Creating the credentials for using the using the Gmail Connector
- Importing the Gmail Connector into ESB Tooling
- Creating the Deployable Artifacts
- Deploying the Artifacts to WSO2 ESB
- Sending requests to WSO2 ESB

Creating the credentials for using the using the Gmail Connector

Creating a Client ID and Client Secret

1. As the email sender, navigate to the URL <https://console.developers.google.com/projectselector/apis/credentials> and log in to your google account.
2. If you do not already have a project, create a new project and navigate to **Create Credential -> OAuth client ID**.

At this point, if the consent screen name is not provided, you will be prompted to do so.

3. Select the Web Application option and create a client. Provide <https://developers.google.com/oauthplayground> as the redirect URL under **Authorized redirect URIs** and click on **Create**. The client ID and client secret will then be displayed.

[See Gmail API documentation](#) for details on creating the Client ID and Client Secret.

4. Click on the Library on the side menu, and select **Gmail API**. Click enable.

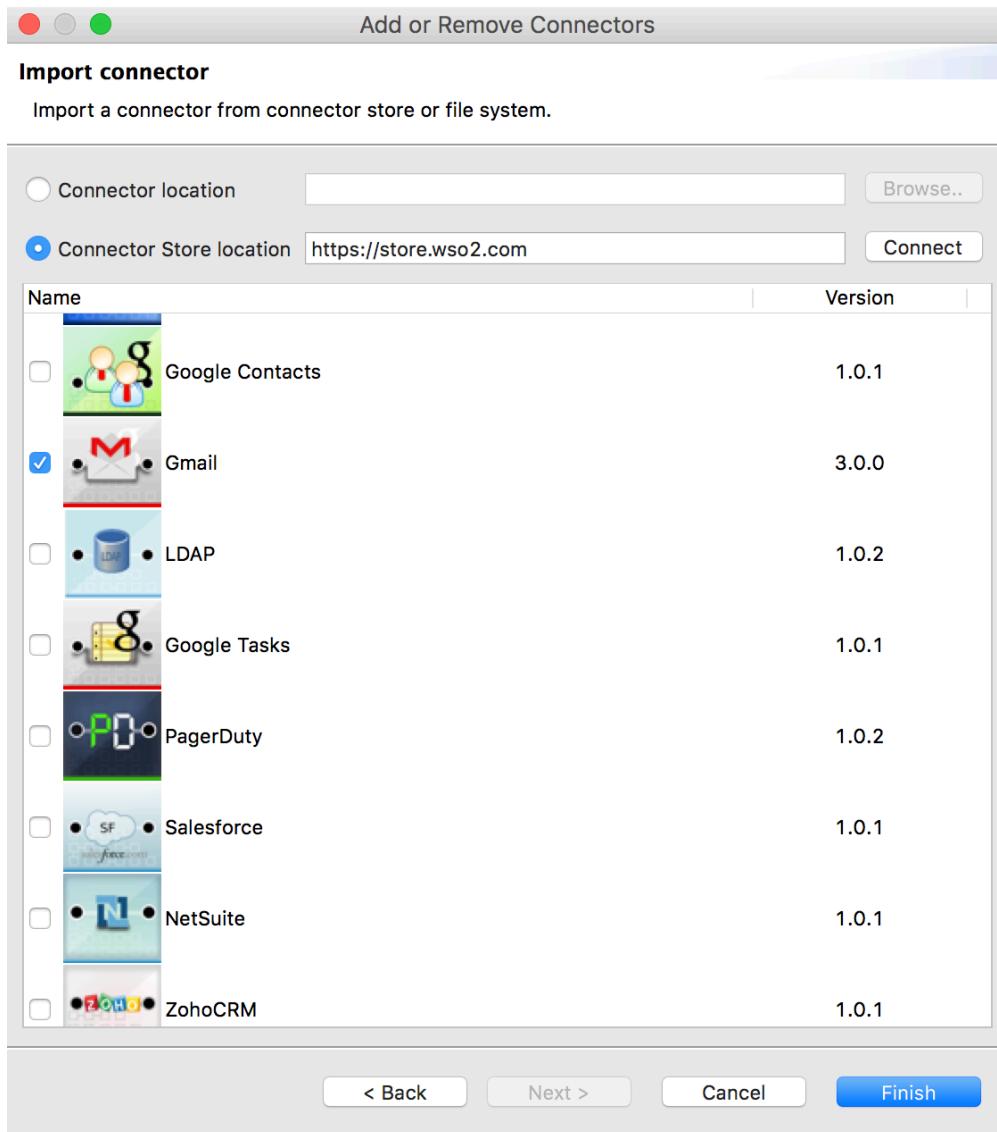
Obtaining the Access Token and Refresh Token

Follow these steps to automatically refresh the expired token when connecting to Google API:

1. Navigate to the URL <https://developers.google.com/oauthplayground> and click on the gear wheel at the top right corner of the screen and select the option **Use your own OAuth credentials**. Provide the client ID and client secret you [previously created](#) and click on **Close**.
2. Now under Step 1, select **Gmail API v1** from the list of APIs and check all the scopes listed down and click on **Authorize APIs**. You will then be prompted to allow permission, click on **Allow**.
3. In Step 2, click on **Exchange authorization code for tokens** to generate an display the access token and refresh token.

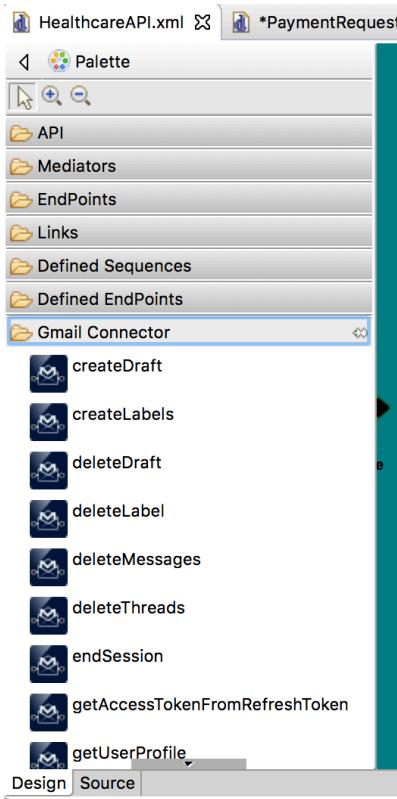
Importing the Gmail Connector into ESB Tooling

1. Right click on **Sample Services** in the Project Explorer and Select **Add or Remove Connector**. Then select **Add Connector** and click **Next**.
2. Select **Connector Store location** and click on **Connect** to connect to [WSO2 Connector store](#) and scroll down and select **Gmail** from the list of connectors.



Click **Finish**.

The connector is now downloaded into your Tooling environment and the connector operations will be available in the Gmail Connector palette.

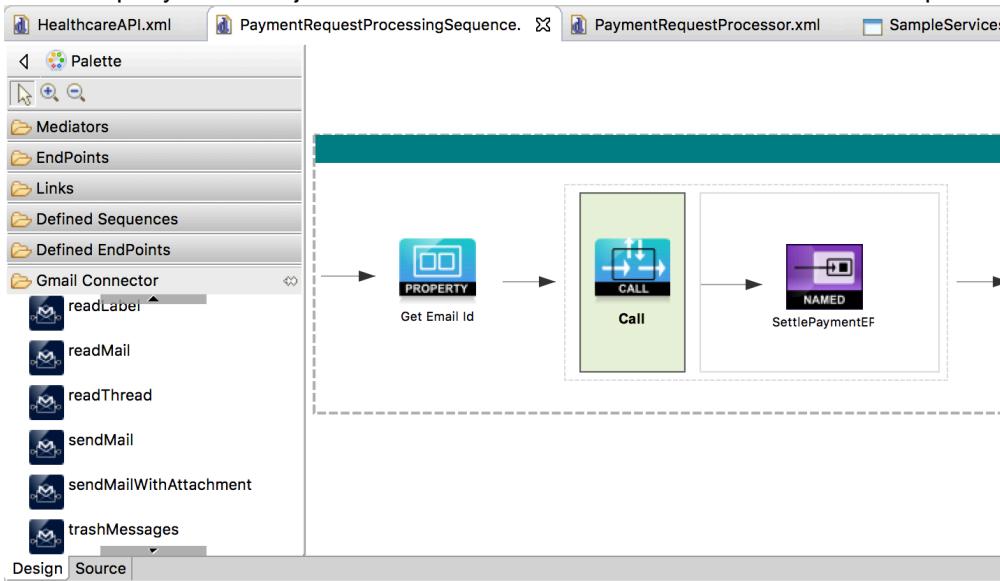


Let's use these connector operations within the ESB configuration.

Creating the Deployable Artifacts

The connector operations are used in the **PaymentRequestProcessingSequence**. Select this sequence, and do the following updates:

1. Add a Property Mediator just before the Call mediator to retrieve and store the patient's email address.

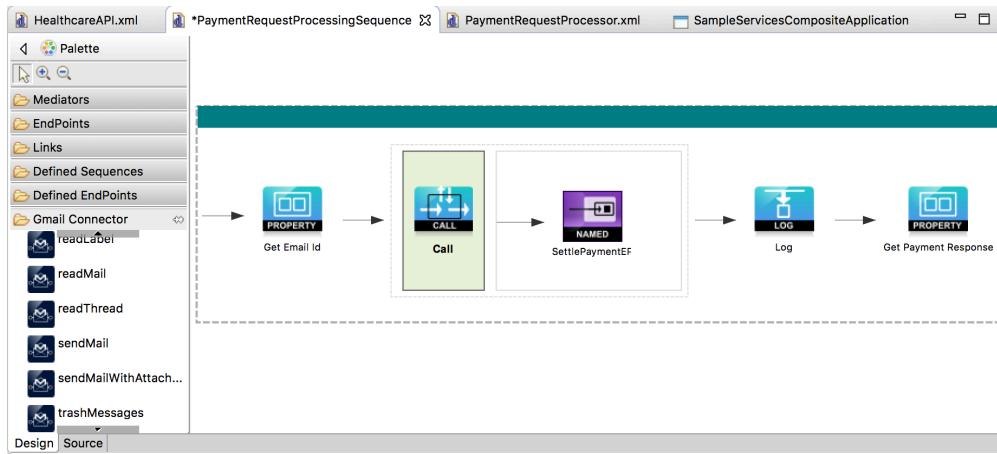


With the Property mediator selected, access the Properties tab of the mediator and fill in the information in the following table:

Field	Value

Property Name	Select New Property
New Property Name	email_id
Property Action	Select Set
Value Type	Select Expression
Value Expression	json-eval(\$.patient.email)
Description	Get Email Id

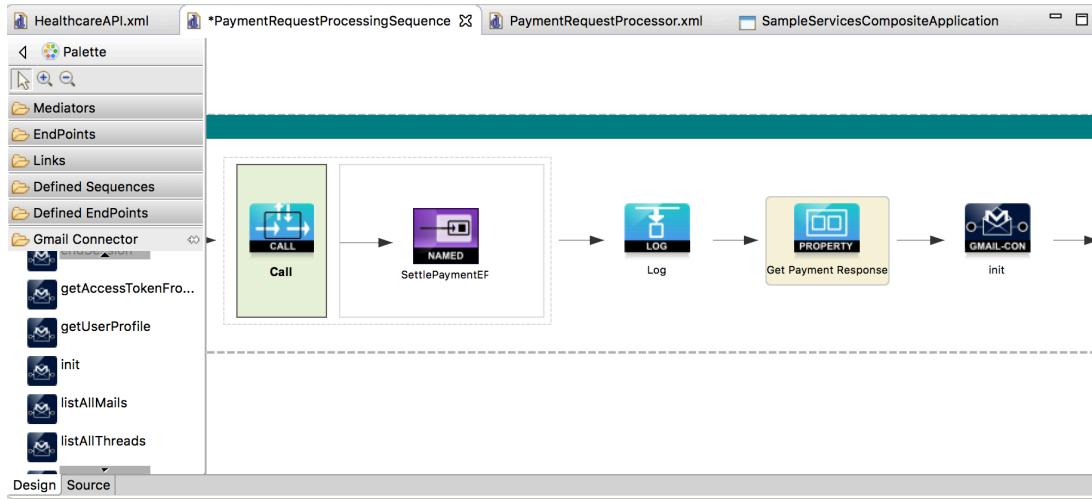
2. Add another Property mediator just after the Log mediator to retrieve and store the response sent from SettlePaymentEP. This will be used within the body of the email.



With the Property mediator selected, access the Properties tab and fill in the information in the following table:

Field	Value
Property Name	Select New Property
New Property Name	payment_response
Property Action	Select Set
Value Type	Select Expression
Value Expression	json-eval(\$.)
Description	Get Payment Response

3. Drag and drop the init method from the Gmail Connector palette adjoining the Property mediator you added in the previous step.



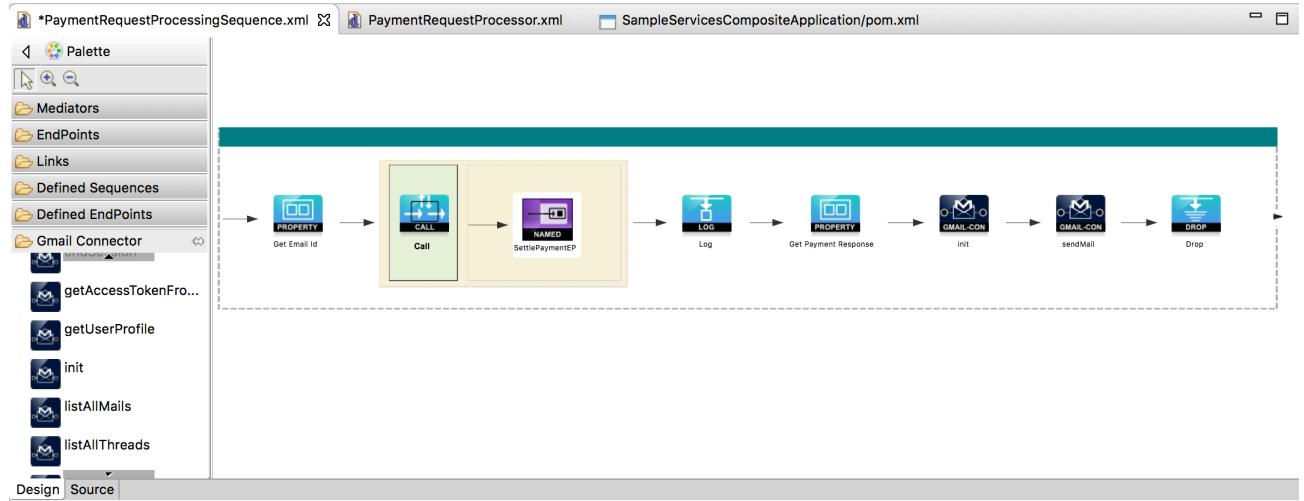
With the init method selected, access the Properties tab and fill in the information in the following table:

Field	Value
Parameter Editor Type	Select inline
userId	The sender's email address
accessToken	The access token you obtained
apiUrl	https://www.googleapis.com/gmail
clientId	The client ID you created.
clientSecret	The client secret you created.
refreshToken	The refresh token you obtained.

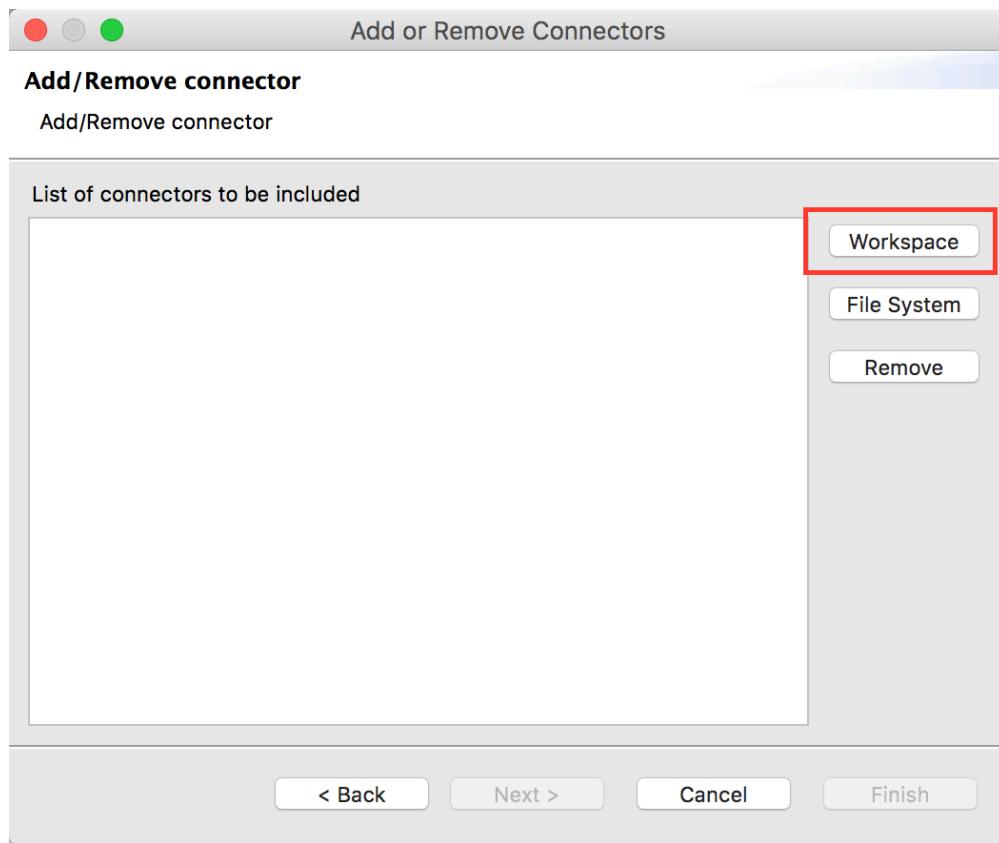
4. Add the sendMail method from the Gmail Conector palette and access the Properties tab and fill in the information in the following table:

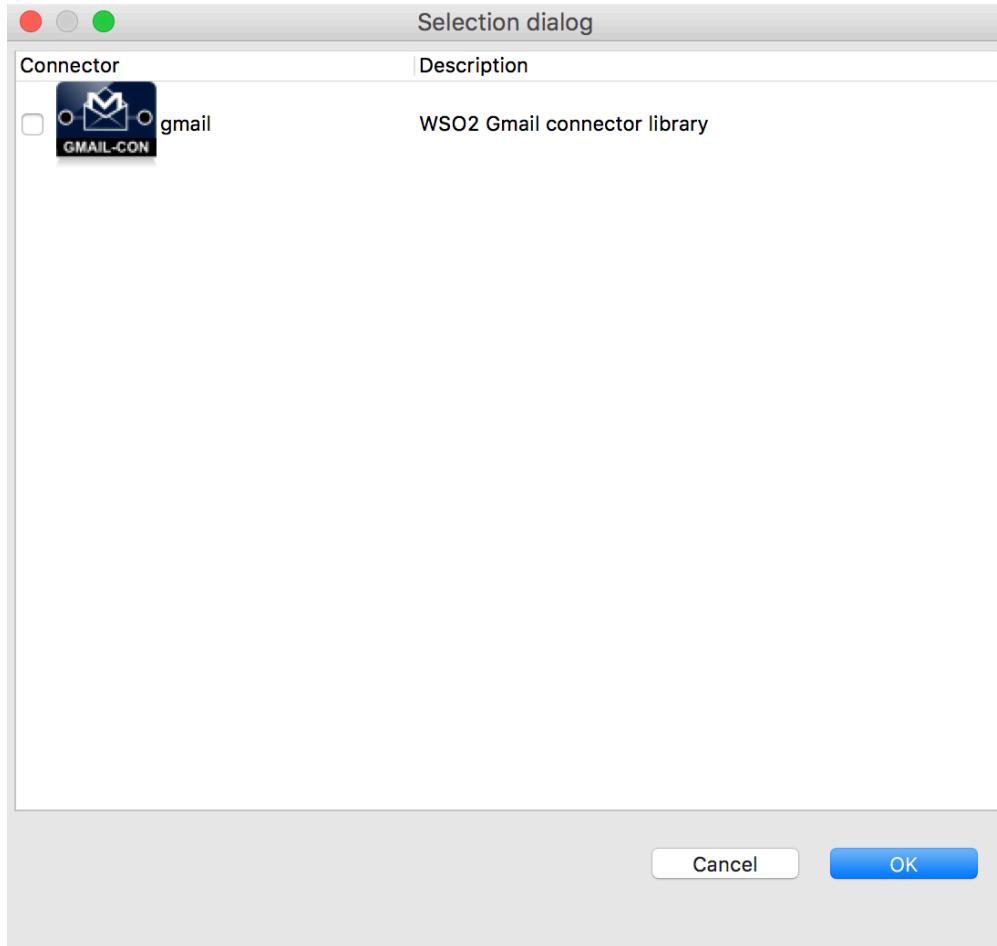
Field	Value	Description
Parameter Editor Type	Select inline	
to	<code> \${ctx:email_id}</code>	Retrieves the patient email address that was stored in the relevant Property mediator.
subject	Payment Status	The subject line in the email that is sent out.
messageBody	<code> \${ctx:payment_response}</code>	Retrieves the payment response that was stored in the relevant Property mediator.

The updated **PaymentRequestProcessingSequence** should now look like this:



5. Save the updated sequence configuration.
6. Right click on SampleServicesConnectorExporter and navigate to **New -> Add/Remove Connectors** and select **Add connector** and click on **Next**. Select **Workspace** to list down the connectors that were added.





Select the Gmail connector from the list and click **OK** and then **Finish**.

Deploying the Artifacts to WSO2 ESB

Since you added a connector to the Conector Exporter Project, this will need to be packaged into our existing CApp.

1. Package the C-App names SampleServicesCompositeApplication project with the artifacts created.

Ensure the following artifact check boxes are selected in the **Composite Application Project POM Editor**.

- SampleServices
 - HealthcareAPI
 - ClemencyEP
 - GrandOakEP
 - PineValleyEP
 - ChannelingFeeEP
 - SettlePaymentEP
- SampleServicesRegistry
- SampleServicesConnectorExporter

2. On the Servers tab, expand the WSO2 Carbon server, right-click **SampleServicesCompositeApplication**, and choose **Redeploy**. The Console window will indicate that the CApp has been deployed successfully.

If you do not have a server added in Eclipse, refer [this tutorial](#).

You can also deploy the artifacts to the ESB server using a [Composite Application Archive \(CAR\)](#) file.

Sending requests to WSO2 ESB

1. Create a JSON file names `request.json` with the following request payload. Make sure you provide a valid email address so that you can test the email being sent to the patient.

```
{
  "name": "John Doe",
  "dob": "1940-03-19",
  "ssn": "234-23-525",
  "address": "California",
  "phone": "8770586755",
  "email": "johndoe@gmail.com",
  "doctor": "thomas collins",
  "hospital": "grand oak community hospital",
  "cardNo": "7844481124110331"
}
```

2. Open a command line terminal and execute the following command from the location where `request.json` file you created is saved:

```
curl -v -X POST --data @request.json http://localhost:8280/healthcare/categories /surgery/reserve --header "Content-Type:application/json"
```

This is derived from the [URI-Template defined](#) when creating the API resource.

`http://<host>:<port>/categories/{category}/reserve`

You will see the response as follows:

```
{"message": "Payment request successfully submitted. Payment confirmation will be sent via email."}
```

An email will be sent to the provided patient email address with the following details:

```
Subject: Payment Status

Message: {"patient": "John
Doe", "actualFee": 7000.0, "discount": 20, "discounted": 5600.0, "paymentID": "21d0e220-1
714-4051-a4f5-d0d31fe42496", "status": "Settled"}
```

You have now explored how to import the Gmail connector in WSO2 ESB and then use the connector operation to send emails.

Using the Analytics Dashboard

WSO2 ESB Analytics Dashboard is used to publish information relating to the message mediation in WSO2 ESB.

The statistics are published in the analytics dashboard in an overview as well as for individual artifacts that are deployed within ESB.

In this tutorial, you use the ESB Analytics dashboard to view and analyze the Service Chaining tutorial mediation statistics.

Before you begin,

1. Install Oracle Java SE Development Kit (JDK) version 1.8.* and set the JAVA_HOME environment variable.

2. Go to <http://wso2.com/products/enterprise-service-bus/>, click **DOWNLOAD** to download the ESB runtime ZIP file, and then extract the ZIP file.

The path to this folder will be referred to as <ESB_HOME> throughout the tutorials.

3. Go to <http://wso2.com/products/enterprise-service-bus/>, click **Tooling** to select and download the relevant ESB tooling ZIP file, and then extract the ZIP file.

The path to this folder will be referred to as <TOOLING_HOME> throughout the tutorials.

For more detailed installation instructions, see the [Installing WSO2 ESB Tooling](#).

4. Go to <http://wso2.com/products/enterprise-service-bus/>, click **Analytics** to download the ESB analytics ZIP file, and then extract the ZIP file.

The path to this folder will be referred to as <ANALYTICS_HOME> throughout the quick start guide.

5. Open the ESB Tooling environment and click **File -> Import**. Then, select **Existing WSO2 Projects into workspace** under the WSO2 category, click **Next** and upload the extracted pre-packaged C-App project. This C-App contains the configurations of the service chaining tutorial. On the Servers tab, right-click the WSO2 ESB server, select **Add and Remove** and choose **SampleServicesComposite Application**, and click **Finish**.

If you do not have a server added in Eclipse, refer [this tutorial](#).

You can also deploy the artifacts to the ESB server using a [Composite Application Archive \(CAR\) file](#).

6. If you are running on Windows, download the snappy-java_1.1.1.7.jar from [here](#) and copy the JAR file to <ANALYTICS_HOME>\repository\components\lib directory.

Let's get started!

- [Setting up Analytics](#)
- [Analyzing the mediation statistics](#)

Setting up Analytics

1. Set the following properties in the <ESB_HOME>/repository/conf/synapse.properties file to true so that the ESB can publish mediation statistics:

```
...
mediation.flow.statistics.enable=true
mediation.flow.statistics.tracer.collect.payloads=true
mediation.flow.statistics.tracer.collect.properties=true
...
mediation.flow.statistics.collect.all=true
```

2. Start the WSO2 ESB Analytics server by going to <ANALYTICS_HOME>/bin using the Command-Line/Terminal and executing one of the following commands:

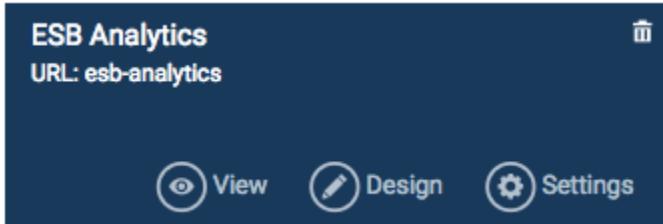
- On Linux/Mac OS: sh wso2server.sh
- On Windows: wso2server.bat --run

3. Start WSO2 ESB server from within ESB Tooling as described in [here](#) OR by navigating to <ESB_HOME>/bin using the Command-Line/Terminal and executing one of the following commands:
 - On Linux/Mac OS: sh wso2server.sh
 - On Windows: wso2server.bat --run

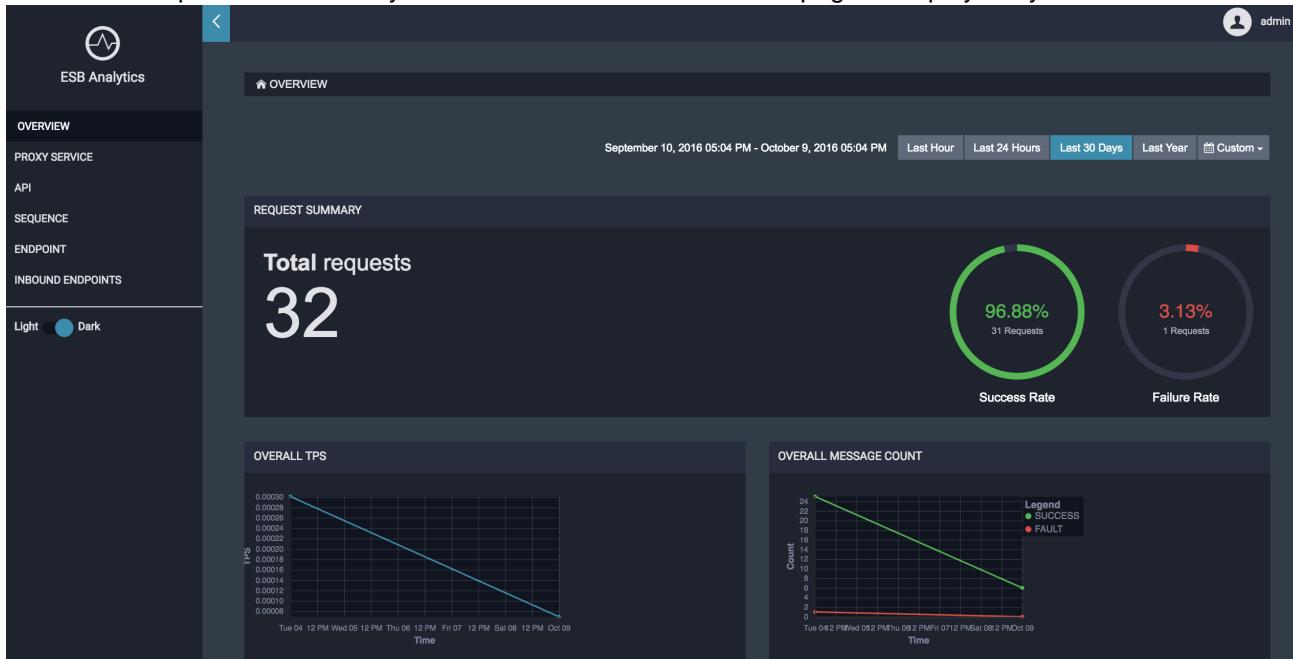
Ensure you have successfully started the Analytics server prior to starting the ESB server.

Analyzing the mediation statistics

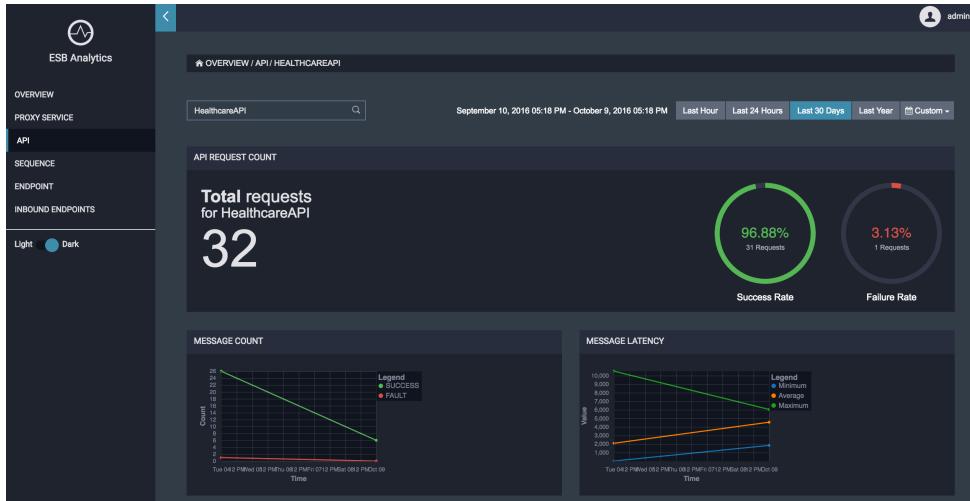
1. In a new browser window or tab, open <https://localhost:9444/carbon/> and log into the Analytics management console using admin for both the username and password.
2. On the **Main** tab, click **Analytics Dashboard** and log in using admin for both the username and password. You will then see the following:



3. Click **View** to open the ESB Analytics Dashboard. The **OVERVIEW** page is displayed by default.



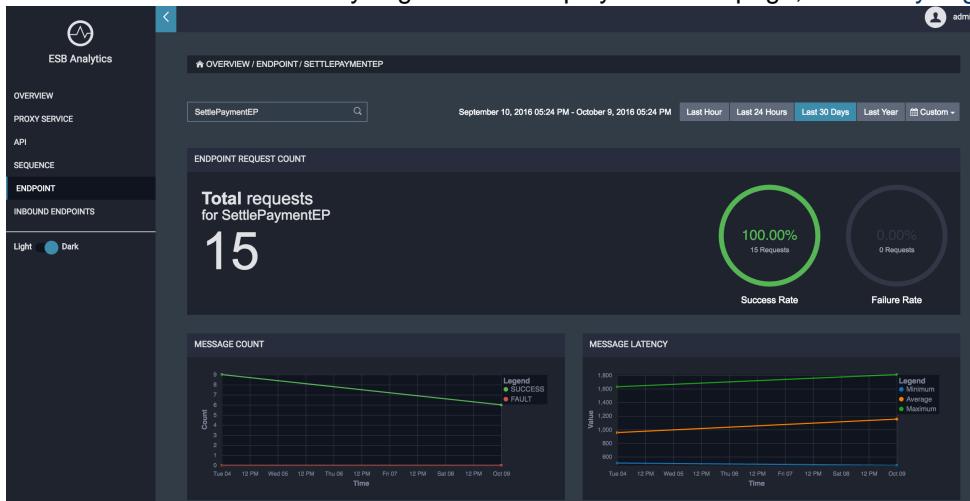
4. To view statistics for the REST API, click **API** on the left navigator and search for **HealthcareAPI**. For more information on analyzing statistics displayed on this page, see [Analyzing Statistics for REST APIs](#).



- To view statistics for an endpoint, click **ENDPOINT** on the left navigator and search for the required endpoint. You can view statistics for the following endpoints on this page:

- GrandOaksEP
- PineValleyEP
- ClemencyEP
- ChannelingFeeEP
- SettlePaymentEP

For more information on analyzing statistics displayed on this page, see [Analyzing Statistics for Endpoints](#).



You have now explored how to use WSO2 ESB Analytics dashboard.

Deep Dive

An ESB user configures how messages flow through the system, creates proxy services to integrate the back-end services that process messages, and essentially wires together the ESB. This section describes these tasks in the following sections:

- Installation Guide
- Product Administration
- Enterprise Integration Patterns
- WSO2 ESB Tooling
- Triggering Messages
- Mediating Messages
- Working with Endpoints
- Working with Web Services
- Working with Templates
- Working with Transports
- Working with Modules
- WSO2 ESB Analytics
- Cloud Services Gateway (CSG)
- Error Handling
- Java Message Service (JMS) Support
- JSON Support
- REST Support
- WebSocket Support
- Integrating with WSO2 BAM, WSO2 DAS and WSO2 CEP
- Integrating with Other Technologies
- Extending the ESB
- Reference Guide

Installation Guide

Installing WSO2 ESB is very fast and easy. Before you begin, be sure you have met the installation prerequisites, and then follow the installation instructions for your platform.

- Installation Prerequisites
- Installing on Linux
- Installing on Solaris
- Installing on Windows
- Installing as a Windows Service
- Installing as a Linux Service
- Running the Product

Installation Prerequisites

Prior to installing any WSO2 Carbon-based product, it is necessary to have the appropriate prerequisite software installed on your system. Verify that the computer has the supported operating system and development platforms before starting the installation.

The config-validation.xml file in the <PRODUCT_HOME>/repository/conf/etc directory contains a list of recommended system parameters, which are validated against your system when the server is started. See [Configuring config-validation.xml](#) for details.

System requirements

Memory	<ul style="list-style-type: none"> ~ 2 GB minimum ~ 1 GB heap size. This is generally sufficient to process typical SOAP messages but the requirements vary with larger message sizes and the number of messages processed concurrently.
Disk	<ul style="list-style-type: none"> ~ 1 GB, excluding space allocated for log files and databases.

Environment compatibility

- All WSO2 Carbon-based products are Java applications that can be run on **any platform that is Oracle JDK 1.7.*/1.8.* compliant**. We do not recommend OpenJDK as we do not support it or test our products with it.

Note

WSO2 ESB is also compatible with IBM JDK 1.7.*/1.8.*. For more information on JDKs that WSO2 products are tested with, see [Tested JDKs](#).

If you want to start WSO2 ESB with IBM JDK, edit the <ESB_HOME>/repository/conf/security/Owasp.CsrfGuard.Carbon.properties file and replace the line org.owasp.csrfguard.PRNG.Provider=SUN with org.owasp.csrfguard.PRNG.Provider=IBMJCE.

- All WSO2 Carbon-based products are generally compatible with most common DBMSs. The embedded H2 database is suitable for development, testing, and some production environments. For most enterprise production environments, however, we recommend you use an industry-standard RDBMS such as Oracle, PostgreSQL, MySQL, MS SQL, etc. For more information, see [Working with Databases](#) in the Administration Guide. Additionally, we do not recommend the H2 database as a user store.
- It is **not recommended to use Apache DS** in a production environment due to scalability issues. Instead, use an LDAP like OpenLDAP for user management.
- For environments that WSO2 products are tested with, see [Compatibility of WSO2 Products](#).
- If you have difficulty in setting up any WSO2 product in a specific platform or database, please [contact us](#).

Required applications

The following applications are required for running the ESB and its samples or for building from the source code. Mandatory installs are marked with *.

Application	Purpose	Version	Download Links
-------------	---------	---------	----------------

Oracle Java SE Development Kit (JDK)*	<ul style="list-style-type: none"> To launch the product as each product is a Java application. <div style="border: 1px solid #f0c080; padding: 10px; margin-top: 10px;"> <p style="text-align: center;">Note</p> <p>To launch WSO2 ESB, you need to have Oracle JDK 1.7.*/1.8.*. You cannot launch WSO2 ESB with Oracle JDK 1.6.* or lower.</p> </div> <ul style="list-style-type: none"> To build the product from the source distribution (both JDK and Apache Maven are required). To run Apache Ant. 	1.7 or later / 1.8.*	http://java.sun.com/javase/downloads/index.jsp
Apache Ant	<ul style="list-style-type: none"> To compile and run the product samples. 	1.7.0 or later	http://ant.apache.org
Apache Maven	<ul style="list-style-type: none"> To build the product from the source distribution (both JDK and Apache Maven are required). If you are installing by downloading and extracting the binary distribution instead of building from the source code, you do not need to install Maven. 	3.0.x	http://maven.apache.org
Web Browser	<ul style="list-style-type: none"> To access each product's Management Console. The Web Browser must be JavaScript enabled to take full advantage of the Management console. <div style="border: 1px solid #f0c080; padding: 10px; margin-top: 10px;"> <p>On Windows Server 2003, you must not go below the medium security level in Internet Explorer 6.x.</p> </div>		

You are now ready to install WSO2 ESB. Click one of the following links for instructions:

- [Installing on Linux](#)
- [Installing on Solaris](#)
- [Installing on Windows](#)
- [Installing as a Windows Service](#)

Installing on Linux

Follow these instructions to install WSO2 ESB on Linux.

Install the required applications

1. Establish an SSH connection to the Linux machine or log in on the text Linux console.
2. Be sure your system meets the [installation prerequisites](#).

Installing the ESB

1. If you have not done so already, [download](#) the latest version of the ESB.
2. Extract the archive file to a dedicated directory for the ESB, which will hereafter be referred to as <PRODUCT_HOME>.

Setting JAVA_HOME

You must set your JAVA_HOME environment variable to point to the directory where the Java Development Kit (JDK) is installed on the computer.

Environment variables are global system variables accessible by all the processes running under the operating system.

1. In your home directory, open the BASHRC file in your favorite Linux text editor, such as vi, emacs, pico, or mcedit.
2. Add the following two lines at the bottom of the file, replacing /usr/java/jdk1.8.0_25 with the actual directory where the JDK is installed.

```
export JAVA_HOME=/usr/java/jdk1.8.0_25
export PATH=${JAVA_HOME}/bin:${PATH}
```

3. Save the file.

If you do not know how to work with text editors in a Linux SSH session, run the following command:

```
cat >> .bashrc
```

Paste the string from the clipboard and press "Ctrl+D."

4. To verify that the JAVA_HOME variable is set correctly, execute the following command:

```
echo $JAVA_HOME
```

The system returns the JDK installation path.

Setting system properties

If you need to set additional system properties when the server starts, you can take the following approaches:

- **Set the properties from a script.** Setting your system properties in the startup script is ideal, because it ensures that you set the properties every time you start the server. To avoid having to modify the script each time you upgrade, the best approach is to create your own startup script that wraps the WSO2 startup script and adds the properties you want to set, rather than editing the WSO2 startup script directly.
- **Set the properties from an external registry.** If you want to access properties from an external registry, you could create Java code that reads the properties at runtime from that registry. Be sure to store sensitive data such as username and password to connect to the registry in a properties file instead of in the Java code and secure the properties file using [secure vault](#).

Note: When using SUSE Linux, it ignores `/etc/resolv.conf` and only looks at the `/etc/hosts` file. This means that the server will throw an exception on startup if you have not specified anything besides `localhost`. To avoid this error, add the following line above `127.0.0.1 localhost` in the `/etc/hosts` file:
`:<ip_address> <machine_name> localhost`

You are now ready to [run the product](#).

Installing on Solaris

Follow these instructions to install WSO2 ESB on Solaris.

Installing the supporting applications

1. Establish an SSH connection to the Solaris machine or log in on the text console.
2. Be sure your system meets the [installation prerequisites](#).

Installing the ESB

1. If you have not done so already, [download](#) the latest version of the ESB.
2. Extract the archive file to a dedicated directory for the ESB, which will hereafter be referred to as `<PRODUCT_HOME>`.

Setting JAVA_HOME

You must set your `JAVA_HOME` environment variable to point to the directory where the Java Development Kit (JDK) is installed on the computer.

Environment variables are global system variables accessible by all the processes running under the operating system.

1. In your home directory, open the `BASHRC` file in your favorite text editor, such as `vi`, `emacs`, `pico`, or `mcedit`.
2. Add the following two lines at the bottom of the file, replacing `/usr/java/jdk1.6.0_25` with the actual directory where the JDK is installed.

```
export JAVA_HOME=/usr/java/jdk1.6.0_25
export PATH=${JAVA_HOME}/bin:${PATH}
```

The file should now look like this:

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

# User specific aliases and functions
export JAVA_HOME=/usr/java/jdk1.6.0_25
export PATH=${JAVA_HOME}/bin:${PATH}
export M2_HOME=/opt/apache-maven-3.0.3
export PATH=${M2_HOME}/bin:${PATH}
```

3. Save the file.

If you do not know how to work with text editors in an SSH session, run the following command:

```
cat >> .bashrc
```

Paste the string from the clipboard and press "Ctrl+D."

4. To verify that the `JAVA_HOME` variable is set correctly, execute the following command:

```
echo $JAVA_HOME
```

```
[suncoma@wso2 ~]$ echo $JAVA_HOME
/usr/java/jdk1.6.0_25
[suncoma@wso2 ~]$
```

The system returns the JDK installation path.

Setting system properties

If you need to set additional system properties when the server starts, you can take the following approaches:

- **Set the properties from a script.** Setting your system properties in the startup script is ideal, because it ensures that you set the properties every time you start the server. To avoid having to modify the script each time you upgrade, the best approach is to create your own startup script that wraps the WSO2 startup script and adds the properties you want to set, rather than editing the WSO2 startup script directly.
- **Set the properties from an external registry.** If you want to access properties from an external registry, you could create Java code that reads the properties at runtime from that registry. Be sure to store sensitive data such as username and password to connect to the registry in a properties file instead of in the Java code and secure the properties file using [secure vault](#).

You are now ready to [run the product](#).

Installing on Windows

Before you begin, please see our [compatibility matrix](#) to find out if this version of the product is fully tested on Windows.

Follow these instructions to install WSO2 ESB on Windows. Alternatively, you can [install it as a Windows service](#).

Installing the required applications

1. Ensure that your system meets the [Installation Prerequisites](#) Java Development Kit (JDK) is essential to run the product.
2. Ensure that the `PATH` environment variable is set to "`C:\Windows\System32`", because the `findstr` Windows .exe file is stored in this path.

Installing the ESB

1. If you have not done so already, [download](#) the latest version of the ESB.
2. Extract the archive file to a dedicated directory for the ESB, which will hereafter be referred to as `<PRODUCT_HOME>`.

Setting JAVA_HOME

You must set your `JAVA_HOME` environment variable to point to the directory where the Java Development Kit (JDK) is installed on the computer. Typically, the JDK is installed in a directory under `C:\Program Files\Java`, such as `C:\Program Files\Java\jdk1.6.0_27`. If you have multiple versions installed, choose the latest one, which

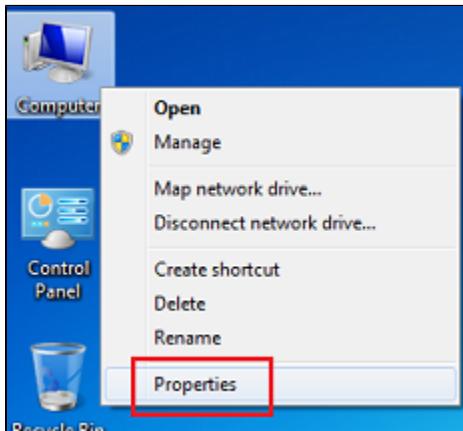
you can find by sorting by date.

Environment variables are global system variables accessible by all the processes running under the operating system. You can define an environment variable as a system variable, which applies to all users, or as a user variable, which applies only to the user who is currently logged in.

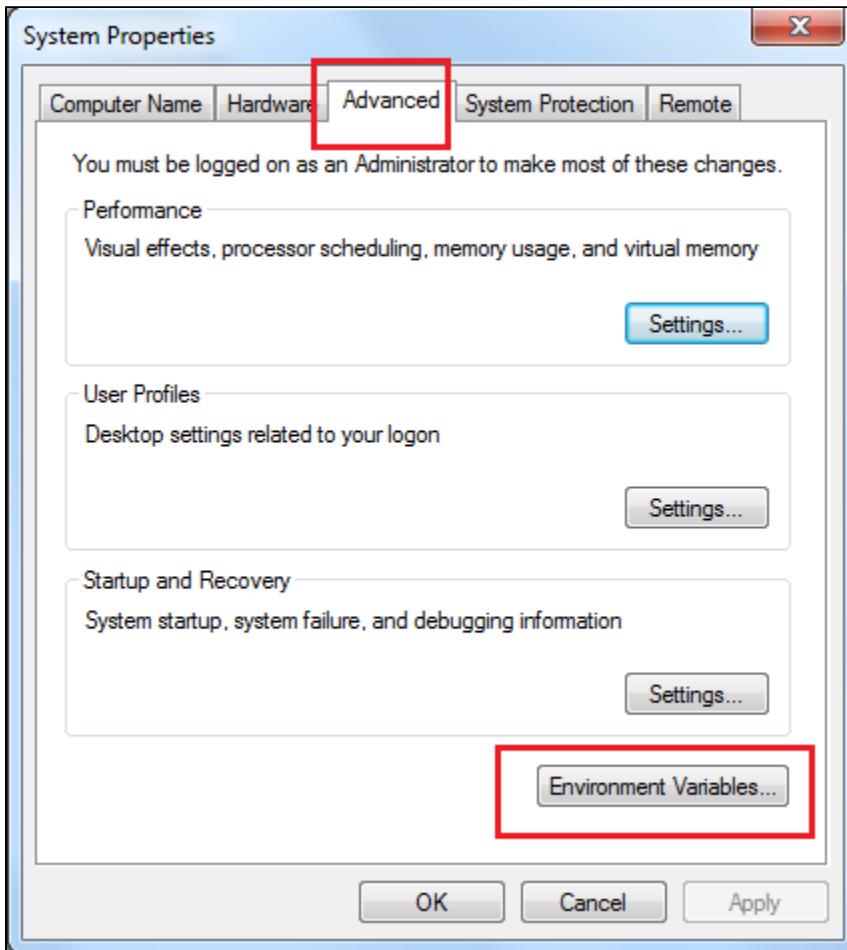
You can set JAVA_HOME using the System Properties, as described below. Alternatively, if you just want to set JAVA_HOME temporarily in the current command prompt window, [set it at the command prompt](#).

Setting JAVA_HOME using the System Properties

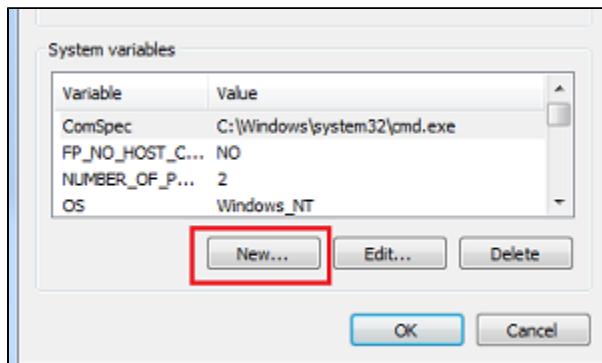
1. Right-click the "My Computer" icon on the desktop and choose **Properties**.



2. In the System Properties window, click the **Advanced** tab, and then click the **Environment Variables** button.



3. Click the New button under "System variables" (for all users) or under "User variables" (just for the user who is currently logged in).



4. Enter the following information:

- In the "Variable name" field, enter: JAVA_HOME
- In the "Variable value" field, enter the installation path of the Java Development Kit, such as: c:\Program Files\Java\jdk1.6.0_27

5. Click OK.

The JAVA_HOME variable is now set and will apply to any subsequent command prompt windows you open. If you have existing command prompt windows running, you must close and reopen them for the JAVA_HOME variable to take effect, or manually set the JAVA_HOME variable in those command prompt windows as described in the next section. To verify that the JAVA_HOME variable is set correctly, open a command window (from the **Start** menu, click **Run**, and then type **CMD** and click **Enter**) and execute the following command:

```
set JAVA_HOME
```

The system returns the JDK installation path.

Setting JAVA_HOME temporarily using the Windows command prompt (CMD)

You can temporarily set the JAVA_HOME environment variable within a Windows command prompt window (CMD). This is useful when you have an existing command prompt window running and you do not want to restart it.

1. In the command prompt window, enter the following command where <JDK_INSTALLATION_PATH> is the JDK installation directory and press **Enter**:

```
set JAVA_HOME=<JDK_INSTALLATION_PATH>
```

For example:

```
set JAVA_HOME=c:\Program Files\java\jdk1.6.0_27
```

The JAVA_HOME variable is now set for the current CMD session only.

2. To verify that the JAVA_HOME variable is set correctly, execute the following command:

```
set JAVA_HOME
```

The system returns the JDK installation path.

Setting system properties

If you need to set additional system properties when the server starts, you can take the following approaches:

- **Set the properties from a script.** Setting your system properties in the startup script is ideal, because it ensures that you set the properties every time you start the server. To avoid having to modify the script each

time you upgrade, the best approach is to create your own startup script that wraps the WSO2 startup script and adds the properties you want to set, rather than editing the WSO2 startup script directly.

- **Set the properties from an external registry.** If you want to access properties from an external registry, you could create Java code that reads the properties at runtime from that registry. Be sure to store sensitive data such as username and password to connect to the registry in a properties file instead of in the Java code and secure the properties file using [secure vault](#).

You are now ready to [run the product](#).

Installing as a Windows Service

WSO2 Carbon and any Carbon-based product can be run as a Windows service as described in the following sections:

- [Prerequisites](#)
- Setting up the YAJSW wrapper configuration file
- Setting up CARBON_HOME
- Running the product in console mode
- Working with the WSO2CARBON service

Prerequisites

- Install JDK and set up the JAVA_HOME environment variable. For more information, see [Installation Prerequisites](#).
- Download and install a service wrapper library to use for running your WSO2 product as a Windows service. WSO2 recommends Yet Another Java Service Wrapper ([YAJSW](#)) version 11.03, and several WSO2 products provide a default wrapper.conf file in their <PRODUCT_HOME>/bin/yajsw/ directory. The instructions below describe how to set up this file.

Setting up the YAJSW wrapper configuration file

The configuration file used for wrapping Java Applications by YAJSW is wrapper.conf, which is located in the <YAJSW_HOME>/conf/ directory and in the <PRODUCT_HOME>/bin/yajsw/ directory of many WSO2 products. Following is the minimal wrapper.conf configuration for running a WSO2 product as a Windows service. Open your wrapper.conf file, set its properties as follows, and save it in <YAJSW_HOME>/conf/ directory.

If you want to set additional properties from an external registry at runtime, store sensitive information like usernames and passwords for connecting to the registry in a properties file and secure it with [secure vault](#).

Minimal wrapper.conf configuration

```
#####
# working directory
#####
wrapper.working.dir=${carbon_home} \\
# Java Main class.
# YAJSW: default is "org.rzo.yajsw.app.WrapperJVMMain"
# DO NOT SET THIS PROPERTY UNLESS YOU HAVE YOUR OWN IMPLEMENTATION
# wrapper.java.mainclass=
#####
# tmp folder
# yajsw creates temporary files named in... out... err... jna...
# per default these are placed in jna.tmpdir.
# jna.tmpdir is set in setenv batch file to <yajsw>/tmp
#####
wrapper.tmp.path = ${jna_tmpdir}
#####
# Application main class or native executable
```

```

# One of the following properties MUST be defined
#*****
# Java Application main class
wrapper.java.app.mainclass=org.wso2.carbon.bootstrap.Bootstrap
# Log Level for console output. (See docs for log levels)
wrapper.console.loglevel=INFO
# Log file to use for wrapper output logging.
wrapper.logfile=${wrapper_home}\log\wrapper.log
# Format of output for the log file. (See docs for formats)
#wrapper.logfile.format=LPTM
# Log Level for log file output. (See docs for log levels)
#wrapper.logfile.loglevel=INFO
# Maximum size that the log file will be allowed to grow to before
# the log is rolled. Size is specified in bytes. The default value
# of 0, disables log rolling by size. May abbreviate with the 'k' (kB) or
# 'm' (mB) suffix. For example: 10m = 10 megabytes.
# If wrapper.logfile does not contain the string ROLLOUT it will be automatically
added as suffix of the file name
wrapper.logfile.maxsize=10m
# Maximum number of rolled log files which will be allowed before old
# files are deleted. The default value of 0 implies no limit.
wrapper.logfile.maxfiles=10
# Title to use when running as a console
wrapper.console.title="WSO2 Carbon"
#*****
# Wrapper Windows Service and Posix Daemon Properties
#*****
# Name of the service
wrapper.ntservice.name="WSO2CARBON"
# Display name of the service
wrapper.ntservice.displayname="WSO2 Carbon"
# Description of the service
wrapper.ntservice.description="Carbon Kernel"
#*****
# Wrapper System Tray Properties
#*****
# enable system tray
wrapper.tray = true
# TCP/IP port. If none is defined multicast discovery is used to find the port
# Set the port in case multicast is not possible.
wrapper.tray.port = 15002
#*****
# Exit Code Properties
# Restart on non zero exit code
#*****
wrapper.on_exit.0=SHUTDOWN
wrapper.on_exit.default=RESTART
#*****
# Trigger actions on console output
#*****
# On Exception show message in system tray
wrapper.filter.trigger.0=Exception
wrapper.filter.script.0=scripts\trayMessage.gv
wrapper.filter.script.0.args=Exception
#*****
# genConfig: further Properties generated by genConfig
#*****
placeHolderSoGenPropsComeHere=
wrapper.java.command = ${java_home}\bin\java

```

```

wrapper.java.classpath.1 = ${java_home}\lib\tools.jar
wrapper.java.classpath.2 = ${carbon_home}\bin\*.jar
wrapper.app.parameter.1 = org.wso2.carbon.bootstrap.Bootstrap
wrapper.app.parameter.2 = RUN
wrapper.java.additional.1 = -Xbootclasspath/a:${carbon_home}\lib\xboot\*.jar
wrapper.java.additional.2 = -Xms256m
wrapper.java.additional.3 = -Xmx1024m
wrapper.java.additional.4 = -XX:MaxPermSize=256m
wrapper.java.additional.5 = -XX:+HeapDumpOnOutOfMemoryError
wrapper.java.additional.6 =
-XX:HeapDumpPath=${carbon_home}\repository\logs\heap-dump.hprof
wrapper.java.additional.7 = -Dcom.sun.management.jmxremote
wrapper.java.additional.8 =
-Djava.endorsed.dirs=${carbon_home}\lib\endorsed;${java_home}\jre\lib\endorsed
wrapper.java.additional.9 = -Dcarbon.registry.root=\
wrapper.java.additional.10 = -Dcarbon.home=${carbon_home}
wrapper.java.additional.11 = -Dwso2.server.standalone=true
wrapper.java.additional.12 = -Djava.command=${java_home}\bin\java
wrapper.java.additional.13 = -Djava.io.tmpdir=${carbon_home}\tmp
wrapper.java.additional.14 = -Dcatalina.base=${carbon_home}\lib\tomcat
wrapper.java.additional.15 =
-Djava.util.logging.config.file=${carbon_home}\repository\conf\log4j.properties
wrapper.java.additional.16 = -Dcarbon.config.dir.path=${carbon_home}\repository\conf

wrapper.java.additional.17 = -Dcarbon.logs.path=${carbon_home}\repository\logs
wrapper.java.additional.18 =
-Dcomponents.repo=${carbon_home}\repository\components\plugins
wrapper.java.additional.19 = -Dconf.location=${carbon_home}\repository\conf
wrapper.java.additional.20 =
-Dcom.atomikos.icatch.file=${carbon_home}\lib\transactions.properties
wrapper.java.additional.21 = -Dcom.atomikos.icatch.hide_init_file_path=true
wrapper.java.additional.22 =
-Dorg.apache.jasper.runtime.BodyContentImpl.LIMIT_BUFFER=true

```

```
wrapper.java.additional.23 = -Dcom.sun.jndi.ldap.connect.pool.authentication=simple
wrapper.java.additional.24 = -Dcom.sun.jndi.ldap.connect.pool.timeout=3000
wrapper.java.additional.25 = -Dorg.terracotta.quartz.skipUpdateCheck=true
```

Setting up CARBON_HOME

Extract the Carbon-based product that you want to run as a Windows service, and then set the Windows environment variable `CARBON_HOME` to the extracted product directory location. For example, if you want to run ESB 4.5.0 as a Windows service, you would set `CARBON_HOME` to the extracted `wso2esb-4.5.0` directory.



Running the product in console mode

You will now verify that YAJSW is configured correctly for running the Carbon-based product as a Windows service.

1. Open a Windows command prompt and go to the `<YAJSW_HOME>/bat/` directory. For example:

```
cd C:\Documents and Settings\yajsw_home\bat
```

2. Start the wrapper in console mode using the following command:

```
runConsole.bat
```

For example:

```
C:\Documents and Settings\yajsw_home\bat>runConsole.bat
```

If the configurations are set properly for YAJSW, you will see console output similar to the following and can now access the WSO2 management console from your web browser via <https://localhost:9443/carbon>.

```
C:\Documents and Settings\yajsw_home\bat>runConsole.bat
C:\Documents and Settings\yajsw_home\bat>cd C:\Documents and Settings\yajsw_home\bat\
C:\Documents and Settings\yajsw_home\bat>call setenv.bat
"java" -Xmx30m -Djna_tmpdir="C:\Documents and Settings\yajsw_home\bat\..\tmp" -
jar "C:\Documents and Settings\yajsw_home\bat\..\wrapper.jar" -c "C:\Documents and Settings\yajsw_home\bat\..\conf\wrapper.conf"
YAJSW: yajsw-stable-11.03
OS : Windows XP/5.2/amd64
JVM : Oracle Corporation/1.7.0_06
Dec 30, 2012 11:12:27 AM org.apache.commons.vfs2.VfsLog info
INFO: Using "C:\DOCUMENTS\ADMINI\LOCALS\Temp\vfs_cache" as temporary files store.
```

Working with the WSO2CARBON service

To install the Carbon-based product as a Windows service, execute the following command in the <YAJSW_HOME>/bat/ directory:

```
installService.bat
```

The console will display a message confirming that the WSO2CARBON service was installed.

```
C:\Documents and Settings\yajsw_home\bat>installService.bat

C:\Documents and Settings\yajsw_home\bat>cd C:\Documents and Settings\yajsw_home\bat\

C:\Documents and Settings\yajsw_home\bat>call setenv.bat
"java" -Xmx30m -Djna_tmpdir="C:\Documents and Settings\yajsw_home\bat\..\tmp" -
jar "C:\Documents and Settings\yajsw_home\bat\..\wrapper.jar" -i "C:\Documents and Settings\yajsw_home\bat\..\conf\wrapper.conf"
YAJSW: yajsw-stable-11.03
OS : Windows XP/5.2/amd64
JVM : Oracle Corporation/1.7.0_06
Dec 30, 2012 12:51:42 PM org.apache.commons vfs2.UfsLog info
INFO: Using "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\ufs_cache" as temporary files store.
platform null
***** INSTALLING "WSO2CARBON" *****

Service "WSO2CARBON" installed
Press any key to continue . . .
```

To start the service, execute the following command in the same console window:

```
startService.bat
```

The console will display a message confirming that the WSO2CARBON service was started.

```
C:\Documents and Settings\yajsw_home\bat>startService.bat

C:\Documents and Settings\yajsw_home\bat>cd C:\Documents and Settings\yajsw_home\bat\

C:\Documents and Settings\yajsw_home\bat>call setenv.bat
"java" -Xmx30m -Djna_tmpdir="C:\Documents and Settings\yajsw_home\bat\..\tmp" -
jar "C:\Documents and Settings\yajsw_home\bat\..\wrapper.jar" -t "C:\Documents and Settings\yajsw_home\bat\..\conf\wrapper.conf"
YAJSW: yajsw-stable-11.03
OS : Windows XP/5.2/amd64
JVM : Oracle Corporation/1.7.0_06
Dec 30, 2012 1:09:00 PM org.apache.commons vfs2.UfsLog info
INFO: Using "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\ufs_cache" as temporary files store.
platform null
***** STARTING "WSO2CARBON" *****

Service "WSO2CARBON" started
Press any key to continue . . .

C:\Documents and Settings\yajsw_home\bat>
```

To stop the service, execute the following command in the same console window:

```
stopService.bat
```

The console will display a message confirming that the WSO2CARBON service has stopped.

```
C:\Documents and Settings\yajsw_home\bat>stopService.bat
C:\Documents and Settings\yajsw_home\bat>cd C:\Documents and Settings\yajsw_home\bat\
C:\Documents and Settings\yajsw_home\bat>call setenv.bat
"java" -Xmx30m -Djna_tmpdir="C:\Documents and Settings\yajsw_home\bat\..\tmp" -
jar "C:\Documents and Settings\yajsw_home\bat\..\wrapper.jar" -p "C:\Documents and Settings\yajsw_home\bat\..\conf\wrapper.conf"
YAJSW: yajsw-stable-11.03
OS : Windows XP/5.2/amd64
JVM : Oracle Corporation/1.7.0_06
Dec 30, 2012 11:11:31 AM org.apache.commons vfs2.UfsLog info
INFO: Using "C:\DOCUMENT\ADMINI\LOCALS\Temp\ufs_cache" as temporary files store.
platform null
***** STOPPING "WSO2CARBON" *****
Service "WSO2CARBON" stopped
Press any key to continue . . .
```

To uninstall the service, execute the following command in the same console window:

```
uninstallService.bat
```

The console will display a message confirming that the WSO2CARBON service was removed.

```
C:\Documents and Settings\yajsw_home\bat>uninstallService.bat
C:\Documents and Settings\yajsw_home\bat>cd C:\Documents and Settings\yajsw_home\bat\
C:\Documents and Settings\yajsw_home\bat>call setenv.bat
"java" -Xmx30m -Djna_tmpdir="C:\Documents and Settings\yajsw_home\bat\..\tmp" -
jar "C:\Documents and Settings\yajsw_home\bat\..\wrapper.jar" -r "C:\Documents and Settings\yajsw_home\bat\..\conf\wrapper.conf"
YAJSW: yajsw-stable-11.03
OS : Windows XP/5.2/amd64
JVM : Oracle Corporation/1.7.0_06
Dec 30, 2012 1:19:14 PM org.apache.commons vfs2.UfsLog info
INFO: Using "C:\DOCUMENT\ADMINI\LOCALS\Temp\ufs_cache" as temporary files store.
platform null
***** REMOVING "WSO2CARBON" *****
Service "WSO2CARBON" removed
Press any key to continue . . .
C:\Documents and Settings\yajsw_home\bat>
```

Installing as a Linux Service

WSO2 Carbon and any Carbon-based product can be run as a Linux service as described in the following sections:

- Prerequisites
- Setting up CARBON_HOME
- Running the product as a Linux service

Prerequisites

Install JDK and set up the JAVA_HOME environment variable. For more information, see [Installation Prerequisites](#).

Setting up CARBON_HOME

Extract the WSO2 product that you want to run as a Linux service and set the environment variable CARBON_HOME to

o the extracted product directory location.

Running the product as a Linux service

1. To run the product as a service, create a startup script and add it to the boot sequence. The basic structure of the startup script has three parts (i.e., start, stop and restart) as follows:

```
#!/bin/bash

case "$1" in
start)
    echo "Starting Service"
;;
stop)
    echo "Stopping Service"
;;
restart)
    echo "Restarting Service"
;;
*)
    echo $"Usage: $0 {start|stop|restart}"
exit 1
esac
```

For example, given below is a startup script written for WSO2 Application Server 5.2.0:

```
#! /bin/sh
export JAVA_HOME="/usr/lib/jvm/jdk1.7.0_07"

startcmd='/opt/WSO2/wso2as-5.2.0/bin/wso2server.sh start > /dev/null &'
restartcmd='/opt/WSO2/wso2as-5.2.0/bin/wso2server.sh restart > /dev/null &'
stopcmd='/opt/WSO2/wso2as-5.2.0/bin/wso2server.sh stop > /dev/null &

case "$1" in
start)
    echo "Starting WSO2 Application Server ..."
    su -c "${startcmd}" user1
;;
restart)
    echo "Re-starting WSO2 Application Server ..."
    su -c "${restartcmd}" user1
;;
stop)
    echo "Stopping WSO2 Application Server ..."
    su -c "${stopcmd}" user1
;;
*)
    echo "Usage: $0 {start|stop|restart}"
exit 1
esac
```

In the above script, the server is started as a user by the name user1 rather than the root user. For example,
`su -c "${startcmd}" user1`

2. Add the script to `/etc/init.d/` directory.

If you want to keep the scripts in a location other than `/etc/init.d/` folder, you can add a symbolic link to the script in `/etc/init.d/` and keep the actual script in a separate location. Say your script name is `appserver` and it is in `/opt/WSO2/` folder, then the commands for adding a link to `/etc/init.d/` is as follows:

- Make executable: `sudo chmod a+x /opt/WSO2/appserver`
- Add a link to `/etc/init.d/`: `sudo ln -snf /opt/WSO2/appserver /etc/init.d/appserver`

3. Install the startup script to respective runlevels using the command `update-rc.d`. For example, give the following command for the sample script shown in step1:

```
sudo update-rc.d appserver defaults
```

The `defaults` option in the above command makes the service to start in runlevels 2,3,4 and 5 and to stop in runlevels 0,1 and 6.

A **runlevel** is a mode of operation in Linux (or any Unix-style operating system). There are several runlevels in a Linux server and each of these runlevels is represented by a single digit integer. Each runlevel designates a different system configuration and allows access to a different combination of processes.

4. You can now start, stop and restart the server using `service <service name> {start|stop|restart}` command. You will be prompted for the password of the user (or root) who was used to start the service.

Running the Product

To run WSO2 products, you start the product server at the command line. You can then run the Management Console to configure and manage the product.

The Management Console uses the default HTTP-NIO transport, which is configured in the `<PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml` file. (`<PRODUCT_HOME>` is the directory where you installed the WSO2 product you want to run.) You must properly configure the HTTP-NIO transport in this file to access the Management Console. For more information on the HTTP-NIO transport, see the related topics section at the bottom of this page.

The following sections describe how to run the product.

- Starting the server
- Accessing the Management Console
- Stopping the server

Starting the server

Follow the instructions below to start your WSO2 product based on the Operating System you use.

On Windows/Linux/Mac OS

To start the server, you run `<PRODUCT_HOME>/bin/wso2server.bat` (on Windows) or `<PRODUCT_HOME>/bin/wso2server.sh` (on Linux/Mac OS) from the command prompt as described below. Alternatively, you can install and run the server as a Windows or Linux service (see the related topics section at the end of this page).

1. Open a command prompt by following the instructions below.
 - On Windows: Click **Start -> Run**, type `cmd` at the prompt, and then press **Enter**.
 - On Linux/Mac OS: Establish an SSH connection to the server, log on to the text Linux console, or open

- a terminal window.
2. Navigate to the <PRODUCT_HOME>/bin/ directory using the Command Prompt.
 3. Execute one of the following commands:
 - To start the server in a typical environment:
 - On Windows: wso2server.bat --run
 - On Linux/Mac OS: sh wso2server.sh
 - To start the server in the background mode of Linux: sh wso2server.sh start
To stop the server running in this mode, you will enter: sh wso2server.sh stop
 - To provide access to the production environment without allowing any user group (including admin) to log in to the Management Console:
 - On Windows: wso2server.bat --run -DworkerNode
 - On Linux/Mac OS: sh wso2server.sh -DworkerNode
 - To check for additional options you can use with the startup commands, type -help after the command, such as:
sh wso2server.sh -help (see the related topics section at the end of this page).
 4. The operation log appears in the command window. When the product server has successfully started, the log displays the message "WSO2 Carbon started in 'n' seconds".

On Solaris

To start the server, you run <PRODUCT_HOME>/bin/wso2server.sh from the Command Prompt as described below.

Following instructions are tested on an Oracle Solaris 10 8/11 x86 environment.

1. Click **Launch -> Run Applications**, type dtterm at the Prompt, and then press **Enter**, to open a Command Prompt.
2. Navigate to the <PRODUCT_HOME>/bin/ directory using the Command Prompt.
3. Execute the following command: bash wso2server.sh
4. The operation log appears in the command window. When the product server has successfully started, the log displays the message "WSO2 Carbon started in 'n' seconds".

If you are starting the product in service/nohup mode in Solaris, do the following:

1. Update the <PRODUCT_HOME>/bin/wso2server.sh file as follows:
 - a. Search for the following occurrences: **nohup sh "\$CARBON_HOME"/bin/wso2server.sh \$args > /dev/null 2>&1 &**
 - b. Replace those occurrences with the following: **nohup bash "\$CARBON_HOME"/bin/wso2server.sh \$args > /dev/null 2>&1 &**

The only change is replacing sh with bash. This is required only for Solaris.

2. Update your **PATH** variable to have **/usr/xpg4/bin/sh** as the first element. This is because **/usr/xpg4/bin/sh** contains an **sh** shell that is newer than the default **sh** shell. You can set this variable as a system property in the wso2server.sh script or you can run the following command on a terminal:

```
export PATH=/usr/xpg4/bin/sh:$PATH
```

3. Start the product by following the above instructions.

Accessing the Management Console

Once the server has started, you can run the Management Console by typing its URL in a Web browser. The following sections provide more information about running the Management Console:

- Working with the URL
- Signing in
- Getting help
- Configuring the session time-out
- Restricting access to the Management Console and Web applications

Working with the URL

The URL appears next to "Mgt Console URL" in the start script log that is displayed in the command window. For example:

```
[2014-12-04 17:53:26,547] INFO {org.wso2.carbon.core.internal.StartupFinalizerServiceComponent} - WSO2 Carbon started in 45 sec
[2014-12-04 17:53:26,787] INFO {org.wso2.carbon.ui.internal.CarbonUIServiceComponent} - Mgt Console URL : https://localhost:9443/carbon/
```

The URL should be in the following format: `https://<Server Host>:9443/carbon`

You can use this URL to access the Management Console on this computer from any other computer connected to the Internet or LAN. When accessing the Management Console from the same server where it is installed, you can type `localhost` instead of the IP address as follows: `https://localhost:9443/carbon`

You can change the Management Console URL by modifying the value of the `<MgtHostName>` property in the `<PRODUCT_HOME>/repository/conf/carbon.xml` file. When the host is internal or not resolved by a DNS, map the hostname alias to its IP address in the `/etc/hosts` file of your system, and then enter that alias as the value of the `<MgtHostName>` property in `carbon.xml`. For example:

```
In /etc/hosts:
127.0.0.1      localhost

In carbon.xml:
<MgtHostName>localhost</MgtHostName>
```

Signing in

At the sign-in screen, you can sign in to the Management Console using **admin** as both the username and password.

When the Management Console sign-in page appears, the Web browser typically displays an "insecure connection" message, which requires your confirmation before you can continue.

The Management Console is based on the HTTPS protocol, which is a combination of HTTP and SSL protocols. This protocol is generally used to encrypt the traffic from the client to server for security reasons. The certificate it works with is used for encryption only, and does not prove the server identity. Therefore, when you try to access the Management Console, a warning of untrusted connection is usually displayed. To continue working with this certificate, some steps should be taken to "accept" the certificate before access to the site is permitted. If you are using the Mozilla Firefox browser, this usually occurs only on the first access to the server, after which the certificate is stored in the browser database and marked as trusted. With other browsers, the insecure connection warning might be displayed every time you access the server.

This scenario is suitable for testing purposes, or for running the program on the company's internal networks. If you want to make the Management Console available to external users, your organization should obtain a certificate signed by a well-known certificate authority, which verifies that the server actually has the name it is accessed by and that this server actually belongs to the given organization.

Getting help

The tabs and menu items in the navigation pane on the left may vary depending on the features you have installed. To view information about a particular page, click the **Help** link at the top right corner of that page, or click the **Docs** link to open the documentation for full information on managing the product.

Configuring the session time-out

If you leave the Management Console unattended for a defined time, its login session will time out. The default timeout value is 15 minutes, but you can change this in the <PRODUCT_HOME>/repository/conf/tomcat/carbon/WEB-INF/web.xml file as follows.

```
<session-config>
    <session-timeout>15</session-timeout>
</session-config>
```

Restricting access to the Management Console and Web applications

You can restrict access to the Management Console of your product by binding the Management Console with selected IP addresses. You can either restrict access to the Management Console only, or you can restrict access to all Web applications in your server as explained below.

- To control access only to the Management Console, add the IP addresses to the <PRODUCT_HOME>/repository/conf/tomcat/carbon/META-INF/context.xml file as follows:

```
<Valve className="org.apache.catalina.valves.RemoteAddrValve"
allow="|<IP-address-02>|<IP-address-03>" />
```

The RemoteAddrValve Tomcat valve defined in this file only applies to the Management Console, and thereby all outside requests to the Management Console are blocked.

- To control access to all Web applications deployed in your server, add the IP addresses to the <PRODUCT_HOME>/repository/conf/context.xml file as follows.

```
<Valve className="org.apache.catalina.valves.RemoteAddrValve"
allow="|<IP-address-02>|<IP-address-03>" />
```

The RemoteAddrValve Tomcat valve defined in this file applies to each Web application hosted on the WSO2 product server. Therefore, all outside requests to any Web application are blocked.

- You can also restrict access to particular servlets in a Web application by adding a Remote Address Filter to the <PRODUCT_HOME>/repository/conf/tomcat/web.xml file and by mapping that filter to the servlet URL. In the Remote Address Filter that you add, you can specify the IP addresses that should be allowed to access the servlet. The following example from a web.xml file illustrates how access to the Management Console page (/carbon/admin/login.jsp) is granted only to one IP address.

```

<filter>
    <filter-name>Remote Address Filter</filter-name>
    <filter-class>org.apache.catalina.filters.RemoteAddrFilter</filter-class>
    <init-param>
        <param-name>allow</param-name>
        <param-value>127.0.0.1</param-value>
    </init-param>
</filter>

<filter-mapping>
    <filter-name>Remote Address Filter</filter-name>
    <url-pattern>/carbon/admin/login.jsp</url-pattern>
</filter-mapping>

```

Any configurations (including values defined in the <PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml file) apply to all Web applications and are globally available across the server, regardless of the host or cluster. For more information about using remote host filters, see the [Apache Tomcat documentation](#).

Stopping the server

To stop the server, press **Ctrl+C** in the command window, or click the **Shutdown/Restart** link in the navigation pane in the Management Console. If you started the server in background mode in Linux, enter the following command instead:

```
sh <PRODUCT_HOME>/bin/wso2server.sh stop
```

- [HTTP-NIO Transport](#)
- [Installing as a Windows Service](#)
- [Installing as a Linux Service](#)

Product Administration

If you are a product administrator, the following topics provide an overview of the tasks that you need to perform when working with WSO2 ESB to deploy, monitor, tune, and maintain an ESB.

[[Upgrading from a previous release](#)] [[Clustering WSO2 ESB](#)] [[Changing the default database](#)] [[Configuring users, roles and permissions](#)] [[Configuring security](#)] [[Configuring multitenancy](#)] [[Configuring the registry](#)] [[Performance tuning](#)] [[Changing the default ports](#)] [[Installing, uninstalling and managing product features](#)] [[Configuring custom proxy paths](#)] [[Customizing error pages](#)] [[Customizing the management console](#)] [[Working with proxy servers](#)] [[Using the Cloud Services Gateway\(CSG\)](#)] [[Monitoring the server](#)] [[Applying patches](#)] [[Working with Composite Applications \(C-Apps\) and artifacts](#)]

Upgrading from a previous release

If you want to upgrade data and configurations from ESB 4.9.0 to ESB 5.0.0, see [Upgrading from a Previous Release](#)

Clustering WSO2 ESB

For information on how to set up a WSO2 ESB worker/manager separated cluster and how to configure a cluster with a third-party load balancer, see [Clustered Deployment](#).

Changing the default database

By default, WSO2 products are shipped with an embedded H2 database, which is used for storing user management and registry data. We recommend the use of an industry-standard RDBMS such as Oracle, PostgreSQL, MySQL, MS SQL, etc. when you set up your production environment. You can change the default database configuration by setting up a new physical database, and then updating the configurations in the production server to connect to the new database.

For information on setting up and configuring databases, see [Working with Databases](#) in the WSO2 Administration Guide.

Configuring users, roles and permissions

The user management feature in WSO2 ESB allows you to create new users as well as to define the permission granted to each user. You can also configure the user stores that are used for storing data related to user management.

For information on how to configure user management, see [Managing Users, Roles and Permissions](#) in the WSO2 Administration Guide.

Configuring security

After you install WSO2 ESB, it is recommended to change the default security settings according to your production environment.

For information on configuring security in your server, see the following topics in the WSO2 Administration Guide.

- [Configuring Transport-Level Security](#)
- [Using Asymmetric Encryption](#)
- [Using Symmetric Encryption](#)
- [Enabling Java Security Manager](#)
- [Securing Passwords in Configuration Files](#)
- [Resolving Hostname Verification](#)

If your proxy service connects to a back-end server through a proxy server, you can enable Secure Socket Layer (SSL) tunneling through the proxy server, which prevents any intermediary proxy services from interfering with the communication. For information on this, see [Enabling SSL Tunneling through a Proxy Server](#).

Configuring multitenancy

You can create multiple tenants in your product server, so that you can maintain tenant isolation in a single server/cluster. For information on configuring multiple tenants for your server, see [Working with Multiple Tenants](#) in the WSO2 Administration Guide.

Configuring the registry

A **registry** is a content store and a metadata repository for various artifacts such as services, WSDLs and configuration files. In WSO2 products, all configurations pertaining to modules, logging, security, data sources and other service groups are stored in the registry by default. For information on setting up and configuring the registry for your server, see [Working with the Registry](#) in the WSO2 Administration Guide.

The **local registry** acts as a memory registry where you can store static content as a key-value pair, where the value could be a static entry such as a text string, XML code, or a URL. For information on local registry entries, see [Working with Local Registry Entries](#).

Performance tuning

You can optimize the performance of WSO2 ESB by configuring the ESB based on a set of recommendations from WSO2 experts.

For information on network and OS level performance tuning, Java Virtual Machine (JVM) level tuning, and WSO2 Carbon platform-level performance tuning recommendations, see [Performance Tuning](#) in the WSO2 Administration Guide.

For information on performance tuning recommendations that are specific to WSO2 ESB, see the [WSO2 ESB Performance Tuning Guide](#).

Changing the default ports

When you run multiple WSO2 products, multiple instances of the same product, or multiple WSO2 product clusters on the same server or virtual machines (VMs), you must change their default ports with an offset value to avoid port conflicts.

For instructions on configuring posts, see [Changing the Default Ports](#) in the WSO2 Administration Guide.

Installing, uninstalling and managing product features

Each WSO2 product is a collection of reusable software units called features where a single feature is a list of components and/or other feature. By default, WSO2 ESB is shipped with the features that are required for your main use cases.

For information on installing new features, or removing/updating an existing feature, see [Working with Features](#) in the WSO2 Administration Guide.

Configuring custom proxy paths

This feature is particularly useful when multiple WSO2 products (fronted by a proxy server) are hosted under the same domain name. By adding a custom proxy path you can host all products under a single domain and assign proxy paths for each product separately.

For instructions on configuring custom proxy paths, see [Adding a Custom Proxy Path](#) in the WSO2 Administration Guide.

Customizing error pages

You can make sure that sensitive information about the server is not revealed in error messages, by customizing the error pages in your product.

For instructions, see [Customizing Error Pages](#) in the WSO2 Administration Guide.

Customizing the management console

Some of the WSO2 products, such as WSO2 ESB consist of a web user interface, which is known as the management console. This allows administrators to configure, monitor, tune, and maintain the product using a simple interface. You can customize the look and feel of the management console for your product.

For instructions, see [Customizing the Management Console](#) in the WSO2 Administration Guide.

Working with proxy servers

When using WSO2 ESB, there can be scenarios where you need to configure the ESB to route messages through a proxy server.

For information on how to work with proxy servers, see [Working with Proxy Servers](#).

Using the Cloud Services Gateway(CSG)

When using the WSO2 ESB, there can be scenarios where you need to expose a private service to the public through a cooperate firewall. You can use the Cloud Services Gateway (CSG) component of the ESB for this purpose.

For information on how to work with the Cloud Services Gateway (CSG) component of the ESB , see [Cloud Services Gateway \(CSG\)](#).

Monitoring the server

WSO2 ESB provides a variety of options to monitor and manage the server runtime through a number of monitoring tools as well as via Java Management Extensions (JMX) monitoring. Results provided by these convenient yet powerful ESB monitoring mechanisms can be used to tune message flows, detect mediation faults, and track usage patterns.

The following topics describe how to monitor the ESB:

- **Monitoring logs:** A properly configured logging system is vital for identifying errors, security threats and usage patterns in your product server. For instructions on monitoring the server logs, see [Monitoring Logs](#) in the WSO2 Administration Guide.
 - **Monitoring HTTP Access Logs:** HTTP Requests/Responses are logged in the access log(s) and are helpful to monitor your application's usage activities, such as the persons who access it, how many hits it receives, what the errors are etc. For more information on access logs, go to [HTTP Access Logging](#) in the WSO2 Administration Guide.
 - **Monitoring using WSO2 metrics:** WSO2 ESB is shipped with JVM Metrics, which allows you to monitor statistics of your server using Java Metrics. For instructions on setting up and using Carbon metrics for monitoring, see [Using WSO2 Metrics](#) in the WSO2 Administration Guide.
 - **JMX-based monitoring:** For information on monitoring your server using JMX, see [JMX-based monitoring](#) in the WSO2 Administration Guide. For information on the various MBeans available for monitoring, see [JMX Monitoring](#).
 - **SNMP monitoring:** Simple Network Management Protocol (SNMP) is an Internet-standard protocol for managing devices on IP networks. For information on how to configure SNMP in WSO2 ESB, see [SNMP Monitoring](#).
 - **Viewing handlers in message flows:** Message flows provide graphical and textual views of the globally engaged handlers of the system at any point of time. The [modules](#) use the handlers to engage in different message flows at defined phases. You can observe the handlers invoked in each phase of each flow in real time. For more information, see [Viewing the Handlers in Message Flows](#).
-

Applying patches

For information on applying patches (issued by WSO2), see [WSO2 Patch Application Process](#) in the WSO2 Administration Guide.

Working with Composite Applications (C-Apps) and artifacts

You can upload applications and artifacts to WSO2 ESB as follows:

- For information on the concept of 'Composite Applications' (C-Apps) and about how C-Apps can be deployed and managed, see [WSO2 ESB Tooling](#).
- For information on uploading artifacts to WSO2 ESB, see [Uploading Artifacts](#).

Upgrading from a Previous Release

This page walks you through the process of upgrading to ESB 5.0.0 from a previous ESB version. Go to the required tab for step by step instructions based on the upgrade you need to perform.

[Upgrading from a Previous ESB Version to ESB 5.0.0](#)[Upgrading from ESB 4.8.1 to ESB 5.0.0](#)

The followings steps describe how you can upgrade data and configurations to ESB 5.0.0 from a previous release (other than ESB 4.8.1). For more information on release versions, see the [Release Matrix](#).

You cannot roll back the upgrade process. However, it is possible to restore a backup of the previous database so that you can restart the upgrade progress.

Preparing to upgrade

The following prerequisites must be completed before upgrading:

- Make a backup of the database and copy the <ESB_HOME> directory in order to backup the product configurations.
- Download ESB 5.0.0 from <http://wso2.com/products/enterprise-service-bus/>.

Note

The downtime is limited to the time taken for switching databases in the production environment.

Upgrading the database

Since there are no database changes between ESB 4.9.0 and ESB 5.0.0, you are only required to migrate the configurations and settings from the previous version as explained below.

Migrating the configuration files

Note

Configurations should not be copied directly between servers.

To connect ESB 5.0.0 to the upgraded database, configure the following files:

1. Go to the <ESB_HOME>/repository/conf/datasources directory and update the master-datasources.xml file . See [Configuring master-datasources.xml](#).
2. Go to the <ESB_HOME>/repository/conf directory and update the datasource references in the user-mgt.xml and registry.xml files to match the updated configurations in the master-datasources.xml file that you made in the above step. See [Configuring user-mgt.xml](#) and [Configuring registry.xml](#).
3. Check for any other configurations that were done for ESB 4.9.0 based on your solution and update the configuration files in ESB 5.0.0 accordingly. For example, configurations related to external user stores, caching, mounting, transports etc.

Note

The following files have changed from ESB 4.9.0 to ESB 5.0.0

- axis2.xml
- axis2_nhttp.xml
- axis2_pt.xml
- tenant-axis2.xml
- cache.xml
- config-validation.xml
- logging-bridge.properties
- osgi-debug.option
- cloud-services-desc.xml
- authenticators.xml
- ciper-tool.properties
- catalina-server.xml
- carbon.xml
- identity.xml
- nhttp.properties
- passthru-http.properties
- synapse.properties
- user-mgt.xml

4. If there are any third party libraries used with ESB 4.9.0 that you want to migrate, copy the following directories as applicable from ESB 4.9.0 to ESB 5.0.0:
 - If you have used JMS libraries, JDBC libraries etc, copy <ESB_HOME>/repository/components/1_ib
 - If you have used OSGi bundles such as SVN kit etc, copy <ESB_HOME>/repository/components/dropins
5. Start the ESB 5.0.0 server.

Migrating Artifacts

You should manually deploy Composite Application Archive (CAR) files that you have in ESB 4.9.0 to ESB 5.0.0. If you have a mediator packed in a CAR, all the artifacts using that mediator should also be included in the same CAR. See [Deploying a Carbon Application](#) for further details.

Note

To migrate deployment artifacts including ESB message flow configurations.

- Copy the required synapse artifacts from the <ESB_HOME>/repository/deployment/server directory of ESB 4.9.0 to ESB 5.0.0. If you do not have axis2 modules or axis2 services, you can copy the required synapse artifacts from the <ESB_HOME>/repository/deployment/server/synapse-configs/default directory of ESB 4.9.0 to ESB 5.0.0.
- If multi-tenancy is used, copy the <ESB_HOME>/repository/tenants directory from ESB 4.9.0 to ESB 5.0.0.

Testing the upgrade

Verify that all the required scenarios are working as expected in ESB 5.0.0. This confirms that the upgrade is successful.

WORK IN PROGRESS

The followings steps describe how you can upgrade data and configurations to upgrade from ESB 4.8.1 to ESB 5.0.0. For more information on release versions, see the [Release Matrix](#).

You cannot roll back the upgrade process. However, it is possible to restore a backup of the previous database so that you can restart the upgrade progress.

Preparing to upgrade

The following prerequisites must be completed before upgrading:

- Make a backup of the database and copy the <ESB_HOME> directory in order to backup the product configurations.
- Download ESB 5.0.0 from <http://wso2.com/products/enterprise-service-bus/>.

Note

The downtime is limited to the time taken for switching databases in the production environment.

Upgrading the database

The instructions in this section describe how you can perform a data migration to upgrade the 4.8.1 database for use in ESB 5.0.0.

1. Before you upgrade to ESB 5.0.0, create a **new** database and restore the backup of the ESB 4.8.1 database in this new database.

You should NOT connect a new version of WSO2 ESB to an older database that has not been upgraded.

2. Select the `mysql.sql` script for the upgrade from [here](#) and run it on the new database. Running this script will ensure that the new database is upgraded to have the additional tables and schemas that are required for ESB 5.0.0.

Once you run the migration scripts on the new database, it becomes the upgraded database for ESB 5.0.0.

Migrating the configuration files

Note

Configurations should not be copied directly between servers.

To connect ESB 5.0.0 to the upgraded database, configure the following files:

1. Go to the <ESB_HOME>/repository/conf/datasources directory and update the master-datasources.xml file . See [Configuring master-datasources.xml](#).
2. Go to the <ESB_HOME>/repository/conf directory and update the datasource references in the user-mgt.xml and registry.xml files to match the updated configurations in the master-datasources.xml file that you made in the above step. See [Configuring user-mgt.xml](#) and [Configuring registry.xml](#).
3. Check for any other configurations that were done for ESB 4.8.1 based on your solution and update the configuration files in ESB 5.0.0 accordingly. For example, configurations related to external user stores, caching, mounting, transports etc.

Note

The following files have changed from ESB 4.8.1 to ESB 5.0.0

- axis2.xml
- axis2_nhttp.xml
- axis2_pt.xml
- tenant-axis2.xml
- cache.xml
- config-validation.xml
- logging-bridge.properties
- osgi-debug.option
- cloud-services-desc.xml
- authenticators.xml
- ciper-tool.properties
- catalina-server.xml
- carbon.xml
- identity.xml
- nhttp.properties
- passthru-http.properties
- synapse.properties
- user-mgt.xml

4. If there are any third party libraries used with ESB 4.8.1 that you want to migrate, copy the following directories as applicable from ESB 4.8.1 to ESB 5.0.0:
 - If you have used JMS libraries, JDBC libraries etc, copy <ESB_HOME>/repository/components/lib
 - If you have used OSGi bundles such as SVN kit etc, copy <ESB_HOME>/repository/components/dropins
5. Start the ESB 5.0.0 server.

Migrating Artifacts

You should manually deploy Composite Application Archive (CAR) files that you have in ESB 4.8.1 to ESB 5.0.0. If you have a mediator packed in a CAR, all the artifacts using that mediator should also be included in the same CAR. See [Deploying a Carbon Application](#) for further details.

Note

To migrate deployment artifacts including ESB message flow configurations.

- Copy the required synapse artifacts from the <ESB_HOME>/repository/deployment/server directory of ESB 4.8.1 to ESB 5.0.0. If you do not have axis2 modules or axis2 services, you can copy the required synapse artifacts from the <ESB_HOME>/repository/deployment/server/synapse-configs/default directory of ESB 4.8.1 to ESB 5.0.0.
- If multi-tenancy is used, copy the <ESB_HOME>/repository/tenants directory from ESB 4.8.1 to ESB 5.0.0.

Prior to copying the above folders remove all secured services from the folder.

With the removal of QoS features from ESB management console, enabling security for services hosted in ESB has changed from ESB 4.9.0 onwards. You now need to secure your services using [ESB Tooling](#) before you can migrate them to ESB 5.0.0.

See, [Applying Security to a Proxy Service](#) for instructions on how to create secured services and deploy them in ESB.

Testing the upgrade

Verify that all the required scenarios are working as expected in ESB 5.0.0. This confirms that the upgrade is successful.

Clustered Deployment

This section describes how to set up a WSO2 ESB worker/manager separated [cluster](#) and how to configure this cluster with a third-party load balancer. The following sections give you information and instructions on how to set up your cluster.

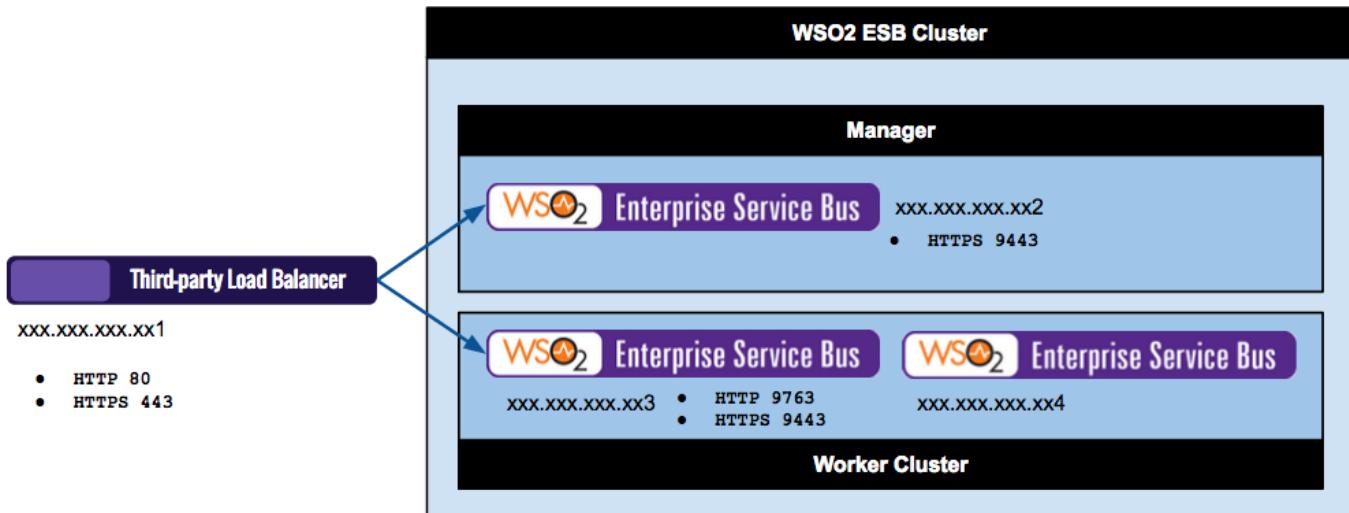
- Worker/manager separated clustering deployment pattern
- Configuring the load balancer
- Setting up the databases
- Configuring the manager node
- Configuring the worker node
- Setting up Scheduled Tasks
- Testing the cluster

Important: When configuring your WSO2 products for clustering, it is necessary to use a specific IP address and not localhost or host names in your configurations. So, keep this in mind when hosting WSO2 products in your production environment.

Worker/manager separated clustering deployment pattern

In this pattern there are three WSO2 ESB nodes; 1 node acts as the manager node and 2 nodes act as worker nodes for high availability and serving service requests. In this pattern, we allow access to the admin console through an external load balancer. Additionally, service requests are directed to worker nodes through this load

balancer. The following image depicts the sample pattern this clustering deployment scenario will follow.



Here, we use two nodes as well-known members, one is the manager node and the other is one of the worker nodes. It is always recommended to use at least two well-known members to prevent restarting all the nodes in the cluster in case a well-known member is shut down.

See [Worker/Manager separated clustering patterns](#) for a wider variety of options if you prefer to use a different clustering deployment pattern.

Configuring the load balancer

The load balancer automatically distributes incoming traffic across multiple WSO2 product instances. It enables you to achieve greater levels of fault tolerance in your cluster and provides the required balancing of load needed to distribute traffic.

About clustering without a load balancer

The configurations in this subsection are not required if your clustering setup does not have a load balancer. If you follow the rest of the configurations in this topic while excluding this section, you will be able to set up your cluster without the load balancer.

Things to keep in mind

The configuration steps in this document are written assuming that default 80 and 443 ports are used and exposed by the 3rd party load balancer for this ESB cluster. If any other ports are used instead of the default ones, please replace 80 and 443 values with the corresponding ports in the relevant places.

So with the above in mind, please note the following:

- Load balancer ports are HTTP 80 and HTTPS 443 as indicated in the deployment pattern above.
- Direct the HTTP requests to the worker nodes using <http://xxx.xxx.xxx.xx3/<service>> via HTTP 80 port.
- Direct the HTTPS requests to the worker nodes using <https://xxx.xxx.xxx.xx3/<service>> via HTTPS 443 port.
- Access the management console as <https://xxx.xxx.xxx.xx2/carbon> via HTTPS 443 port
- In a WSO2 ESB cluster, the worker nodes address service requests on the [PassThrough Transport ports](#) (8280 and 8243) and can access the Management Console using the HTTPS 9443 port.

Tip: We recommend that you use NGINX Plus as your load balancer of choice.

Use the following steps to configure [NGINX Plus](#) version 1.7.11 as the load balancer for WSO2 products.

1. Install NGINX Plus in a server configured in your cluster.
2. Configure NGINX Plus to direct the HTTP requests to the two worker nodes via the HTTP 80 port using the `http://esb.wso2.com/<service>`. To do this, create a VHost file (`esb.http.conf`) in the `/etc/nginx/conf.d` directory and add the following configurations into it.

```
upstream wso2.esb.com {
    server xxx.xxx.xxx.xx3:8280;
    server xxx.xxx.xxx.xx4:8280;
}

server {
    listen 80;
    server_name esb.wso2.com;
    location / {
        proxy_set_header X-Forwarded-Host $host;
        proxy_set_header X-Forwarded-Server $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_read_timeout 5m;
        proxy_send_timeout 5m;
        proxy_pass http://wso2.esb.com;
    }
}
```

3. Configure NGINX Plus to direct the HTTPS requests to the two worker nodes via the HTTPS 443 port using `https://esb.wso2.com/<service>`. To do this, create a VHost file (`esb.https.conf`) in the `/etc/nginx/conf.d` directory and add the following configurations into it.

```

upstream ssl.wso2.esb.com {
    server xxx.xxx.xxx.xx3:8243;
    server xxx.xxx.xxx.xx4:8243;

    sticky learn create=$upstream_cookie_jsessionid
    lookup=$cookie_jsessionid
    zone=client_sessions:1m;
}

server {
listen 443;
server_name esb.wso2.com;
ssl on;
ssl_certificate /etc/nginx/ssl/wrk.crt;
ssl_certificate_key /etc/nginx/ssl/wrk.key;
location / {
    proxy_set_header X-Forwarded-Host $host;
    proxy_set_header X-Forwarded-Server $host;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_read_timeout 5m;
    proxy_send_timeout 5m;
    proxy_pass https://ssl.wso2.esb.com;
}
}

```

- Configure NGINX Plus to access the Management Console as `https://mgt.esb.wso2.com/carbon` via HTTPS 443 port. This is to direct requests to the manager node. To do this, create a VHost file (`mgt.esb.https.conf`) in the `/etc/nginx/conf.d` directory and add the following configurations into it.

```

server {
listen 443;
server_name mgt.esb.wso2.com;
ssl on;
ssl_certificate /etc/nginx/ssl/mgt.crt;
ssl_certificate_key /etc/nginx/ssl/mgt.key;

location / {
    proxy_set_header X-Forwarded-Host $host;
    proxy_set_header X-Forwarded-Server $host;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_read_timeout 5m;
    proxy_send_timeout 5m;
    proxy_pass https://xxx.xxx.xxx.xx2:9443/;
}
error_log /var/log/nginx/mgt-error.log ;
access_log /var/log/nginx/mgt-access.log;
}

```

- Restart the NGINX Plus server.

`$sudo service nginx restart`

Tip: You do not need to restart the server if you are simply making a modification to the VHost file. The following command should be sufficient in such cases.

```
$sudo service nginx reload
```

Create SSL certificates

Create SSL certificates for both the manager and worker nodes using the instructions that follow.

1. Create the Server Key.

```
$sudo openssl genrsa -des3 -out server.key 1024
```

2. Certificate Signing Request.

```
$sudo openssl req -new -key server.key -out server.csr
```

3. Remove the password.

```
$sudo cp server.key server.key.org
```

```
$sudo openssl rsa -in server.key.org -out server.key
```

4. Sign your SSL Certificate.

```
$sudo openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

While creating keys, enter the host name (`esb.wso2.com` or `mgt.esb.wso2.com`) as the common name.

Setting up the databases

See [Setting up the Database](#) for information on how to set up the databases for a cluster. The datasource configurations must be done in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file for both the manager and worker nodes. You would also have to configure the shared registry database and mounting details in the `<PRODUCT_HOME>/repository/conf/registry.xml` file.

Configuring the manager node

1. Download and unzip the WSO2 ESB binary distribution. Consider the extracted directory as `<PRODUCT_HOME>`.

2. Set up the cluster configurations. Edit the `<PRODUCT_HOME>/repository/conf/axis2/axis2.xml` file as follows.

- a. Enable clustering for this node:

```
<clustering
    class="org.wso2.carbon.core.clustering.hazelcast.HazelcastClusteringAgent"
    enable="true">
```

- b. Set the membership scheme to wka to enable the well-known address registration method (this node sends cluster initiation messages to the WKA members that we define later).

```
<parameter name="membershipScheme">wka</parameter>
```

- c. Specify the name of the cluster this node will join.

```
<parameter name="domain">wso2.esb.domain</parameter>
```

- d. Specify the host used to communicate cluster messages.

```
<parameter name="localMemberHost">xxx.xxx.xxx.xx2</parameter>
```

- e. Specify the port used to communicate cluster messages. This port number is not affected by the port offset value specified in the `<PRODUCT_HOME>/repository/conf/carbon.xml`. If this port number is already assigned to another server, the clustering framework automatically increments this port number. However, if two servers are running on the same machine, you must ensure that a unique port is set for each server.

```
<parameter name="localMemberPort">4100</parameter>
```

- f. Specify the well-known members. In this example, the well-known member is a worker node. The port value for the WKA worker node must be the same value as its `localMemberPort` (in this case 4200).

```
<members>
<member>
<hostName>xxx.xxx.xxx.xx3</hostName>
<port>4200</port>
</member>
</members>
```

Although this example only indicates one well-known member, it is recommended to add at least two well-known members here. This is done to ensure that there is high availability for the cluster.

You can also use IP address ranges for the hostName. For example, 192.168.1.2-10. This should ensure that the cluster eventually recovers after failures. One shortcoming of doing this is that you can define a range only for the last portion of the IP address. You should also keep in mind that the smaller the range, the faster the time it takes to discover members since each node has to scan a lesser number of potential members.

g. Change the following clustering properties.

```
<parameter name="properties">
<property name="backendServerURL"
value="https://${hostName}:${httpsPort}/services/" />
<property name="mgtConsoleURL"
value="https://${hostName}:${httpsPort}/" />
</parameter>
```

- Configure the HostName. To do this, edit the <PRODUCT_HOME>/repository/conf/carbon.xml file as follows.

```
<HostName>esb.wso2.com</HostName>
<MgtHostName>mgt.esb.wso2.com</MgtHostName>
```

- Enable SVN-based deployment synchronization with the AutoCommit property marked as true. To do this, edit the <PRODUCT_HOME>/repository/conf/carbon.xml file as follows. See [Configuring Deployment Synchronizer](#) for more information on this.

```
<DeploymentSynchronizer>
<Enabled>true</Enabled>
<AutoCommit>true</AutoCommit>
<AutoCheckout>true</AutoCheckout>
<RepositoryType>svn</RepositoryType>
<SvnUrl>https://svn.wso2.org/repos/esb</SvnUrl>
<SvnUser>svnuser</SvnUser>
<SvnPassword>xxxxxxxx</SvnPassword>
<SvnUrlAppendTenantId>true</SvnUrlAppendTenantId>
</DeploymentSynchronizer>
```

- In the <PRODUCT_HOME>/repository/conf/carbon.xml file, you can also specify the port offset value. This is **ONLY** applicable if you have multiple WSO2 products hosted on the same server.
 - [Click here for more information on configuring the port offset.](#)

When you run multiple products/clusters or multiple instances of the same product on the same server or virtual machines (VMs), you must change their default ports with an offset value to avoid port conflicts. An offset defines the number by which all ports in the runtime (e.g., HTTP(S) ports) are increased. For example, if the default HTTP port is 9763 and the offset is 1, the effective HTTP port will change to 9764. For each additional product instance, you set the port offset to a unique value. The offset of the default ports is considered to be 0.

The port value will automatically be increased as shown in the **Port Value** column in the following table, allowing all five WSO2 product instances or servers to run on the same machine.

WSO2 product instance	Port Offset	Port Value
WSO2 server 1	0	9443
WSO2 server 2	1	9444
WSO2 server 3	2	9445
WSO2 server 4	3	9446
WSO2 server 5	4	9447

Important: This step is optional and only required if all server instances are running on the same machine. This is not a recommended approach for production environments. Note that you need to change all ports used in your configurations based on the offset value if you are to do this.

```
<Ports>
  ...
  <Offset>0</Offset>
  ...
</Ports>
```

- Map the host names to the IP. Add the below host entries to your DNS, or “/etc/hosts” file (in Linux) in all the nodes of the cluster. You can map the IP address of the database server. In this example, MySQL is used as the database server, so <MySQL-DB-SERVER-IP> is the actual IP address of the database server.

```
<IP-of-MYSQL-DB-SERVER> carbondb.mysql-wso2.com
```

- Allow access the management console only through the load balancer. Configure the HTTP/HTTPS proxy ports to communicate through the load balancer by editing the <PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml file as follows.

```
<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"
  port="9763"
  proxyPort="80"
  ...
/>
<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"
  port="9443"
  proxyPort="443"
  ...
/>
```

▼ [Click here for more information on this configuration.](#)

The `Connector protocol` tag sets the protocol to handle incoming traffic. The default value is `HTTP/1.1`, which uses an auto-switching mechanism to select either a blocking Java-based connector or an APR/native connector. If the `PATH` (Windows) or `LD_LIBRARY_PATH` (on most UNIX systems) environment variables contain the Tomcat native library, the APR/native connector will be used. If the native library cannot be found, the blocking Java-based connector will be used. Note that the APR/native connector has different settings from the Java connectors for HTTPS.

The non-blocking Java connector used is an explicit protocol that does not rely on the auto-switching mechanism described above. The following is the value used:

```
org.apache.coyote.http11.Http11NioProtocol
```

The TCP port number is the value that this `Connector` will use to create a server socket and await incoming connections. Your operating system will allow only one server application to listen to a particular port number on a particular IP address. If the special value of 0 (zero) is used, Tomcat will select a free port at random to use for this connector. This is typically only useful in embedded and testing applications.

Configuring the worker node

1. Download and unzip the WSO2 ESB binary distribution. Consider the extracted directory as `<PRODUCT_HOME>`.
2. Set up the cluster configurations. Edit the `<PRODUCT_HOME>/repository/conf/axis2/axis2.xml` file as follows.
 - a. Enable clustering for this node.


```
<clustering
class="org.wso2.carbon.core.clustering.hazelcast.HazelcastClusteringAgent"
enable="true">
```
 - b. Set the membership scheme to `wka` to enable the well-known address registration method (this node will send cluster initiation messages to WKA members that we will define later).


```
<parameter name="membershipScheme">wka</parameter>
```
 - c. Specify the name of the cluster this node will join.


```
<parameter name="domain">wso2.esb.domain</parameter>
```
 - d. Specify the host used to communicate cluster messages.


```
<parameter name="localMemberHost">xxx.xxx.xxx.xx3</parameter>
```
 - e. Specify the port used to communicate cluster messages. If this node is on the same server as the manager node, or another worker node, set this to a unique value, such as 4000 and 4001 for worker nodes 1 and 2. This port number will not be affected by the port offset in `carbon.xml`. If this port number is already assigned to another server, the clustering framework will automatically increment this port number.


```
<parameter name="localMemberPort">4200</parameter>
```
 - f. Define the sub-domain as worker by adding the following property inside the `<parameter name="properties">` element:


```
<property name="subDomain" value="worker" />
```
 - g. Specify the well-known member by providing the hostname and `localMemberPort` values. Here, the well-known member is the manager node. Defining the manager node is useful since it is required for the [Deployment Synchronizer](#) to function in an efficient manner. The deployment synchronizer uses this configuration to identify the manager and synchronize deployment artifacts across the nodes of a cluster.

```
<members>
<member>
<hostName>xxx.xxx.xxx.xx2</hostName>
<port>4100</port>
</member>
</members>
```

Although this example only indicates one well-known member, it is recommended to add at least two well-known members here. This is done to ensure that there is high availability for the cluster.

You can also use IP address ranges for the hostName. For example, 192.168.1.2-10. This should ensure that the cluster eventually recovers after failures. One shortcoming of doing this is that you can define a range only for the last portion of the IP address. You should also keep in mind that the smaller the range, the faster the time it takes to discover members since each node has to scan a lesser number of potential members.

- h. Uncomment and edit the `WSDLEPRPrefix` element under `org.apache.synapse.transport.passthru.PassThroughHttpListener` and `org.apache.synapse.transport.passthru.PassThroughHttpSSLListener` in the `transportReceiver`.

```
<parameter name="WSDLEPRPrefix"
locked="false">http://esb.wso2.com:80</parameter>

<parameter name="WSDLEPRPrefix"
locked="false">https://esb.wso2.com:443</parameter>
```

3. Configure the `<PRODUCT_HOME>/repository/conf/carbon.xml` file to match your clustering configurations with your product.
 - a. Configure the HostName. To do this, edit the `<PRODUCT_HOME>/repository/conf/carbon.xml` file as follows.

```
<HostName>esb.wso2.com</HostName>
```

- b. Enable SVN-based deployment synchronization with the `AutoCommit` property marked as `false`. To do this, edit the `<PRODUCT_HOME>/repository/conf/carbon.xml` file as follows.

```
<DeploymentSynchronizer>
<Enabled>true</Enabled>
<AutoCommit>false</AutoCommit>
<AutoCheckout>true</AutoCheckout>
<RepositoryType>svn</RepositoryType>
<SvnUrl>https://svn.wso2.org/repos/esb</SvnUrl>
<SvnUser>svnuser</SvnUser>
<SvnPassword>xxxxxxxx</SvnPassword>
<SvnUrlAppendTenantId>true</SvnUrlAppendTenantId>
</DeploymentSynchronizer>
```

- c. In the `<PRODUCT_HOME>/repository/conf/carbon.xml` file, you can also specify the port offset

value. This is **ONLY** applicable if you have multiple WSO2 products hosted on the same server.

▼ [Click here for more information on configuring the port offset.](#)

When you run multiple products/clusters or multiple instances of the same product on the same server or virtual machines (VMs), you must change their default ports with an offset value to avoid port conflicts. An offset defines the number by which all ports in the runtime (e.g., HTTP/S ports) are increased. For example, if the default HTTP port is 9763 and the offset is 1, the effective HTTP port will change to 9764. For each additional product instance, you set the port offset to a unique value. The offset of the default ports is considered to be 0.

The port value will automatically be increased as shown in the **Port Value** column in the following table, allowing all five WSO2 product instances or servers to run on the same machine.

WSO2 product instance	Port Offset	Port Value
WSO2 server 1	0	9443
WSO2 server 2	1	9444
WSO2 server 3	2	9445
WSO2 server 4	3	9446
WSO2 server 5	4	9447

Important: This step is optional and only required if all server instances are running in the same machine. This is not a recommended approach for production environments. Note that you need to change all ports used in your configurations based on the offset value if you are to do this.

```
<Ports>
...
<Offset>0</Offset>
...
</Ports>
```

- Map the host names to the IP. Add the following host entries to your DNS, or “/etc/hosts” file (in Linux) in all the nodes of the cluster. You can map the IP address of the database server. In this example, MySQL is used as the database server, so <MySQL-DB-SERVER-IP> is the actual IP address of the database server.

```
<IP-of-MYSQL-DB-SERVER> carbondb.mysql-wso2.com
```

- Create the second worker node by getting a copy of the WSO2 product you just configured as a worker node and change the following in the <PRODUCT_HOME>/repository/conf/axis2/axis2.xml file. This copy of the WSO2 product can be moved to a server of its own.

```
<parameter name="localMemberPort">4300</parameter>
```

Setting up Scheduled Tasks

This section is applicable if you have **Scheduled Tasks** configured in your ESB server nodes.

For instructions on how to configure task distribution in a clustered deployment, see [Configuring a clustered task server](#) in the Administration Guide.

When there is no worker manager separation in your clustering configuration, to ensure that tasks are

scheduled in all ESB nodes, you need to edit the `clusteringPattern` parameter in `<ESB_HOME>/repository/conf/axis2/axis2.xml` file as follows:

```
<parameter name="clusteringPattern">nonWorkerManager</parameter>
```

Testing the cluster

1. Restart the configured load balancer.
2. Start the manager node. The additional `-Dsetup` argument creates the required tables in the database.
`sh <PRODUCT_HOME>/bin/wso2server.sh -Dsetup`
3. Start the two worker nodes. The additional `-DworkerNode=true` argument indicates that this is a worker node. This parameter basically makes a server read-only. A node with this parameter will not be able to do any changes such as writing or making modifications to the deployment repository etc. This parameter also enables the [worker profile](#), where the UI bundles will not be activated and only the back end bundles will be activated once the server starts up. When you configure the `axis2.xml` file, the cluster sub-domain must indicate that this node belongs to the "worker" sub-domain in the cluster.

Tip: It is recommendation is to delete the `<PRODUCT_HOME>/repository/deployment/server` directory and create an empty server directory in the worker node. This is done to avoid any SVN conflicts that may arise.

4. Check for 'member joined' log messages in all consoles.

Additional information on logs and new nodes

When you terminate one node, all nodes identify that the node has left the cluster. The same applies when a new node joins the cluster.

If you want to add another new worker node, you can simply copy `worker1` without any changes if you are running it on a new server (such as `xxx.xxx.xxx.184`). If you intend to use the new node on a server where another WSO2 product is running, you can use a copy of `worker1` and change the port offset accordingly in the `carbon.xml` file. You also have to change `localMemberPort` in `axis2.xml` if that product has clustering enabled. Be sure to map all host names to the relevant IP addresses when creating a new node. The log messages indicate if the new node joins the cluster.

5. Access management console through the LB using the following URL: <https://xxx.xxx.xxx.xx2:443/carbon>
6. Test load distribution via <http://xxx.xxx.xxx.xx3:80/> or <https://xxx.xxx.xxx.xx3:443/>.
7. Add a sample proxy service with the log mediator in the `inSequence` so that logs will be displayed in the worker terminals, and then observe the cluster messages sent from the manager node to the workers.

The load balancer manages the active and passive states of the worker nodes, activating nodes as needed and leaving the rest in passive mode. To test this, send a request to the endpoint through the load balancer to verify that the proxy service is activated only on the active worker node(s) while the remaining worker nodes remain passive. For example, you would send the request to the following URL:

```
http://{{Load_Balancer_Mapped_URL_for_worker}}/services/{{Sample_Proxy_Name}}
```

Clustering with JMS

In a typical ESB cluster, you have several ESBs running in parallel with each having a JMS proxy listening to a single JMS queue. All the ESBs usually have the same back-end. With a setup like this, you have to implement fault tolerance and concurrent processing of JMS messages. Fault tolerance ensures that even when one or more ESBs go down, the system will still keep running until there is at least one ESB in the cluster.

In this clustered setup, Apache ActiveMQ creates a queue by the name of the proxy service. When two ESBs with

the same proxy configuration are in the system, ActiveMQ will mark both of them as JMS consumers of a single JMS queue. When the JMS queue gets JMS messages, it will dispatch them to the JMS consumers (ESBs) in a round robin manner.

The ESB configuration of this cluster setup is as follows.

```
<proxy xmlns="http://ws.apache.org/ns/synapse" name="StockQuoteProxy" transports="jms"
statistics="disable" trace="disable" startOnLoad="true">
    <target>
        <inSequence>
            <log level="full"/>
            <property name="OUT_ONLY" value="true"/>
        </inSequence>
        <outSequence>
            <send/>
        </outSequence>
        <endpoint>
            <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
        </endpoint>
    </target>
    <publishWSDL uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
    <parameter name="transport.jms.ContentType">
        <rules>
            <jmsProperty>contentType</jmsProperty>

            <default>application/xml</default>
        </rules>
    </parameter>
    <description></description>
</proxy>
```

Follow the steps below to configure the cluster and invoke the message flow.

1. Configure two ESB instances to connect with ActiveMQ by copying the relevant jars to <ESB_HOME>/repository/components/lib directory. Both instances should have the same JMS proxy configuration.
2. Increase the port offset value of one ESB instance by editing <ESB_HOME>/repository/conf/carbon.xml file as follows. This is done to ensure that there are no port conflicts by running two WSO2 Carbon instances simultaneously in the same environment.

```
<!-- Ports offset. This entry will set the value of the ports defined below to the
define value + Offset. e.g. Offset=2 and HTTPS port=9443 will set the effective HTTPS
port to 9445 -->

<Offset>1</Offset>
```

3. Set up the prerequisites given in the **Prerequisites** section in **Setting Up the ESB Samples**. Then, deploy the **SimpleStockQuoteService** client by navigating to <ESB_HOME>/samples/axis2Server/src/SimpleStockQuoteService, and running the **ant** command in the command prompt or shell script. This will build the sample and deploy the service for you.
4. WSO2 ESB comes with a default Axis2 server, which you can use as the back-end service for this sample. To start the Axis2 server, navigate to <ESB_HOME>/samples/axis2server and run axis2Server.sh (on Linux) or axis2Server.bat (on Windows).
5. Start ActiveMQ and the two ESB instances.

6. Run the JMS client in axis2client using the following command.

```
ant jmsclient -Djms_type=pox -Djms_dest=dynamicQueues/StockQuoteProxy  
-Djms_payload=MSFT
```

The message flow of the cluster is as follow.

The Message Flow

- The JMS client sends a JMS message to the message queue named **StockQuoteProxy** with information for a stock order.
- When the message goes to the JMS queue, ActiveMQ dispatches the message to one of the JMS proxy services in an ESB. This is done in a round robin manner.
- The JMS proxy then sends the message to the back-end.

Performance Tuning

The out of the box high performance of WSO2 ESB can be further optimised by following a set of recommendations from WSO2 experts. This section highlights general performance tips, transport level tuning guidelines as well as tuning best practices you can follow for specific use cases.

See the following topics for detailed information on how you can tune the ESB for optimal performance:

Important

- Performance tuning requires you to modify important system files, which affect all programs running on the server. We recommend you to familiarize yourself with these files using Unix/Linux documentation before editing them.
- The parameter values we discuss below are just examples. They might not be the optimal values for the specific hardware configurations in your environment. We recommend you to carry out load tests on your environment to tune the ESB accordingly.

- Network and OS Level Performance Tuning
- Java Virtual Machine (JVM) Level Tuning
- WSO2 Carbon Platform-Level Tuning
- Tuning the HTTP Transport
- Tuning the JMS Transport
- Tuning the VFS Transport
- Tuning the RabbitMQ Transport
- Tuning Inbound Endpoints
- Tuning the Performance based on Use Case
 - Split-Aggregate Pattern
 - Message Transformations
 - JMS Scenarios
- Tuning Analytics

For an example that illustrates how to tune the performance of the ESB, see [Performance Tuning WSO2 ESB with a practical example](#).

Network and OS Level Performance Tuning

When it comes to performance, the OS that the server runs plays an important role. There are several parameters that you can configure to optimize the network and OS performance.

- Configure the following parameters in the `/etc/sysctl.conf` file of Linux for maximum concurrency. These parameters can be set to specify a larger port range, a more effective TCP connection timeout value, and a number of other important settings at the OS-level based on your requirement.

Note

Since all these settings apply at the OS level, changing settings can affect other programs running on the server. The sample values specified here might not be the optimal values for your production system. You need to apply the values and run a performance test to find the best values for your production system.

Parameter	Description	Recommended Value
net.ipv4.tcp_fin_timeout	This is the length of time (in seconds) that TCP takes to receive a final FIN before the socket is closed. Setting this is required to prevent DoS attacks.	30
net.ipv4.tcp_tw_recycle	This enables fast recycling of TIME_WAIT sockets.	1
	<p>Note</p> <p>Change this with caution and ONLY in internal networks where the network connectivity speeds are faster. It is not recommended to use net.ipv4.tcp_tw_recycle = 1 when working with network address translation (NAT), such as if you are deploying products in EC2 or any other environment configured with NAT.</p>	
net.ipv4.tcp_tw_reuse	This allows reuse of sockets in TIME_WAIT state for new connections when it is safe from the network stack's perspective.	1
net.core.rmem_default	This sets the default OS receive buffer size for all types of connections.	524288
net.core.wmem_default	This sets the default OS send buffer size for all types of connections.	524288
net.core.rmem_max	This sets the maximum OS receive buffer size for all types of connections.	67108864
net.core.wmem_max	This sets the maximum OS send buffer size for all types of connections.	67108864

net.ipv4.tcp_rmem	<p>This specifies the receive buffer space for each TCP connection and has three values that hold the following information:</p> <p>The first value is the minimum receive buffer space for each TCP connection, and this buffer is always allocated to a TCP socket, even under high pressure on the system.</p> <p>The second value is the default receive buffer space allocated for each TCP socket. This value overrides the <code>/proc/sys/net/core/rmem_default</code> value used by other protocols.</p> <p>The last value is the maximum receive buffer space allocated for a TCP socket.</p>	4096 87380 16777216
net.ipv4.tcp_wmem	<p>This specifies the send buffer space for each TCP connection and has three values that hold the following information:</p> <p>The first value is the minimum TCP send buffer space available for a single TCP socket.</p> <p>The second value is the default send buffer space allowed for a single TCP socket to use.</p> <p>The third value is the maximum TCP send buffer space.</p> <p>Every TCP socket has the specified amount of buffer space to use before the buffer is filled up, and each of the three values are used under different conditions.</p>	4096 65536 16777216
net.ipv4.ip_local_port_range	<p>This defines the local port range that is used by TCP and UDP to choose the local port. The first number is the first local port allowed for TCP and UDP traffic, and the second number is the last port number.</p> <p>If your Linux server is opening a large number of outgoing network connections, you need to increase the default local port range. In Linux, the default range of IP port numbers allowed for TCP and UDP traffic is small, and if this range is not changed accordingly, a server can come under fire if it runs out of ports.</p>	1024 65535
fs.file-max	<p>This is the maximum number of file handles that the kernel can allocate. The kernel has a built-in limit on the number of files that a process can open. If you need to increase this limit, you can increase the <code>fs.file-max</code> value although it can take up some system memory.</p>	2097152

- Configure the following parameters in the `/etc/security/limits.conf` file of Linux if you need to alter the maximum number of open files allowed for system users.

```
* soft nofile 4096
* hard nofile 65535
```

Note

The * character denotes that the limit is applicable to all system users in the server, and the values specified above are the default values for normal system usage.

The hard limit is used to enforce hard resource limits and the soft limit is to enforce soft resource limits. The hard limit is set by the super user and is enforced by the Kernel. You cannot increase the hard limit unless you have super user privileges. You can increase or decrease the soft limit as necessary, but the maximum limit you can increase this is up to the hard limit value that is set.

- Configure the following settings in the `/etc/security/limits.conf` file of Linux if you need to alter the maximum number of processes a system user is allowed to run at a given time. Each carbon server instance you run requires upto 1024 threads with the default thread pool configuration. Therefore, you need to increase both the hard and soft nproc value by 1024 per carbon server.

```
* soft nproc 20000
* hard nproc 20000
```

Note

The * character denotes that the limit is applicable to all system users in the server, and the values specified above are the default values for normal system usage.

Java Virtual Machine (JVM) Level Tuning

You can tune the Java Virtual Machine (JVM) settings to make a production system more efficient.

You can configure the JVM parameters in the `<ESB_HOME>/bin/wso2server.bat` file (on Windows) or the `<ESB_HOME>/bin/wso2server.sh` file (on Linux/Solaris). Following are the most important JVM parameters you need to configure:

- Maximum Heap Memory Allocation (Xmx)** - This parameter sets the maximum heap memory allocated for the JVM. Increasing the memory allocation increases the memory available for the ESB, which results in increasing the maximum TPS and reducing the latency. We recommend al least 2 GB of heap memory allocation for instances.

For example, If you want to set 2 GB as the maximum heap memory size, you need to configure the parameter as follows:

```
-Xms2048m -Xmx2048m -XX:MaxPermSize=1024m
```

Here,

Xmx is the maximum memory allocation pool for a JVM.

Xms is the initial memory allocation pool.

XX:MaxPermSize is the permanent space where the classes, methods, internalized strings, and similar objects used by the VM are stored and never deallocated.

- Entity Expansion** - If one or more worker nodes in a clustered deployment require access to the management console, you need to increase the entity expansion limit. The default entity expansion limit is 64000.

If you want to set expansion limit is 100000, you need to configure the parameter as follows:

```
-DentityExpansionLimit=100000
```

WSO2 Carbon Platform-Level Tuning

WSO2 ESB runs on top of WSO2 Carbon platform. Therefore, there are some performance tuning parameters that can be set at the Carbon platform level.

In multi-tenant mode, the WSO2 Carbon runtime limits the thread execution time. That is, if a thread is stuck or taking a long time to process, Carbon detects such threads, interrupts and stops them. Carbon prints the current stack trace before interrupting the thread. This mechanism is implemented as an Apache Tomcat valve. Therefore, it should be configured in the <PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml file as shown below.

```
<Valve className="org.wso2.carbon.tomcat.ext.valves.CarbonStuckThreadDetectionValve"
threshold="600" />
```

- The `className` is the Java class used for the implementation. Set it to `org.wso2.carbon.tomcat.ext.valves.CarbonStuckThreadDetectionValve`.
- The `threshold` is the minimum duration in seconds after which a thread is considered stuck. The default value is 600 seconds.

Tuning the HTTP Transport

WSO2 ESB's HTTP transport can be used to handle blocking and non-blocking calls. This section describes how you can tune the HTTP transport of WSO2 ESB for better performance.

Improving the non-blocking invocation performance

WSO2 ESB supports two non-blocking transports, namely the passthrough transport and the nhttp transport. The passthrough transport is the default transport of the ESB, but you can set the NHTTP transport as the default transport by renaming the <ESB_Server>/repository/conf/axis2/axis2_nhttp.xml file to axis2.xml.

You can improve the non-blocking invocation performance by either configuring properties related to the HTTP Pass Through transport or configuring properties related to the NHTTP transport, based on which transport you are using as the default ESB transport in your production environment.

Configuring passthru-http.properties

You can configure the following properties as required in the <ESB_Home>/repository/conf/passthru-http.properties file:

Property	Description	Default Value
<code>worker_pool_size_core</code>	WSO2 ESB uses a thread pool executor to create threads and handle incoming requests. This parameter controls the number of core threads used by the executor pool. If you increase this parameter value, the number of requests received that can be processed by the ESB increases, hence, the throughput also increases.	400

worker_pool_size_max	This is the maximum number of threads in the worker thread pool. Specifying a maximum limit avoids performance degradation that can occur due to context switching. If the specified value is reached, you will see the error "SYSTEM ALERT - HttpServerWorker threads were in BLOCKED state during last minute". This can occur due to an extraordinarily high number of requests sent at a time when all the threads in the pool are busy, and the maximum number of threads is already reached.	500
http.socket.timeout	This is the maximum period of inactivity between two consecutive data packets, specified in milliseconds.	120000
worker_thread_keepalive_sec	This defines the keep-alive time for extra threads in the worker pool. The value specified here should be less than the socket timeout value. Once this time has elapsed for an extra thread, it will be destroyed. The purpose of this parameter is to optimize the usage of resources by avoiding wastage that results by having unutilized extra threads.	60
worker_pool_queue_length	This defines the length of the queue that is used to hold runnable tasks to be executed by the worker pool. The thread pool starts queuing jobs when all the existing threads are busy, and the pool has reached the maximum number of threads. The value for this parameter should be -1 to use an unbound queue. If a bound queue is used and the queue gets filled to its capacity, any further attempts to submit jobs fail causing some messages to be dropped by Synapse.	-1
io_threads_per_reactor	This defines the number of IO dispatcher threads used per reactor. The value specified should not exceed the number of cores in the server.	The default value is equal to number of cores in the server.
io_buffer_size	This is the value of the memory buffer allocated when reading data into the memory from the underlying socket/file channels. You should leave this property set to the default value.	16384
http.max.connection.per.host.port	This defines the maximum number of connections allowed per host port.	32767

http.socket.reuseaddr	If this parameter is set to true, it is possible to open another socket on the same port as the socket that is currently used by the ESB server to listen to connections. This is useful when recovering from a crash. In such instances, if the socket is not properly closed, a new socket can be opened to listen to connections.	true
http.socket.buffer-size	This is used to configure the SessionInputBuffer size of http core. The SessionInputBuffer is used to fill data that is read from the OS socket. This parameter does not affect the OS socket buffer size.	8192
http.block_service_list	If this parameter is set to true, all services deployed to WSO2 ESB cannot be accessed via the <code>http(s) :<esb> : 8280 /services/</code> URL.	true
http.headers.preserve	This parameter allows you to specify the header field/s of messages passing through the ESB that need to be preserved and printed in the outgoing message (e.g., <code>http.headers.preserve = Location, Date, Server</code>). Supported header fields are: Location, Keep-Alive, Content-Length, Content-Type, Date, Server, User-Agent and Host.	
http.connection.disable.keepalive	If this parameter is set to true, the HTTP connections with the back end service are closed soon after the request is served. It is recommended to set this property to false so that the ESB does not have to create a new connection every time it sends a request to a back end service. However, you may need to close connections after they are used if the back end service does not provide sufficient support for keep alive connections.	false

Configuring nhttp.properties

You can configure the NHTTP properties as required in the `<ESB_Home>/repository/conf/nhttp.properties` file:

- Following are the properties used by the non-blocking HTTP transport:

Property	Description	Default Value
<code>http.socket.timeout.receiver</code>	This is the maximum period of inactivity between two consecutive data packets on the transport listener side. This is the socket timeout value for connection between the client and the ESB server.	60000
<code>http.socket.timeout.sender</code>	This is the maximum period of inactivity between two consecutive data packets for the transport sender side. This is the socket timeout value for connection between the ESB server and the backend server.	60000
<code>nhttp_buffer_size</code>	This is the size of the buffer through which data passes when receiving/transmitting NHTTP requests.	8192

<code>http.tcp.nodelay</code>	This determines whether Nagle's algorithm (for improving the efficiency of TCP/IP by reducing the number of packets sent over the network) is used. Use the value 0 to enable this algorithm and the value 1 to disable it. The algorithm should be enabled if you need to reduce bandwidth consumption.	1
<code>http.connection.stalecheck</code>	This determines whether stale connection check is used. Use the value 0 to enable stale connection check and the value 1 to disable it. When this parameter is enabled, connections that are no longer used are identified and disabled before each request execution. Stale connection check should be disabled when performing critical operations.	0
<code>http.block_service_list</code>	If this parameter is set to <code>true</code> , all services deployed to WSO2 ESB cannot be viewed via the <code>http(s) :<esb> :8280/services/</code> URL.	<code>false</code>
<code>http.headers.preserve</code>	This parameter allows you to specify the header field/s of messages passing through the ESB that need to be preserved and printed in the outgoing message (e.g., <code>http.headers.preserve = Location, Date, Server</code>). Supported header fields are: Date, Server and User-Agent.	

- Following are the HTTP sender thread pool properties:

Parameter Name	Description	Default Value
<code>snd_t_core</code>	Transport listener worker pool's initial thread count.	20
<code>snd_t_max</code>	Transport listener worker pool's maximum thread count. Once this limit is reached and all the threads in the pool are busy, they will be in a BLOCKED state. In such situations an increase in the number of messages would fire the error <code>SYSTEM ALERT - HttpServerWorker threads were in BLOCKED state during last minute.</code>	100
<code>snd_alive_sec</code>	Listener-side keep-alive seconds.	5
<code>snd_qlen</code>	The listener queue length.	-1
<code>snd_io_threads</code>	Listener-side IO workers.	2

When there is an increased load, it is recommended to increase the number of threads mentioned in the properties above to balance it.

- Following are the HTTP listener thread pool properties:

Note

Listener-side properties generally have the same values as the sender-side properties.

Property	Description	Default Value
lst_t_core	Transport sender worker pool's initial thread count.	20
lst_t_max	Transport sender worker pool's maximum thread count. Once this limit is reached and all the threads in the pool are busy, they will be in a BLOCKED state. In such situations an increase in the number of messages would fire the error SYSTEM ALERT - HttpServerWorker threads were in BLOCKED state during last minute .	100
lst_alive_sec	Sender-side keep-alive seconds.	5
lst_glen	The sender queue length.	-1
lst_io_threads	Sender-side IO workers.	2

- Following is a property for AIX based deployment:

Property	Description	Default Value
http.nio.interest-ops-queueing	Determines whether interestOps() queueing is enabled for the I/O reactors.	true

Improving the blocking invocation performance

The [Callout mediator](#) as well as the [Call mediator](#) in blocking mode uses the axis2 CommonsHTTPTransportSender internally to invoke services. It uses the MultiThreadedHttpConnectionManager to handle connections, but in default configuration it only allows two simultaneous connections per host. So if there are more than two requests per host, the requests have to wait until a connection is available. Therefore if the backend service is slow, many requests have to wait until a connection is available from the MultiThreadedHttpConnectionManager. This can lead to a significant degrade in the performance of the ESB.

In order to overcome this issue you can edit the CommonsHTTPTransportSender configuration in the <ESB_HOME>/repository/conf/axis2/axis2_blocking_client.xml file, and increase the value of the defaultMaxConnectionsPerHost parameter.

For example, if you need to set 100 simultaneous connections per second, you can set the value of the defaultMaxConnectionsPerHost parameter as follows:

```
<transportsender class="org.apache.axis2.transport.http.CommonsHTTPTransportSender"
name="http">
    <parameter name="PROTOCOL">HTTP/1.1</parameter>
    <parameter name="Transfer-Encoding">chunked</parameter>
    <parameter name="cacheHttpClient">true</parameter>
    <parameter name="defaultMaxConnectionsPerHost">100</parameter>
</transportsender>
```

Tuning the JMS Transport

WSO2 ESB's Java Message Service (JMS) transport allows you to easily send and receive messages to queues and topics of any JMS service that implements the JMS specification. The following sections describe how you can tune the JMS transport of WSO2 ESB for better performance.

- Increase the maximum number of JMS proxies

- Configuring JMS Listener
- Configuring JMS Sender
- Remove ClientApiNonBlocking when sending messages via JMS

Increase the maximum number of JMS proxies

If you create several JMS proxy services in WSO2 ESB, you will see a message similar to the following:

```
WARN - JMSListener Polling tasks on destination : JMSToHTTPStockQuoteProxy18 of type queue for service JMSToHTTPStockQuoteProxy18 have not yet started after 3 seconds ..
```

This issue occurs when you do not have enough threads available to consume messages from JMS queues. The maximum number of concurrent consumers (that is, the number of JMS proxies) that can be deployed is limited by the base transport worker pool that is used by the JMS transport. You can configure the size of this worker pool using the system properties `lst_t_core` and `lst_t_max`. Note that increasing these values will also increase the memory consumption, because the worker pool will allocate more resources.

Similarly, you can configure the current number and the anticipated number of outbound JMS proxies using the system properties `snd_t_core` and `snd_t_max`.

To adjust the values of these properties, you can modify the server startup script if you want to increase the available threads for all transports (requires more memory), or create a `jms.properties` file if you want to increase the available threads just for the JMS transport. Both approaches are described below.

To increase the threads for all transports

1. Open the `wso2server.sh` or `wso2server.bat` file in your `<ESB_HOME>/bin` directory for editing.
2. Change the values of the properties as follows:

- `-Dlst_t_core=200`
- `-Dlst_t_max=250`
- `-Dsnd_t_core=200`
- `-Dsnd_t_max=250`

To increase the threads for just the JMS transport

1. Create a file named `jms.properties` with the following properties:
 - `lst_t_core=200`
 - `lst_t_max=250`
 - `snd_t_core=200`
 - `snd_t_max=250`
2. Create a directory called `conf` under your `<ESB_HOME>` directory and save the file in this directory.

Configuring JMS Listener

You can increase the JMS listener performance by

1. [Using concurrent consumers](#)
2. [Enabling caching](#)

Using concurrent consumers

Concurrent consumers is the minimum number of threads for message consuming. If there are more messages to be consumed while the running threads are busy, then additional threads are started until the total number of threads reaches the value of the maximum number of concurrent consumers (ie., `MaxConcurrentConsumers`). The maximum number of concurrent consumers (or the number of JMS proxies) that can be deployed is limited by the base transport worker pool that is used by the JMS transport. The size of this worker pool can be configured via the system property '`lst_t_core`' and '`lst_t_max`' as described above. The number of concurrent producers are generally limited by the Synapse core worker pool.

Note

Concurrent consumers are only applicable to JMS queues not for JMS topics.

To increase the JMS listener performance

Add the following parameters to the JMS listener configuration of the <ESB_HOME>/repository/conf/axis2/axis2.xml file:

```
<parameter name="transport.jms.ConcurrentConsumers" locked="false">50</parameter>
<parameter name="transport.jms.MaxConcurrentConsumers" locked="false">50</parameter>
```

Parameters

- `transport.jms.ConcurrentConsumers`: the concurrent threads that need to be started to consume messages when polling.
- `transport.jms.MaxConcurrentConsumers`: the maximum number of concurrent threads to use during polling.

Enable caching

Add the following parameter to the JMS listener configuration of the <ESB_HOME>/repository/conf/axis2/axis2.xml file to enable caching:

```
<parameter name="transport.jms.CacheLevel">consumer</parameter>
```

The possible values for the cache level are `none`, `auto`, `connection`, `session` and `consumer`. Out of the possible values, `consumer` is the highest level that provides maximum performance.

After adding concurrency consumers and cache level, your complete configuration would be as follows:

Sample JMS listener configuration with concurrent consumers and caching

```
<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
...
<parameter name="myQueueConnectionFactory" locked="false">
<parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
<parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
<parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
<parameter name="transport.jms.ConnectionFactoryType" locked="false">queue</parameter>

<parameter name="transport.jms.ConcurrentConsumers" locked="false">50</parameter>
<parameter name="transport.jms.MaxConcurrentConsumers" locked="false">50</parameter>
<parameter name="transport.jms.CacheLevel">consumer</parameter>
</parameter>
...
</transportReceiver>
```

Configuring JMS Sender

Enabling caching

Add the following parameter to the JMS sender configuration of the <ESB_HOME>/repository/conf/axis2/axis2.xml file:

is2.xml file:

```
<parameter name="transport.jms.CacheLevel">producer</parameter>
```

The possible values for the cache level are none, auto, connection, session and producer. Out of the possible values, producer is the highest level that provides maximum performance.

When using producer as the cache level, ensure to add the JMS destination parameter to avoid the following error:

```
INFO - AxisEngine [MessageContext:  
logID=2eabe85aeeb3bb62c26bb46d21b11b087ebf1e5e0b350839] JMSCC0029: A  
destination must be specified when sending from this producer.
```

Remove ClientApiNonBlocking when sending messages via JMS

By default, Axis2 spawns a new thread to handle each outgoing message. To change this behavior, you need to remove the ClientApiNonBlocking property from the message.

Note

Removal of this property can be vital when queuing transports like JMS are involved.

To remove the ClientApiNonBlocking property

Add the following parameter to the configuration:

```
<property name="ClientApiNonBlocking" action="remove" scope="axis2" />
```

Tuning the VFS Transport

The Virtual File System (VFS) transport is used by WSO2 ESB to process files in a specified source directory. It supports numerous file systems such as ftp, ftps, sftp, local and samba.

When you work with the VFS transport, you might have a scenario where you need to send large files to a destination. If you use the normal VFS configuration, the memory consumption will be very high since the ESB builds the file inside it. To overcome this issue, WSO2 ESB provides VFS file streaming support. With VFS file streaming, only the stream is passed and therefore memory consumption is less.

To enable VFS file streaming

To use the streaming mode with the VFS transport, follow the steps below:

1. In <ESB_HOME>/repository/conf/axis2/axis2.xml, in the messageBuilders section, add the binary message builder as follows:

```
<messageBuilder contentType="application/binary"  
class="org.apache.axis2.format.BinaryBuilder" />
```

and in the messageFormatters section, add the binary message formatter as follows:

```
<messageFormatter contentType="application/binary"  
class="org.apache.axis2.format.BinaryFormatter"/>
```

2. In the proxy service where you use the VFS transport, add the following parameter to enable streaming:

```
<parameter name="transport.vfs.Streaming">true</parameter>
```

3. In the same proxy service, before the Send mediator, add the following property:

Note

You also need to add the following property if you want to use the VFS transport to transfer files from VFS to VFS.

```
<property name="ClientApiNonBlocking" value="true" scope="axis2"  
action="remove"/>
```

For more information, see [Example 3 of the Send Mediator](#).

Following is a sample configuration that uses the VFS transport to handle large files:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
    <proxy name="StockQuoteProxy" transports="vfs">
        <parameter name="transport.vfs.FileURI">smb://host/test/in</parameter>
        <parameter name="transport.vfs.ContentType">text/xml</parameter>
        <parameter name="transport.vfs.FileNamePattern">.*\.\xml</parameter>
        <parameter name="transport.PollInterval">15</parameter>
        <parameter name="transport.vfs.Streaming">true</parameter>
        <parameter
name="transport.vfs.MoveAfterProcess">smb://host/test/original</parameter>
        <parameter
name="transport.vfs.MoveAfterFailure">smb://host/test/original</parameter>
            <parameter name="transport.vfs.ActionAfterProcess">MOVE</parameter>
            <parameter name="transport.vfs.ActionAfterFailure">MOVE</parameter>
        <target>
            <inSequence>
                <property name="transport.vfs.ReplyFileName"
expression="fn:concat(fn:substring-after(get-property('MessageID'), 'urn:uuid:'),
'.xml')" scope="transport"/>
                    <property action="set" name="OUT_ONLY" value="true"/>
                    <property name="ClientApiNonBlocking" value="true" scope="axis2"
action="remove"/>
                <send>
                    <endpoint>
                        <address uri="vfs:smb://host/test/out" />
                    </endpoint>
                </send>
            </inSequence>
        </target>
        <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl" />
    </proxy>
</definitions>

```

Tuning the RabbitMQ Transport

RabbitMQ transport allows you to send or receive AMQP messages by calling an AMQP broker (RabbitMQ) directly. In order to improve the performance of the RabbitMQ transport you can do following.

- Set queue and exchange declaration parameters to false
- Increase the connection pool size
- Reuse the connection factory in the publisher

Set queue and exchange declaration parameters to false

Setting `rabbitmq.queue.autodeclare` and `rabbitmq.exchange.autodeclare` parameters in the publish url to `false` can improve the RabbitMQ transport performance.

When these parameters are set to `false` RabbitMQ does not try to create queues or exchanges if queues or exchanges are not present. However, you should set these parameters if and only if queues and exchanges are declared prior on the broker.

Increase the connection pool size

You can increase the connection pool size to improve the performance of the RabbitMQ sender and listener. The default connection pool size is 20. To change this, specify a required value for the following parameter in the

RabbitMQ transport sender and listener configurations in the <ESB_HOME>/repository/conf/axis2/axis2.xml file.

```
<parameter name="rabbitmq.connection.pool.size" locked="false">25</parameter>
```

Sample Receiver Configuration

```
<transportReceiver name="rabbitmq"
class="org.apache.axis2.transport.rabbitmq.RabbitMQListener">
    <parameter name="AMQPConnectionFactory" locked="false">
        <parameter name="rabbitmq.server.host.name"
locked="false">localhost</parameter>
        <parameter name="rabbitmq.server.port" locked="false">5672</parameter>
        <parameter name="rabbitmq.server.user.name" locked="false"></parameter>
        <parameter name="rabbitmq.server.password" locked="false"></parameter>
        <parameter name="rabbitmq.connection.retry.interval"
locked="false">10000</parameter>
        <parameter name="rabbitmq.connection.retry.count" locked="false">5</parameter>
        <parameter name="rabbitmq.connection.pool.size" locked="false">25</parameter>
    </parameter>
</transportReceiver>
```

Sample Sender Configuration

```
<transportSender name="rabbitmq"
class="org.apache.axis2.transport.rabbitmq.RabbitMQSender">
    <parameter name="RabbitMQConnectionFactory" locked="false">
        <parameter name="rabbitmq.server.host.name"
locked="false">localhost</parameter>
        <parameter name="rabbitmq.server.port" locked="false">5672</parameter>
        <parameter name="rabbitmq.server.user.name" locked="false"></parameter>
        <parameter name="rabbitmq.server.password" locked="false"></parameter>
        <parameter name="rabbitmq.connection.retry.interval"
locked="false">10000</parameter>
        <parameter name="rabbitmq.connection.retry.count"
locked="false">5</parameter>
        <parameter name="rabbitmq.connection.pool.size" locked="false">10</parameter>
    </parameter>
</transportSender>
```

Reuse the connection factory in the publisher

In the publisher url set the connection factory name instead of the connection parameters as specified below in the rabbitmq.connection.factory parameter . This reuses the connection factories and thereby improves performance.

```
<address
uri="rabbitmq://?rabbitmq.connection.factory=RabbitMQConnectionFactory&amp;rabbitmq.queue.name=queue1&amp;rabbitmq.queue.routing.key=queue1&amp;rabbitmq.replyto.name=replyqueue&amp;rabbitmq.exchange.name=ex1&amp;rabbitmq.queue.autodeclare=false&amp;rabbitmq.exchange.autodeclare=false&amp;rabbitmq.replyto.name=response_queue"/>
```

Tuning Inbound Endpoints

An inbound endpoint is a message entry point that can inject messages directly from the transport layer to the mediation layer, without going through the Axis engine. This section describes how you can tune the HTTP,HTTPS as well as the Kafka inbound endpoint for better performance.

Improving the performance of HTTP/HTTPS inbound

The HTTP inbound protocol is used to separate endpoint listeners for each HTTP inbound endpoint, so that messages are handled separately. The HTTP inbound endpoint can bypass the inbound side axis2 layer and directly inject messages to a given sequence or API. For proxy services, messages are routed through the axis2 transport layer in a manner similar to normal transports. You can start dynamic HTTP inbound endpoints without restarting the server.

By default inbound endpoints share the PassThrough transport worker pool to handle incoming requests. If you need a separate worker pool for the inbound endpoint to increase the performance, you need to configure the following parameters:

Parameter	Description
inbound.worker.pool.size.core	The initial number of threads in the worker thread pool. It is based on the number of messages to be processed. The value here is the value of the inbound.worker.pool.size.core parameter.
inbound.worker.pool.size.max	The maximum number of threads in the worker thread pool. It is used to avoid performance degradation that can occur due to constant thread creation and destruction.
inbound.worker.thread.keep.alive.sec	The keep-alive time for extra threads in the worker pool. It is used to avoid unnecessary thread creation and destruction. When this time is elapsed for an extra thread, it is terminated. The value for this parameter is to optimize the usage of resources by avoiding extra threads that are not utilized.
inbound.worker.pool.queue.length	The length of the queue that is used to hold runnable tasks. The thread pool starts queuing jobs when all existing threads have reached the maximum number of threads. The value for this parameter is the maximum length of the queue. If a bounded queue is used and the queue gets filled up, new threads will fail to submit jobs causing synapse to drop some messages.
inbound.thread.group.id	Unique Identifier of the thread group.
inbound.thread.id	Unique Identifier of the thread.
dispatch.filter.pattern	The regular expression that defines the proxy services a endpoint. Provide the <code>.*</code> expression to expose all proxy services. For example, the expression <code>^(/foo /bar /services/MyProxy)</code> exposes all services via the inbound endpoint. If you do not provide a filter, none of the inbound endpoint will be accessible.

Improving the performance of Kafka inbound

WSO2 ESB kafka inbound endpoint acts as a message consumer. It creates a connection to zookeeper and requests messages for a topic, topics or topic filters.

You can follow the recommendations described below to gain the maximum performance with the Kafka inbound.

- Set the `sequential` parameter to `false` to use the Kafka inbound in a non-sequential mode as it allows better performance than the sequential mode.

```
<parameter name="sequential">false</parameter>
```

- Change the inbound thread pool size based on your use case. Recommended values are specified below. These parameters can be configured in the <ESB_HOME>/repository/conf/synapse.properties file.

```
inbound.threads.core = 200
inbound.threads.max = 1000
```

Tuning the Performance based on Use Case

This section walks you through the tuning best practices you can follow for specific use cases. Following are the use cases for which tuning best practices are described. Click on a required use case to see detailed information:

- [Split-Aggregate Pattern](#)
- [Message Transformations](#)
- [JMS Scenarios](#)

Split-Aggregate Pattern

The split-aggregate pattern sends an incoming request from the client to several target endpoints simultaneously. Then it combines all the responses from each back-end to a single response, and sends the response back to the client.

This pattern can be implemented in the ESB using the [Iterate mediator](#), [Clone mediator](#) and [Aggregate mediator](#). The Iterate mediator splits the message into a number of different messages that are derived from the parent message using Xpath, and sends the messages to target endpoints using the same sequence. The Clone mediator sends the same message to target endpoints using a different target sequences. The Aggregate mediator collects all the response messages and creates one response message.

Split-aggregate use case

The following use case demonstrates the split-aggregate pattern:

The Iterate mediator splits the original request message to a number of different messages using the Xpath expression `//m0:getQuote/m0:request`. Then the messages are sent to the SimpleStockQuoteService soap back-end. All the responses are aggregated and sent back to the client at the OutSequence.

Following is the proxy service used in this scenario:

```

<?xml version="1.0" encoding="UTF-8"?>
<proxy xmlns="http://ws.apache.org/ns/synapse" name="SplitAggregateProxy"
startOnLoad="true">
    <target>
        <inSequence>
            <iterate xmlns:m0="http://services.samples" preservePayload="true"
                    attachPath="//m0:getQuote" expression="//m0:getQuote/m0:request">
                <target>
                    <sequence>
                        <send>
                            <endpoint>
                                <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
                            </endpoint>
                        </send>
                    </sequence>
                </target>
            </iterate>
        </inSequence>
        <outSequence>
            <aggregate>
                <completeCondition>
                    <messageCount/>
                </completeCondition>
                <onComplete xmlns:m0="http://services.samples"
expression="//m0:getQuoteResponse">
                    <send/>
                </onComplete>
            </aggregate>
        </outSequence>
    </target>
</proxy>

```

The sample request sent to the proxy service is given below. This request is split into 4 messages and sent to the SimpleStockQuoteService. Then, the Aggregate mediator aggregates the responses and sends back the aggregated response.

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
  <soapenv:Header/>
  <soapenv:Body>
    <ser:getQuote>
      <ser:request>
        <xsd:symbol>ABC</xsd:symbol>
      </ser:request>
      <ser:request>
        <xsd:symbol>WSO2</xsd:symbol>
      </ser:request>
      <ser:request>
        <xsd:symbol>AAA</xsd:symbol>
      </ser:request>
      <ser:request>
        <xsd:symbol>BBB</xsd:symbol>
      </ser:request>
    </ser:getQuote>
  </soapenv:Body>
</soapenv:Envelope>

```

Tuning the performance

The Iterate, Clone and Aggregate mediators demonstrate high performance due to default threading and memory configuration. The performance of these mediators can be further increased by tuning the following parameters in the `<ESB_HOME>/repository/conf/synapse.properties` file:

`synapse.threads.core` and `synapse.threads.max`

Iterate and Clone mediators use a thread pool to create new threads when processing messages and sending messages parallelly. You can configure the size of the thread pool by the `synapse.threads.core` parameter. The number of threads specified via this parameter should be increased as required to balance an increased load. Increasing the value specified for this parameter results in higher performance of the Iterate and Clone mediators. You can specify the maximum number of synapse threads in the pool by the `synapse.threads.max` parameter.

`synapse.threads.keepalive`

The keep-alive time for extra threads is in milliseconds. This parameter is applicable only if the Iterate or the Clone mediator is used to handle a high load. You can increase the number of threads specified via this parameter as required to balance an increased load.

`synapse.threads.qlen`

You can use this parameter to specify the length of the queue that is used to hold the runnable tasks to be executed by the pool. You can specify a finite value as the queue length by giving any positive number. If this parameter is set to (-1) it means that the task queue length is infinite.

If the queue length is finite there can be situations where requests are rejected when the task queue is full, and all the cores are occupied. If the queue length is infinite, and if some thread locking happens, the server can go out of memory. Therefore, you need to decide on an optimal value based on the actual load.

Message Transformations

Message Transformation is a common WSO2 ESB use case. When it comes to message transformations, you can use the [FastXSLT mediator](#), [XSLT mediator](#), [Script mediator](#) as well as the [PayloadFactory mediator](#). This section describes different message transformation use cases and parameters you can configure for optimal performance.

Comparison of the fast XSLT mediator and the XSLT mediator

The FastXSLT Mediator and XSLT mediator can be used to do XSLT based message transformations in WSO2

ESB. Both mediators can be used to perform complex message transformations. The FastXSLT mediator has significant performance over the XSLT mediator as it uses the streaming XPath parser and applies the XSLT transformation to the message stream instead of applying it to the XML message payload, but with the FastXSLT mediator you cannot specify the source, properties, features, or resources as you can with the XSLT mediator.

Note

Always use the FastXSLT mediator instead of the default XSLT mediator for better performance in implementations where the original message remains unmodified.

In implementations where the message payload needs to be pre-processed, use the XSLT mediator instead of the FastXSLT mediator. Any pre-processing performed on the message payload is not visible to the FastXSLT mediator, because the transformation logic is applied to the original message stream instead of the message payload.

Optimizing the XSLT Mediator

In real world scenarios, you might come across use cases where you do pre-processing before doing the XSLT message transformation using the XSLT mediator. In such use cases, you have to use the XSLT mediator instead of the FastXSLT mediator. You can specify the source, properties, features, or resources with the XSLT mediator, but the downside with the XSLT mediator is that it does not perform well when the message size is larger.

It is possible to tune the XSLT mediator configuration to perform better for large message sizes.

In the XSLT mediator, the following two parameters control the memory usage of the the server. These two parameters can be configured in the <ESB_HOME>/repository/conf/synapse.properties file.

```
synapse.temp_data.chunk.size=3072
synapse.temp_data.chunk.threshold=8
```

These parameters decide when to write to the file system when the message size is large. The default values for the parameters allow WSO2 ESB to process messages of size $3072 \times 8 = 24K$ with the XSLT mediator without writing to the file system. This means that there is no performance drop in the XSLT mediator when the message size is less than 24K, but when the message size is larger than that, you see a clear performance degradation.

You can improve the performance of the XSLT mediator by allowing the use of more memory when processing large messages with the XSLT mediator.

For example if you know that your message size is going to be less than 512k (i.e., 16384×32), you can set the values of the above parameters as follows:

```
synapse.temp_data.chunk.size=16384
synapse.temp_data.chunk.threshold=32
```

Then the XSLT transformation happens in memory without writing data to disk. Therefore, performance is increased.

Optimizing the fastXSLT Mediator

If you want to perform transformation on the original message, you can use the FastXSLT mediator. Any pre-processing that applies to the message payload is not be visible to the FastXSLT mediator as the transformation logic is applied to the message stream instead of the message payload.

If you cannot perform your transformation with the FastXSLT mediator, then it is recommended to write a custom class mediator to do your transformation, since it will be much faster than the XSLT mediator.

You can enable the FastXSLT mediator by following the steps below:

- Add the following parameter in the <ESB_HOME>/repository/conf/synapse.properties file to enable streaming XPath for fastXSLT mediator.

```
synapse.streaming.xpath.enabled=true
```

- Configure the following parameters in the <ESB_HOME>/repository/conf/synapse.properties file depending on the expected message size.

```
synapse.temp_data.chunk.size=3072
synapse.temp_data.chunk.threshold=32
```

Note

To enable the FastXSLT mediator, your XSLT script must include the following parameter in the XSL output.

```
omit-xml-declaration="yes"
```

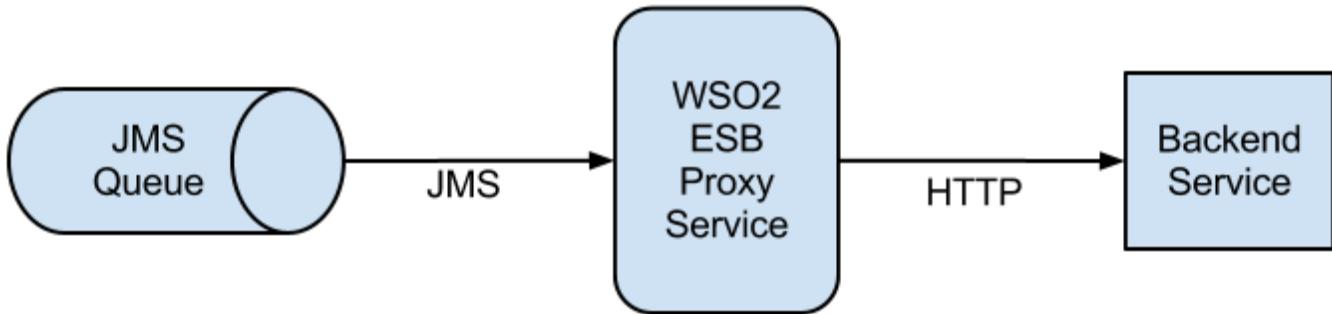
For example,

```
<xsl:output method="xml" omit-xml-declaration="yes" encoding="UTF-8"
indent="yes" />
```

JMS Scenarios

This section provides information on how to improve the performance of the following most common JMS use cases of WSO2 ESB.

ESB as a JMS consumer



In this scenario, WSO2 ESB listens to a JMS queue, consumes messages, and sends the messages to a HTTP back-end service.

You can improve the performance of this scenario by following the steps below.

- If the queue gets filled up at a high rate, and the queue is long, you can improve the performance by increasing the number of concurrent consumers. Add the following parameters to the JMS listener

configuration of the <ESB_HOME>/repository/conf/axis2/axis2.xml file to increasing the number of concurrent consumers:

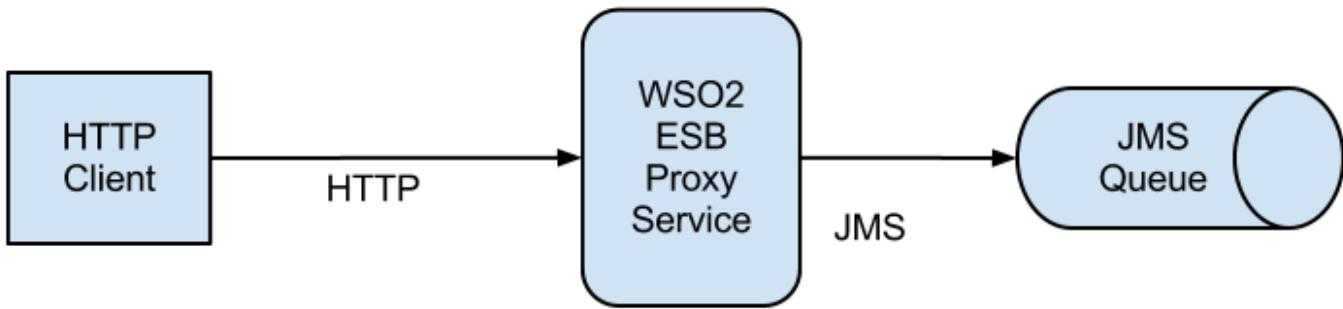
```
<parameter name="transport.jms.ConcurrentConsumers" locked="false">50</parameter>
<parameter name="transport.jms.MaxConcurrentConsumers" locked="false">50</parameter>
```

- Add the following parameter to the JMS listener configuration of the <ESB_HOME>/repository/conf/axis2/axis2.xml file to enable JMS listener caching:

```
<parameter name="transport.jms.CacheLevel">consumer</parameter>
```

The possible values for the cache level are none, auto, connection, session and consumer. Out of the possible values, consumer is the highest level that provides maximum performance.

ESB as a JMS Producer



In this scenario, an ESB proxy service accepts messages from an HTTP client via HTTP and sends the messages to a JMS queue.

You can improve the performance of this scenario by following the steps below.

- Add the following parameter to the JMS sender configuration of the <ESB_HOME>/repository/conf/axis2/axis2.xml file to enable JMS sender caching:

```
<parameter name="transport.jms.CacheLevel">producer</parameter>
```

The possible values for the cache level are none, auto, connection, session and producer. Out of the possible values, producer is the highest level that provides maximum performance.

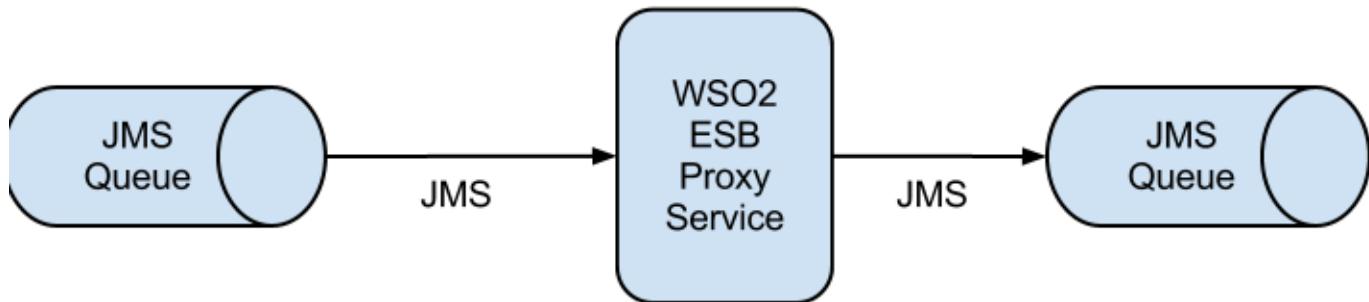
- Add the following parameter to the configuration to remove ClientApiNonBlocking when sending messages via JMS:

```
<property name="ClientApiNonBlocking" action="remove" scope="axis2"/>
```

Noe

By default, Axis2 spawns a new thread to handle each outgoing message. To change this behavior, you need to remove the `ClientApiNonBlocking` property from the message. Removal of this property is vital when queuing transports like JMS are involved.

ESB as Both a JMS Producer and Consumer



In this scenario, ESB listens to a JMS queue and consume messages, and also sends messages to a JMS queue.

To improve the performance of this scenario, you can follow the steps described in the scenario where the [ESB acts as a JMS consumer](#) as well the steps described in the scenario where the [ESB acts as a JMS producer](#).

Tuning Analytics

This section explains how to tune parameters that control the performance of ESB Analytics when it is affected by high loads, network traffic etc. These parameters should be tuned based on the deployment environment.

Tuning carbon.xml parameters

The following parameter is configured in the `<ESB_HOME>/repository/conf/carbon.xml` file.

Parameter	Description	Default Value	Tuning Recommendation

AnalyticsServerPublishingInterval	The number of milliseconds that should elapse after a batch of statistical data is processed to be published in the Analytics Dashboard before sending another batch.	2000	<p>The default value of 2000 milliseconds (i.e. 5 seconds) is recommended.</p> <p>When WSO2 ESB is handling a high load of requests, this value can be reduced to increase the frequency with which the resulting statistics published in the Analytics Dashboard. This helps to avoid storing too much data in the ESB server causing an overconsumption of memory.</p> <p>When the load of requests handled by the ESB is comparatively low, this time interval can be increased to reduce the system overhead incurred by frequent processing.</p>
-----------------------------------	---	------	---

Tuning data-agent parameters

The parameters in the `<ESB-HOME>/repository/conf/data-bridge/data-agent-config.xml` file can be tuned as follows.

Parameter	Description	Default Value	Tuning Recommendation
QueueSize	The number of messages that can be stored in the ESB server at a given time before they are sent to be published in the Analytics Dashboard.	32768	<p>This value should be increased when the ESB Analytics server is busy due to a request overload or if there is high network traffic. This prevents the generation of the queue full, dropping message error.</p> <p>When the ESB Analytics server is not very busy and when the network traffic is relatively low, the queue size can be reduced to avoid an overconsumption of memory.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <p>The number specified for this parameter should be a power of 2.</p> </div>

BatchSize	The ESB statistical data sent to the ESB Analytics server to be published in the Analytics Dashboard are grouped into batches. This parameter specifies the number of requests to be included in a batch.	200	This value should be tuned in proportion to the volume of requests sent from the ESB server to the ESB Analytics server. This value should be reduced if you want to reduce the system overhead of the ESB Analytics server. This value should be increased if the ESB server is generating a high amount of statistics and the QueueSize cannot be further increased without causing an overconsumption of memory.	
CorePoolSize	The number of threads allocated to publish ESB statistical data to the Analytics Dashboard via Thrift at the time the ESB server is started. This value increases when the throughput of statistics generated increases. However, the number of threads will not exceed the number specified for the MaxPoolSize parameter.	1	The number of available CPU cores should be taken into account when specifying this value. Increasing the core pool size may improve the throughput of statistical data published in the Analytics Dashboard, but latency will also be increased due to context switching.	
MaxPoolSize	The maximum number of threads that should be allocated at any given time to publish ESB statistical data in the Analytics Dashboard.	1	The number of available CPU cores should be taken into account when specifying this value. Increasing the maximum core pool size may improve the throughput of statistical data published in the Analytics Dashboard, since more threads can be spawned to handle an increased number of events. However, latency will also increase since a higher number of threads would cause context switching to take place more frequently.	

▼ [Click here to view a list of errors that can be avoided by tuning the above parameters](#)

- TID: [-1234] [] [2016-05-05 19:12:02,393] WARN
`{org.wso2.carbon.databridge.agent.DataPublisher}` - Event queue is full, unable to process the event for endpoint group [(Receiver URL : `tcp://127.0.0.1:7611`, Authentication URL : `ssl://127.0.0.1:7711`), dropping the event.
`{org.wso2.carbon.databridge.agent.DataPublisher}`
- TID: [-1] [] [2016-05-05 19:11:18,642] ERROR
`{org.wso2.carbon.databridge.agent.endpoint.DataEndpoint}` - Unable to send events to the endpoint.
`{org.wso2.carbon.databridge.agent.endpoint.DataEndpoint}`
`org.wso2.carbon.databridge.agent.exception.DataEndpointException`: Cannot send Events
- TID: [-1] [] [2016-05-05 19:11:18,642] ERROR
`{org.wso2.carbon.databridge.agent.endpoint.DataEndpoint}` - Unable to send events to the endpoint.
`{org.wso2.carbon.databridge.agent.endpoint.DataEndpoint}`
`org.wso2.carbon.databridge.agent.exception.DataEndpointException`: Cannot send Events

JMX Monitoring

WSO2 ESB exposes a number of management resources as JMX MBeans that can be used for managing and monitoring the running server. These MBeans can be accessed remotely using a JMX client such as JConsole. This

page describes how to monitor the ESB using JMX in the following sections:

- [JMX service URL](#)
- [Using JConsole](#)
- [Attributes and Operations of MBeans](#)
 - Transport MBeans
 - Latency MBeans
 - Connection MBeans
 - Threading MBeans

JMX service URL

When WSO2 ESB is starting up, it will display the JMX Service URL in the console as follows.

```
INFO - JMXServerManager JMX Service URL :
service:jmx:rmi://localhost:11111/jndi/rmi://localhost:9999/jmxrmi
```

This URL can be used to remotely access the JMX Service exposed by WSO2 ESB. The following section describes how to use JConsole as the JMX client to access this service.

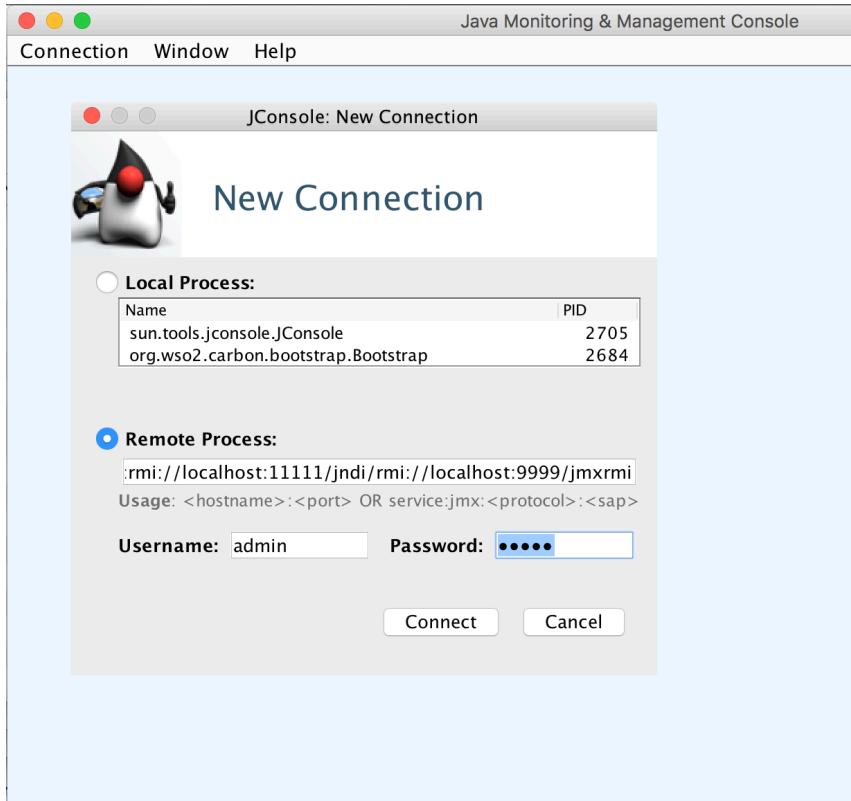
Using JConsole

Start JConsole and enter the above URL as the JMX Service URL. Type `admin` in both username and password fields and click **Connect**.

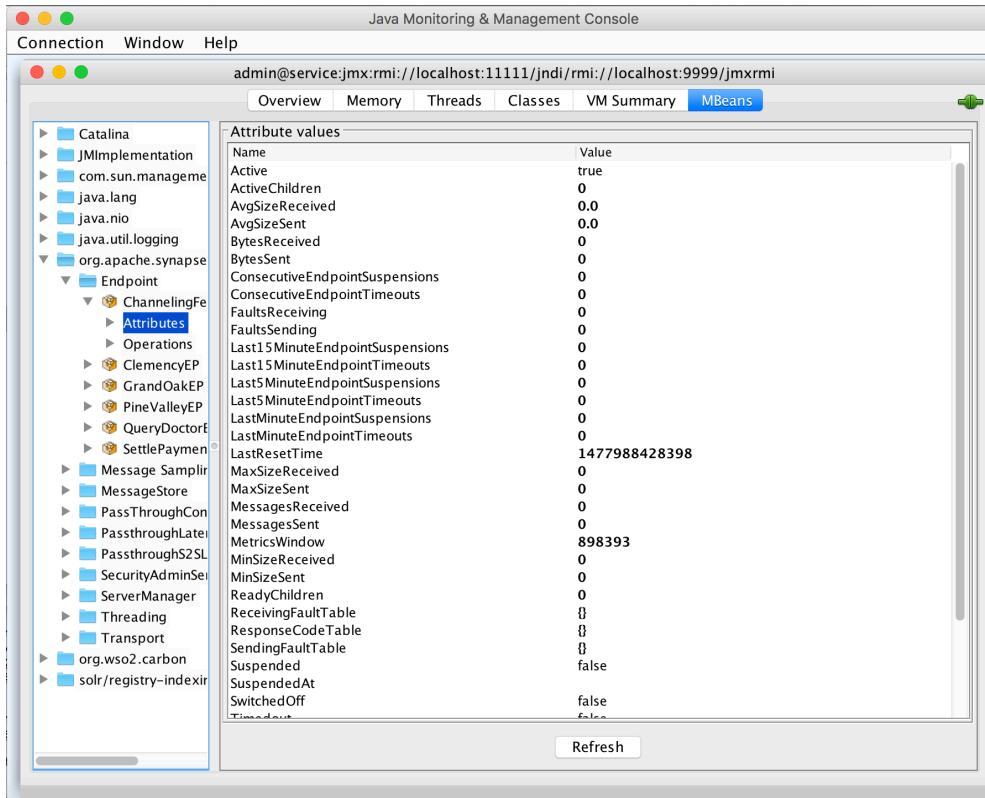
Make sure that the user ID you are using for JMX monitoring is assigned a role which has the **Server Admin** permission in the **Permissions of the Role** page. See [Configuring Roles](#) in WSO2 Administration Guide for further information about configuring roles assigned to users.

Tip

If you are intending to establish a remote connection (over a network or into a VM), make sure that the server has been started with the `-Djava.rmi.server.hostname=<IP_ADDRESS_WHICH_YOU_USE_TO_CONNECT_TO_SERVER>` option. For more information read [Troubleshooting Connection Problems in JConsole](#).



After successfully connecting to the JMX service, click on "MBeans" tab of JConsole to view MBeans exposed by the ESB, which are listed under the org.apache.axis2, org.apache.synapse, and org.wso2.carbon nodes. The following section summarizes the attributes and operations available in MBeans exposed by the WSO2 ESB.



Attributes and Operations of MBeans

Transport MBeans

For each transport listener and sender enabled in the ESB, there will be an MBean under the org.apache.axis2/Transport node that has the following attributes and operations. For example, when the JMS transport is enabled, the following MBean will be exposed:

- org.apache.axis2/Transport/jms-sender-n

Additionally, the MBeans under the org.apache.synapse/Transport node expose these attributes for the default HTTP and HTTPS transports:

- org.apache.synapse/Transport/passthru-http-receiver
- org.apache.synapse/Transport/passthru-http-sender
- org.apache.synapse/Transport/passthru-https-receiver
- org.apache.synapse/Transport/passthru-https-sender

Attribute/Operation Name	Description
ActiveThreadCount	Threads active in this transport listener/sender.
AvgSizeReceived	Average size of received messages.
AvgSizeSent	Average size of sent messages.
BytesReceived	Number of bytes received through this transport.
BytesSent	Number of bytes sent through this transport.
FaultsReceiving	Number of faults encountered while receiving.
FaultsSending	Number of faults encountered while sending.
LastResetTime	Last time transport listener/sender statistic recording was reset.
MaxSizeReceived	Maximum message size of received messages.
MaxSizeSent	Maximum message size of sent messages.
MetricsWindow	Time difference between current time and last reset time in milliseconds.
MinSizeReceived	Minimum message size of received messages.
MinSizeSent	Minimum message size of sent messages.
MessagesReceived	Total number of messages received through this transport.
MessagesSent	Total number of messages sent through this transport.
QueueSize	Number of messages currently queued. Messages get queued if all the worker threads in this transport thread pool are busy.
ResponseCodeTable	Number of messages sent against their response codes.
TimeoutsReceiving	Message receiving timeout.
TimeoutsSending	Message sending timeout.
resetStatistics()	Clear recorded transport listener/sender statistics and restart recording.
start()	Start this transport listener/sender.

stop()	Stop this transport listener/sender.
resume()	Resume this transport listener/sender which is currently paused.
resetStatistics()	Clear recorded transport listener/sender statistics and restart recording.
pause()	Pause this transport listener/sender which has been started.
maintenenceShutdown(long gracePeriod)	Stop processing new messages, and wait the specified maximum time for in-flight requests to complete before a controlled shutdown for maintenance.

Latency MBeans

This view provides statistics of the latencies from all backend services connected through the HTTP and HTTPS transports. These statistics are provided as an aggregate value.

- org.apache.synapse/PassthroughLatencyView/nio-http-http
- org.apache.synapse/PassthroughLatencyView/nio-https-https

Attribute/Operation Name	Description
AllTimeAvgLatency	Average latency since latency recording was last reset.
LastXxxAvgLatency	Average latency for last Xxx time period. For example, LastHourAvgLatency return the average latency for last hour.
LastResetTime	Last time latency statistic recording was reset.
reset()	Clear recorded latency statistics and restart recording.

Connection MBeans

These MBeans provide connection statistics for the HTTP and HTTPS transports. For example:

- org.apache.synapse/PassThroughConnections/http-listener
- org.apache.synapse/PassThroughConnections/http-sender
- org.apache.synapse/PassThroughConnections/https-listener
- org.apache.synapse/PassThroughConnections/https-sender

Attribute/Operation Name	Description
ActiveConnections	Number of currently active connections.
ActiveConnectionsPerHosts	A map of number of connections against hosts.
LastXxxConnections	Number of connections created during last Xxx time period.
RequestSizesMap	A map of number of requests against their sizes.
ResponseSizesMap	A map of number of responses against their sizes.
LastResetTime	Last time connection statistic recordings was reset.
reset()	Clear recorded connection statistics and restart recording.

Threading MBeans

These MBeans are only available in the NHTTP transport and not in the default Pass Through transport.

- org.apache.synapse/Threading/HttpClientWorker
- org.apache.synapse/Threading/HttpServerWorker

Attribute/Operation Name	Description
TotalWorkerCount	Total worker threads related to this server/client.
AvgUnblockedWorkerPercentage	Time-averaged unblocked worker thread percentage.
AvgBlockedWorkerPercentage	Time-averaged blocked worker thread percentage.
LastXxxBlockedWorkerPercentage	Blocked worker thread percentage averaged for last Xxx time period.
DeadLockedWorkers	Number of deadlocked worker threads since last statistics reset.
LastResetTime	Last time thread statistic recordings was reset.
reset()	Clear recorded thread statistic and restart recording.

SNMP Monitoring

Simple Network Management Protocol (SNMP) is an Internet-standard protocol for managing devices on IP networks. Given below is how to configure SNMP in WSO2 ESB, which exposes various MBeans via SNMP.

1. Download the following jar files from <http://www.snmp4j.org> and add them to <ESB_HOME>/repository/components/lib.
 - snmp4j-2.1.0.jar
 - snmp4j-agent-2.0.6.jar
2. Enable SNMP in the <ESB_HOME>/repository/conf/synapse.properties file by adding the following entry:

```
synapse.snmp.enabled=true
```

The ESB can now monitor MBeans with SNMP. For example:

```
Monitoring Info : OID branch "1.3.6.1.4.1.18060.14" with the following sub-branches:
1 - ServerManager MBean
2 - Transport MBeans
3 - NHttpConnections MBeans
4 - NHTTPLatency MBeans
5 - NHTTPS2SLatency MBeans
```

MBean OID mappings

Following are the OID equivalents of the server manager and transport MBeans, which are described in [JMX Monitoring](#):

Name=ServerManager@ServerState as OID: 1.3.6.1.4.1.18060.14.1.21.1.0

Name=passthru-http-sender@ActiveThreadCount as OID: 1.3.6.1.4.1.18060.14.2.17.1.0

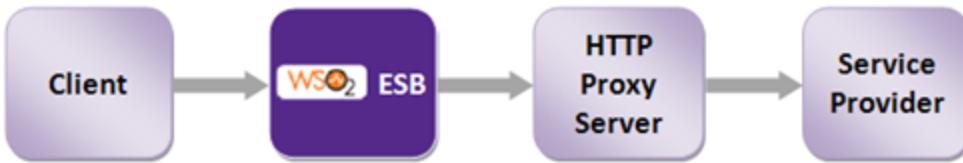
Name=passthru-http-sender@AvgSizeReceived as OID: 1.3.6.1.4.1.18060.14.2.17.2.0

Name=passthru-http-sender@AvgSizeSent as OID: 1.3.6.1.4.1.18060.14.2.17.3.0
Name=passthru-http-sender@BytesReceived as OID: 1.3.6.1.4.1.18060.14.2.17.4.0
Name=passthru-http-sender@BytesSent as OID: 1.3.6.1.4.1.18060.14.2.17.5.0
Name=passthru-http-sender@FaultsReceiving as OID: 1.3.6.1.4.1.18060.14.2.17.6.0
Name=passthru-http-sender@FaultsSending as OID: 1.3.6.1.4.1.18060.14.2.17.7.0
Name=passthru-http-sender@LastResetTime as OID: 1.3.6.1.4.1.18060.14.2.17.8.0
Name=passthru-http-sender@MaxSizeReceived as OID: 1.3.6.1.4.1.18060.14.2.17.9.0
Name=passthru-http-sender@MaxSizeSent as OID: 1.3.6.1.4.1.18060.14.2.17.10.0
Name=passthru-http-sender@MessagesReceived as OID: 1.3.6.1.4.1.18060.14.2.17.11.0
Name=passthru-http-sender@MessagesSent as OID: 1.3.6.1.4.1.18060.14.2.17.12.0
Name=passthru-http-sender@MetricsWindow as OID: 1.3.6.1.4.1.18060.14.2.17.13.0
Name=passthru-http-sender@MinSizeReceived as OID: 1.3.6.1.4.1.18060.14.2.17.14.0
Name=passthru-http-sender@MinSizeSent as OID: 1.3.6.1.4.1.18060.14.2.17.15.0
Name=passthru-http-sender@QueueSize as OID: 1.3.6.1.4.1.18060.14.2.17.16.0
Name=passthru-http-sender@TimeoutsReceiving as OID: 1.3.6.1.4.1.18060.14.2.17.18.0
Name=passthru-http-sender@TimeoutsSending as OID: 1.3.6.1.4.1.18060.14.2.17.19.0
Name=passthru-https-sender@ActiveThreadCount as OID: 1.3.6.1.4.1.18060.14.2.18.1.0
Name=passthru-https-sender@AvgSizeReceived as OID: 1.3.6.1.4.1.18060.14.2.18.2.0
Name=passthru-https-sender@AvgSizeSent as OID: 1.3.6.1.4.1.18060.14.2.18.3.0
Name=passthru-https-sender@BytesReceived as OID: 1.3.6.1.4.1.18060.14.2.18.4.0
Name=passthru-https-sender@BytesSent as OID: 1.3.6.1.4.1.18060.14.2.18.5.0
Name=passthru-https-sender@FaultsReceiving as OID: 1.3.6.1.4.1.18060.14.2.18.6.0
Name=passthru-https-sender@FaultsSending as OID: 1.3.6.1.4.1.18060.14.2.18.7.0
Name=passthru-https-sender@LastResetTime as OID: 1.3.6.1.4.1.18060.14.2.18.8.0
Name=passthru-https-sender@MaxSizeReceived as OID: 1.3.6.1.4.1.18060.14.2.18.9.0
Name=passthru-https-sender@MaxSizeSent as OID: 1.3.6.1.4.1.18060.14.2.18.10.0
Name=passthru-https-sender@MessagesReceived as OID: 1.3.6.1.4.1.18060.14.2.18.11.0
Name=passthru-https-sender@MessagesSent as OID: 1.3.6.1.4.1.18060.14.2.18.12.0
Name=passthru-https-sender@MetricsWindow as OID: 1.3.6.1.4.1.18060.14.2.18.13.0
Name=passthru-https-sender@MinSizeReceived as OID: 1.3.6.1.4.1.18060.14.2.18.14.0
Name=passthru-https-sender@MinSizeSent as OID: 1.3.6.1.4.1.18060.14.2.18.15.0
Name=passthru-https-sender@QueueSize as OID: 1.3.6.1.4.1.18060.14.2.18.16.0
Name=passthru-https-sender@TimeoutsReceiving as OID: 1.3.6.1.4.1.18060.14.2.18.18.0
Name=passthru-https-sender@TimeoutsSending as OID: 1.3.6.1.4.1.18060.14.2.18.19.0
Name=passthru-http-receiver@ActiveThreadCount as OID: 1.3.6.1.4.1.18060.14.2.19.1.0
Name=passthru-http-receiver@AvgSizeReceived as OID: 1.3.6.1.4.1.18060.14.2.19.2.0

Name=passthru-http-receiver@AvgSizeSent as OID: 1.3.6.1.4.1.18060.14.2.19.3.0
Name=passthru-http-receiver@BytesReceived as OID: 1.3.6.1.4.1.18060.14.2.19.4.0
Name=passthru-http-receiver@BytesSent as OID: 1.3.6.1.4.1.18060.14.2.19.5.0
Name=passthru-http-receiver@FaultsReceiving as OID: 1.3.6.1.4.1.18060.14.2.19.6.0
Name=passthru-http-receiver@FaultsSending as OID: 1.3.6.1.4.1.18060.14.2.19.7.0
Name=passthru-http-receiver@LastResetTime as OID: 1.3.6.1.4.1.18060.14.2.19.8.0
Name=passthru-http-receiver@MaxSizeReceived as OID: 1.3.6.1.4.1.18060.14.2.19.9.0
Name=passthru-http-receiver@MaxSizeSent as OID: 1.3.6.1.4.1.18060.14.2.19.10.0
Name=passthru-http-receiver@MessagesReceived as OID: 1.3.6.1.4.1.18060.14.2.19.11.0
Name=passthru-http-receiver@MessagesSent as OID: 1.3.6.1.4.1.18060.14.2.19.12.0
Name=passthru-http-receiver@MetricsWindow as OID: 1.3.6.1.4.1.18060.14.2.19.13.0
Name=passthru-http-receiver@MinSizeReceived as OID: 1.3.6.1.4.1.18060.14.2.19.14.0
Name=passthru-http-receiver@MinSizeSent as OID: 1.3.6.1.4.1.18060.14.2.19.15.0
Name=passthru-http-receiver@QueueSize as OID: 1.3.6.1.4.1.18060.14.2.19.16.0
Name=passthru-http-receiver@TimeoutsReceiving as OID: 1.3.6.1.4.1.18060.14.2.19.18.0
Name=passthru-http-receiver@TimeoutsSending as OID: 1.3.6.1.4.1.18060.14.2.19.19.0
Name=passthru-https-receiver@ActiveThreadCount as OID: 1.3.6.1.4.1.18060.14.2.20.1.0
Name=passthru-https-receiver@AvgSizeReceived as OID: 1.3.6.1.4.1.18060.14.2.20.2.0
Name=passthru-https-receiver@AvgSizeSent as OID: 1.3.6.1.4.1.18060.14.2.20.3.0
Name=passthru-https-receiver@BytesReceived as OID: 1.3.6.1.4.1.18060.14.2.20.4.0
Name=passthru-https-receiver@BytesSent as OID: 1.3.6.1.4.1.18060.14.2.20.5.0
Name=passthru-https-receiver@FaultsReceiving as OID: 1.3.6.1.4.1.18060.14.2.20.6.0
Name=passthru-https-receiver@FaultsSending as OID: 1.3.6.1.4.1.18060.14.2.20.7.0
Name=passthru-https-receiver@LastResetTime as OID: 1.3.6.1.4.1.18060.14.2.20.8.0
Name=passthru-https-receiver@MaxSizeReceived as OID: 1.3.6.1.4.1.18060.14.2.20.9.0
Name=passthru-https-receiver@MaxSizeSent as OID: 1.3.6.1.4.1.18060.14.2.20.10.0
Name=passthru-https-receiver@MessagesReceived as OID: 1.3.6.1.4.1.18060.14.2.20.11.0
Name=passthru-https-receiver@MessagesSent as OID: 1.3.6.1.4.1.18060.14.2.20.12.0
Name=passthru-https-receiver@MetricsWindow as OID: 1.3.6.1.4.1.18060.14.2.20.13.0
Name=passthru-https-receiver@MinSizeReceived as OID: 1.3.6.1.4.1.18060.14.2.20.14.0
Name=passthru-https-receiver@MinSizeSent as OID: 1.3.6.1.4.1.18060.14.2.20.15.0
Name=passthru-https-receiver@QueueSize as OID: 1.3.6.1.4.1.18060.14.2.20.16.0
Name=passthru-https-receiver@TimeoutsReceiving as OID: 1.3.6.1.4.1.18060.14.2.20.18.0
Name=passthru-https-receiver@TimeoutsSending as OID: 1.3.6.1.4.1.18060.14.2.20.19.0

Working with Proxy Servers

When using WSO2 ESB, there can be scenarios where you need to configure WSO2 ESB to route messages through a proxy server. For example, if WSO2 ESB is behind a firewall, your proxy service might need to talk to a server through a proxy server as illustrated in the following diagram:



For such scenarios, you need to configure the ESB transport sender to forward messages through a proxy server.

Configuring WSO2 ESB to route messages through a proxy server

To configure WSO2 ESB to route messages through a proxy server:

- Edit the <ESB_HOME>/repository/conf/axis2/axis2.xml file, and add the following parameters in the <transportSender> configuration of the http transport:
 - `http.proxyHost` - The host name of the proxy server.
 - `http.proxyPort` - The port number of the proxy server.

Following is a sample <transportSender> configuration where the parameters mentioned above are set:

```

<transportSender name="http"
class="org.apache.synapse.transport.passthru.PassThroughHttpSender">
    <parameter name="non-blocking" locked="false">true</parameter>
    <parameter name="http.proxyHost" locked="false">localhost</parameter>
    <parameter name="http.proxyPort" locked="false">8080</parameter>
</transportSender>
  
```

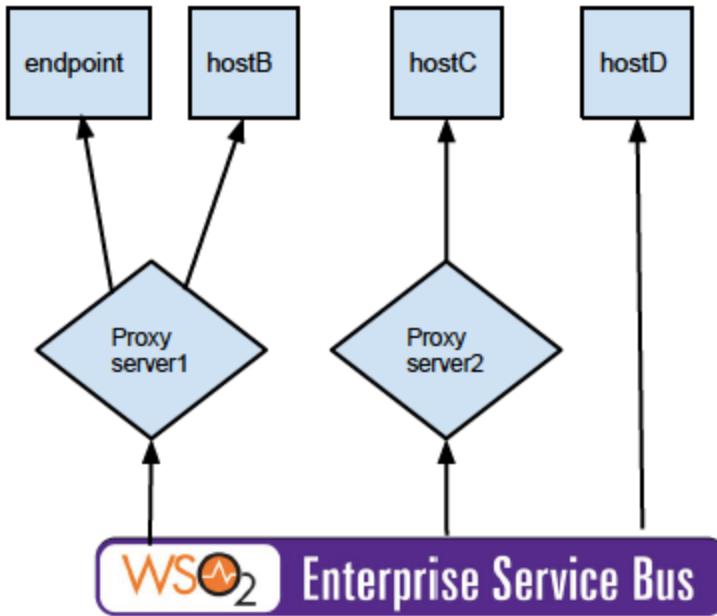
When you configure the <transportSender> with the `http.proxyHost` and `http.proxyPort` parameters , all HTTP requests pass through the configured proxy server.

A proxy server might require HTTP basic authentication before it handles communication from the ESB.

Configuring proxy profiles in WSO2 ESB

If there is a scenario where you need to configure multiple proxy servers to route messages to different endpoints, you can achieve this by configuring proxy profiles. For information on how to configure proxy profiles in the ESB, see [Configuring Proxy Profiles in WSO2 ESB](#).

When using WSO2 ESB, there can be scenarios where you need to configure multiple proxy servers to route messages to different endpoints as illustrated in the following diagram.



When you need to route messages to different endpoints through multiple proxy servers, you can configure proxy profiles.

To configure proxy profiles in WSO2 ESB:

- Edit the `<ESB_HOME>/repository/conf/axis2/axis2.xml` file, add the `proxyProfiles` parameter in the `<transportSender>` configuration of the http transport, and then define multiple profiles based on the number of proxy servers you need to have.

When you define a profile, it is mandatory to specify the `targetHosts`, `proxyHost` and `proxyPort` parameters for each profile.

Following is a sample proxy profile configuration that you can have in the `<transportSender>` configuration of the http transport:

```

<parameter name="proxyProfiles">
    <profile>
        <targetHosts>example.com, *.sample.com</targetHosts>
        <proxyHost>localhost</proxyHost>
        <proxyPort>3128</proxyPort>
        <proxyUserName>squidUser</proxyUserName>
        <proxyPassword>password</proxyPassword>
        <bypass>xxx.sample.com</bypass>
    </profile>
    <profile>
        <targetHosts>localhost</targetHosts>
        <proxyHost>localhost</proxyHost>
        <proxyPort>7443</proxyPort>
    </profile>
    <profile>
        <targetHosts>*</targetHosts>
        <proxyHost>localhost</proxyHost>
        <proxyPort>7443</proxyPort>
        <bypass>test.com, direct.com</bypass>
    </profile>
</parameter>

```

When you configure a proxy profile, following are details of the parameters that you need to define in a `<profile>`:

Parameter	Description	Required
targetHosts	A host name or a comma separated list of host names for a target endpoint. Host names can be specified as regular expressions that match a pattern. When <code>targetHosts</code> is specified as an asterisks (*), it will match all the hosts in the profile	Yes
proxyHost	The host name of the proxy server.	Yes
proxyPort	The port number of the proxy server.	Yes
proxyUserName	The user name for the proxy server authentication.	No
proxyPassword	The password for the proxy server authentication.	No
bypass	<p>A host name or a comma separated list of host names that should not be sent via the proxy server.</p> <p>For example, if you want all requests to <code>*.sample.com</code> to be sent via a proxy server, but need to directly send requests to <code>hello.sample.com</code>, instead of going through the proxy server, you can add <code>hello.sample.com</code> as a bypass host name.</p> <p>You can specify host names as regular expressions that match a pattern.</p>	No

Viewing the Handlers in Message Flows

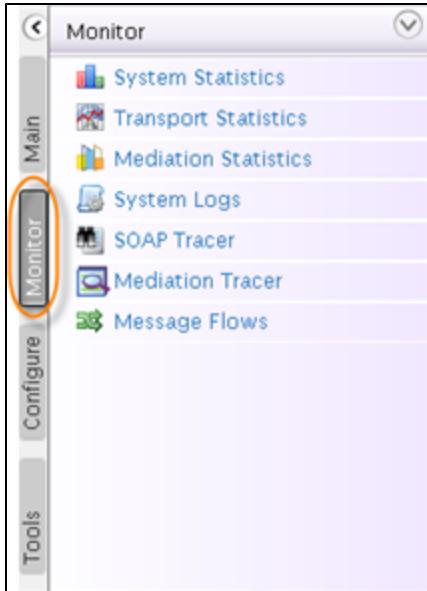
Message flows provide graphical and textual views of the globally engaged handlers of the system at any point of time. The [modules](#) use the handlers to engage in different message flows at defined phases. There are four different message flows defined in the system (explained in the following section). You can observe the handlers invoked in each phase of each flow in real time.

For example, the "Apache/Rampart" module (the security module of the system) defines the handlers in the security

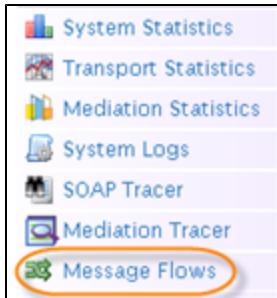
phase of each flow, which handles the security aspects of the messages that are transferred through these flows. So if the "Rampart" module is engaged, you will see the "Apache/Rampart" handlers in the message flows in real time.

Follow the instructions below to access the Message Flows.

1. Sign in. Enter your user name and password to log on to the ESB Management Console.
2. Click on "Monitor" on the left side to access the "Monitor" menu.



3. In the "Monitor" menu, click on "Message Flows."



4. The "Message Flows" page with the graphical view of the message flows appears. There are four different flows defined in the system:

- **In Flow** - A correct message coming into the system.
- **Out Flow** - A correct message going out of the system.
- **In Fault Flow** - A faulty message coming into the system.
- **Out Fault Flow** - A faulty message going out of the system.

In each flow, a message passes through a set of phases to reach the service. These phases vary according to the currently engaged modules within the system. The interface displays the current phases in each and every flow as shown in the following figure.

Message Flows (Graphical View)

[T Show Text View](#)

In Flow



Out Flow



In Fault Flow



Out Fault Flow



5. In the graphical view of the message flows, click the links to get a view of the engaged handlers in each phase.

For example, the figure below shows the handlers engaged in the Addressing phase at system start up.

In flow : Addressing Phase Handlers



6. You can see the text view of message flows. Click on the "Show Text View" link.

Message Flows (Graphical View)

[T Show Text View](#)

In Flow



7. The page with the text view of message flows appears. The textual view provides the name and the fully qualified classes of all handlers within each and every phase.

Message Flows (Text View)

Show Graphical View

In Flow

MsgInObservation

InOnlyMEPHandler	org.wso2.carbon.statistics.module.InOnlyMEPHandler
<hr/>	
Transport	
RequestURIBasedDispatcher	org.apache.axis2.dispatchers.RequestURIBasedDispatcher
SOAPActionBasedDispatcher	org.apache.axis2.dispatchers.SOAPActionBasedDispatcher
REST/POX Security handler	org.wso2.carbon.security.pox.POXSecurityHandler
<hr/>	
Addressing	
AddressingInHandler	org.apache.axis2.handlers.addressing.AddressingInHandler
AddressingBasedDispatcher	org.apache.axis2.dispatchers.AddressingBasedDispatcher

Enabling SSL Tunneling through a Proxy Server

If your proxy service connects to a back-end server through a proxy server, you can enable secure socket layer (SSL) tunneling through the proxy server, which prevents any intermediary proxy services from interfering with the communication. SSL tunneling is available when your proxy service uses the [HTTP PassThrough transport](#) or the [HTTP-NIO transport](#).

To set up Squid:

1. Install Squid as described [here](#).
2. Add the following lines in <SQUID_HOME>/etc/squid3/squid.conf file.

```

acl SSL_ports port 443 8443 8448 8248 8280
acl Safe_ports port 80 # http
acl Safe_ports port 21 # ftp
acl Safe_ports port 443 # https
acl Safe_ports port 70 # gopher
acl Safe_ports port 210 # wais
acl Safe_ports port 1025-65535 # unregistered ports
acl Safe_ports port 280 # http-mgmt
acl Safe_ports port 488 # gss-http
acl Safe_ports port 591 # filemaker
acl Safe_ports port 777 # multiling http
acl CONNECT method CONNECT

auth_param basic program /usr/lib/squid3/basic_ncsa_auth /etc/squid3/basic_pw
auth_param basic children 5
auth_param basic realm Squid proxy-caching web server
auth_param basic credentialssttl 2 hours
auth_param basic casesensitive off

acl ncsa_users proxy_auth REQUIRED
http_access allow ncsa_users

http_port 3128

```

To configure SSL tunneling through the proxy server:

1. In <ESB_HOME>/repository/conf/axis2/axis2.xml, add the following parameters to the transport Sender configuration for PassThroughHttpSender, PassThroughHttpSSLSSender, HttpCoreNIOSSender, and HttpCoreNIOSSLSSender:
 - <parameter name="http.proxyHost" locked="false">hostName</parameter>
 - <parameter name="http.proxyPort" locked="false">portNumber</parameter>
 where *hostName* and *portNumber* specify the host name and port number of the proxy server.
2. Uncomment the following parameter in the PassThroughHttpSSLSSender and HttpCoreNIOSSLSSender configurations and change the value to AllowAll.

```
<parameter name="HostnameVerifier">AllowAll</parameter>
```

For example, if the host name and port number of proxy server is localhost:8080, your transportSender configurations for PassThroughHttPSender and PassThroughHttpSSLSSender would look like this:

PassThroughHTTPSSender

```
<transportSender name="http"
class="org.apache.synapse.transport.passthru.PassThroughHttpSender">
  <parameter name="non-blocking" locked="false">true</parameter>
  <parameter name="http.proxyHost" locked="false">localhost</parameter>
  <parameter name="http.proxyPort" locked="false">8080</parameter>
</transportSender>
```

PassThroughHttpSSLSSender

```
<transportSender name="https"
class="org.apache.synapse.transport.passthru.PassThroughHttpSSLSSender">
  <parameter name="non-blocking" locked="false">true</parameter>
  <parameter name="keystore" locked="false">
    <KeyStore>
      <Location>repository/resources/security/wso2carbon.jks</Location>
      <Type>JKS</Type>
      <Password>wso2carbon</Password>
      <KeyPassword>wso2carbon</KeyPassword>
    </KeyStore>
  </parameter>
  <parameter name="truststore" locked="false">
    <TrustStore>
      <Location>repository/resources/security/client-truststore.jks</Location>
      <Type>JKS</Type>
      <Password>wso2carbon</Password>
    </TrustStore>
  </parameter>
  <parameter name="http.proxyHost" locked="false">localhost</parameter>
  <parameter name="http.proxyPort" locked="false">8080</parameter>
  <parameter name="HostnameVerifier">AllowAll</parameter>
</transportSender>
```

Governing External References Across Environments

Content is under review!

Some artifacts must change based on the environment where the application is deployed. For example, when you deploy an ESB application to Dev, QA, and Production, the service endpoints are different in each of those environments, so you must update the proxy services accordingly. This page provides details on managing environment-specific artifacts across different environments. It focuses specifically on the management of endpoints that are used as external references from within a composite application deployed in different environments.

Services are used from an application. This application is then deployed across various environments: Dev, QA, Integration, etc. For each environment, the service endpoint value is changing and the ESB proxy reference needs to change accordingly.

To attain a comprehensive understanding of this section, please view the [VM image from WSO2Con](#). The VM image provides the pre-configured deployment setup (i.e., the configured ESB and AS servers for Dev and Test environments along with DevStudio for creating the artifacts). This enables you to easily create and deploy the projects on the relevant servers.

Certain sections of this page include detailed steps on setting up the required environment, deploying artifacts using WSO2 Developer Studio, etc. However, in the VM image, these are already setup for your convenience. See [here](#) for more information on how to setup the VM.

The following are the topics related to governing external references across environments. These topics are aimed at aiding you to do this from scratch:

- [Understanding the Users](#)
- [Working with the Server Registry](#)
- [Design Assumptions](#)
- [Best Practices for Migration](#)
- [Sample Scenario](#)
- [Building and Deploying Applications in Dev](#)
- [Testing the Service](#)
- [Editing Resources for QA](#)
- [Updating the Endpoint URI Value](#)

Understanding the Users

Users interacting with artifacts in each environment often have different roles and have access to different resources and tools. For example:

- **Developer:** uses Developer Studio (or Eclipse) to create services and composite applications (CApps) and to push project artifacts to a source code repository, such as Subversion (SVN). Typically, the developer has no access to QA or Production resources.
- **DevOps or Operations team member:** uses scripts and the WSO2 Management Console to pull the applications created by the developers from the source code repository and deploy them to the QA and Production environments. These users need to update the endpoints before they deploy in the different environments. Typically, they do not have Developer Studio.

Working with the Server Registry

All WSO2 servers contain an embedded registry, which runs in-process. This registry store the server configurations, as well as artifacts such as endpoints and data sources. The registry is split into three sections: local, governance, and configuration.

- **local:** used to store configurations and runtime data that are local to the server. This partition cannot be shared. Local metadata is stored under `_system/local` in the registry file system.
- **configuration:** used to store product-specific configurations. This partition can be shared across multiple instances of the same product, which is useful for replicating server configurations. Configuration metadata is stored under `_system/config` in the registry file system.
- **governance:** used to store configurations and data that is shared across the whole platform. This typically includes services, service description, endpoints, and data sources. Governance metadata is stored under `_system/governance` in the registry file system.

Mounting to External Registries

While the local section is always local to the server and therefore stored in the embedded registry, you can configure the governance and

configuration sections to point to external registries (an approach known as **mounting**) instead of using the embedded registry. This is particularly useful if the same configuration or governance assets must be shared across a group of server instances. When a registry is shared across multiple WSO2 servers, the servers must be configured to mount their remote registry on the locations listed above. For example:

```
<mount path="/_system/governance" overwrite="true">
    <instanceId>sharedRegistryInstance</instanceId>
    <targetPath>/_system/governance/branches/qa</targetPath>
</mount>
```

The target path in the mounting setup above must match the location of the external references. For a tutorial on registry mounting options, see [here](#).

You can browse the contents of the registry from any WSO2 server administration console as shown below:

The screenshot shows the WSO2 Enterprise Service Bus administration interface. On the left, there's a vertical sidebar with tabs for Main (selected), Monitor, Configure, and Tools. Under Main, there are links for Home, Manage, Registry, Browse (which is selected and highlighted in blue), and Search. The main content area has a header "Home > Registry > Browse" and a title "Browse". It shows a tree view of the registry structure under the "Root" node. The tree includes nodes for "/", "_system", "config", "governance", and "local". A "Location:" input field with a value of "/" and a "Go" button are also visible.

Referencing Registry Entries

When an application needs to reference an entry in the registry, it uses the naming convention `gov:`, `conf:`, and `local:` to reference the three registry spaces.

Design Assumptions

This page is based on the following design assumptions:

- Definitions of external references are stored in the governance space of the registry.
- External references are stored in the location:
 - `_system/governance/trunk` for the dev environment,
 - `_system/governance/branches/<env_name>` for other environments, such as `_system/governance/branches/qa`.
- When a registry is shared across multiple WSO2 servers, the servers must be configured to mount their remote registry on the locations listed above, e.g.:

```
<mount path="/_system/governance" overwrite="true">
<instanceId>sharedRegistryInstance</instanceId>
<targetPath>/_system/governance/trunk</targetPath>
</mount>
```

The target path in the mounting setup above must match the location of the external references. For more information about registry mounting options, see the following tutorials:

[Sharing Registry Space Across Multiple Product Instances Online Tutorial](#) and [Sharing Registry Space across Multiple Product Instances](#)

Best Practices for Migration

The following are the best practices that allow you to easily migrate applications across environments:

- Whenever you need to reference an endpoint, create it in the registry. This registry can either be embedded or stand-alone, as explained above. This approach ensures that the application can be deployed unchanged from one environment to another.

Do not directly reference an endpoint or any external reference inside your application.

- Reference endpoints from client applications using a key pointing to the governance registry (for example: go v:/public/endpoints/HelloWorldService). This value is constant across all environments.
- Ensure the endpoint definitions are present and accurate in all environments prior to deploying an application using those endpoints. You can either manually edit the endpoint definitions in the registry prior to deploying the application, or make this an automatic part of your deployment process.

Sample Scenario

The rest of this page walks you through a sample scenario of governing external references across environments. It makes the following assumptions:

- A simple HelloWorld service is created and hosted in the Application Server, in both the Dev and QA environments.
- The respective endpoints URLs for the services are:
 - `http://10.200.3.92:9764/services/t/dev-as.com/HelloService.HelloServiceHttpSoap12Endpoint/`
 - `http://10.200.3.92:9764/services/t/qa-as.com/HelloService.HelloServiceHttpsSoap12Endpoint/`
- This HelloWorld service is proxied through the ESB.
- The HelloWorld service endpoint URL varies from Dev to QA and therefore must be changed when the proxy service is deployed from Dev to QA.
- The external references are stored in the governance space in `_system/governance/trunk` for the dev environment and `system/governance/branches/qa` for the QA environment.
- WSO2 Developer Studio 2.x is used to create the composite application containing WSO2 server artifacts.
- WSO2 Governance Registry is used as the central registry/repository for services and server configurations.
- WSO2 Application Server is hosting the HelloWorld service.
- WSO2 Enterprise Service Bus is hosting the proxy service.
- SVN is used as repository and version control of the CApp projects.

Deployment Setup

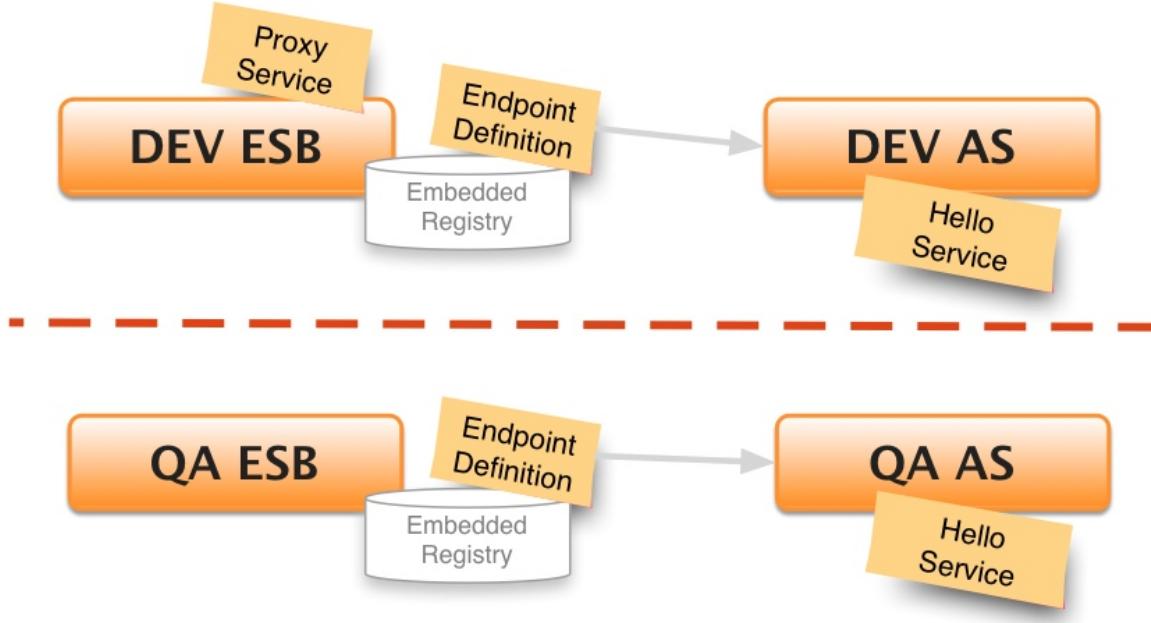
In terms of deployment, you have several choices available. You may wish to rely on the embedded registry only, or share a registry instance across several environments.

One Registry Per Environment

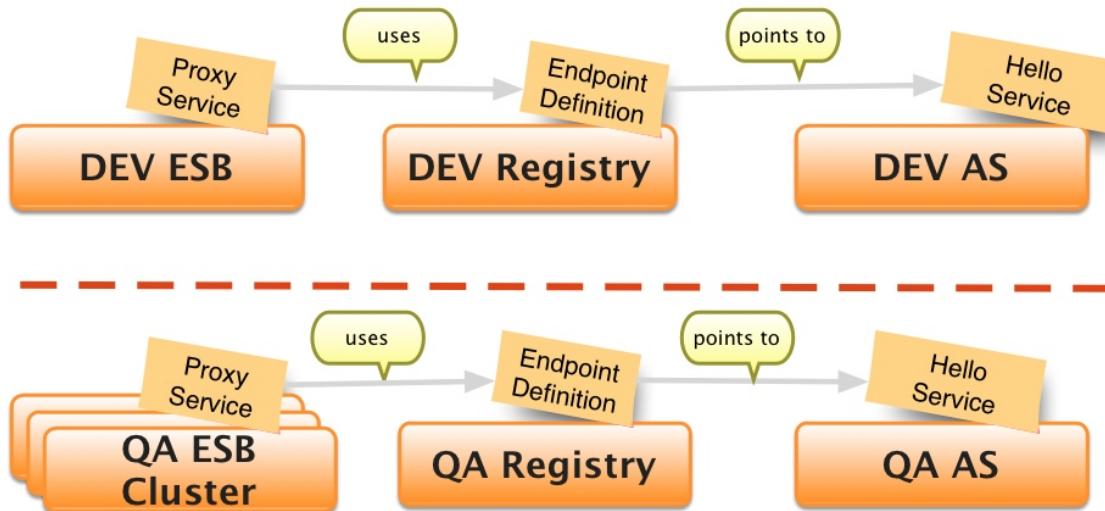
One registry per environment is the default setup if you use the embedded registry as depicted in the first diagram below or if you decide to assign one registry instance per environment as depicted in the second diagram. In this

setup, all external references can be saved in the same registry branch, such as `_system/governance/trunk/references`, because there is no risk of overlap between the various environments.

One Embedded Registry Per Environment:

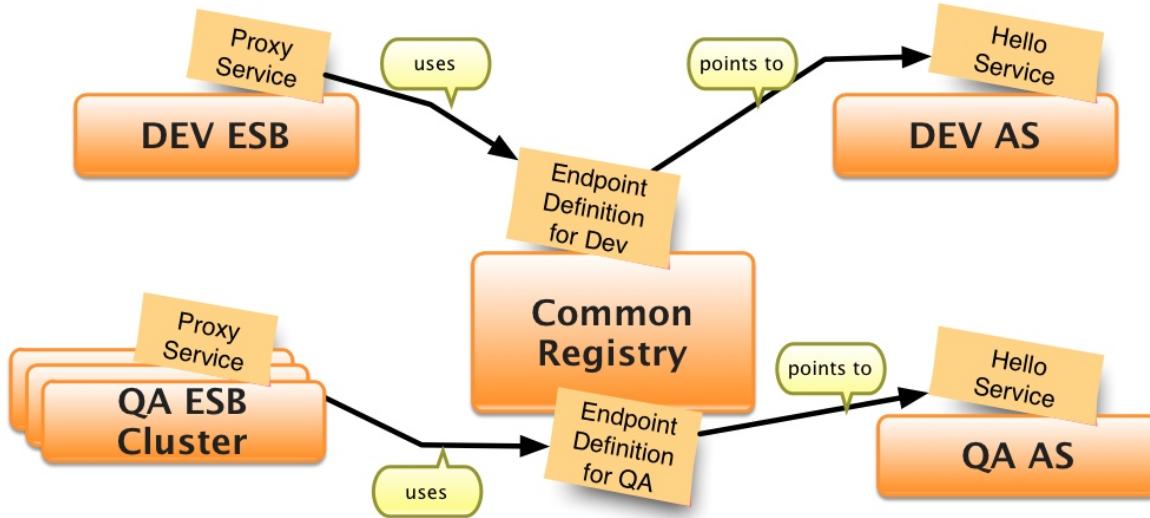


One Standalone Registry Per Environment:



Shared Registry Instance Across Multiple Environments

Another option is to share a registry instance among multiple environments, as depicted below. In such a setup, you must store the external references in separate registry branches, such as `_system/governance/trunk/references` and `_system/governance/branches/qa/references`. Please see [here](#) for more details.



Packaging and Deploying the Application

The application and the definition of the endpoints for all environments are packaged together into a single Maven project. At a high level, the setup is similar to this:

1. A multi-module Maven project is used to contain all the project information. For more information on Maven multi-model projects, see <http://www.sonatype.com/books/mvnex-book/reference/multimodule.html>.
2. Inside the resources folder, one can find the resources for each of the various environments.
 - a. In the basic setup, environment specific files are kept in the same project, under separate folders.
 - b. An alternative to this setup is to create one resources project per environment. In such a setup, resources for each environment can be accessed/controlled separately.
 - c. You can also mix and match and use the same resources project for non-production resources and a separate one for production ones.
3. Independently from how you are storing resources as explained above, we recommend you create one Composite Application (CApp) for each deployment environment, namely HelloWorldResourcesDev and HelloWorldResourcesQA. This allows you to deploy and manage them separately. Additionally, you need to define a composite application for the application itself, i.e., the proxy services. Each composite application is a Maven module that can be built and deployed separately.

The following is the folder structure used:

```

HelloWorldApp/
    HelloWorldDynamicResources/
        pom.xml
        Development/
            endpoints/HelloWorld/HelloWorldServiceEP.xml
            wsdl/HelloWorld/HelloWorldServiceWSDL.xml
        policies/HelloWorld/Policy.xml
        QA/
            endpoints/HelloWorld/HelloWorldServiceEP.xml
            wsdl/HelloWorld/HelloWorldServiceWSDL.xml

HelloWorldStaticResources/
    pom.xml
    policies/HelloWorld/Policy.xml
HelloWorldProxyServices/
    pom.xml
    src/ /* src for the HelloWorld ESB proxy service */
HelloWorldCApp/
    pom.xml /* maven distribution project aggregating
HelloWorldStaticResources and ProxyService and creates
HelloWorld.car */
HelloWorldResourcesDevCApp/
    pom.xml /* maven distribution project aggregating Resources
and creates HelloWorldResourcesDev.car */
HelloWorldResourcesQACApp/
    pom.xml /* maven distribution project aggregating Resources
and creates HelloWorldResourcesQA.car */

```

Deployment Setup

The deployment setup is as follows:

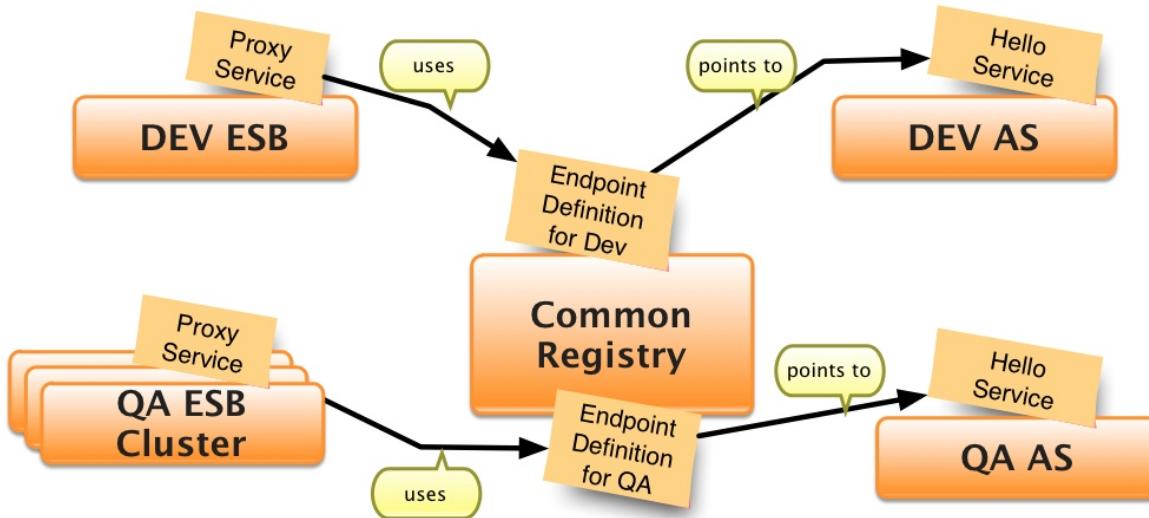
1. The HelloWorld Service has been deployed in two different WSO2 App Server tenants, to simulate the Dev and QA versions of the same service.
2. External reference definitions are stored in the governance space of the registry. The following locations are used to store external references:
 - a. /_system/governance/trunk for the Dev environment
 - b. /_system/governance/branches/qa for the QA environment
3. A WSO2 Governance Registry instance is shared across both ESB servers and has been configured to mount its remote registry on the locations listed above, e.g.:

```

<mount path="/_system/governance" overwrite="true">
    <instanceId>sharedRegistryInstance</instanceId>
    <targetPath>/_system/governance/trunk</targetPath>
</mount>

```

The target path in the mounting setup above must match the location of external references.



4. WSO2 Developer Studio 3.0.0 is used to create the composite application (CApp) containing WSO2 server artifacts.

Samples

The following samples are described in their respective sections:

1. Use a [Pass-through Proxy Service in ESB](#) to expose services hosted on AS. Services as well as proxy services are deployed across Dev and QA environments. The endpoint value keeps changing for these environments and the ESB proxy service references need to change accordingly.
2. Use a [Secure Proxy Service to Expose Services](#) hosted on AS. The endpoint values and the policies are stored in the registry and referenced by the proxy service. The endpoint values keep changing for Dev and QA environments and the ESB proxy service reference needs to change accordingly.

Building and Deploying Applications in Dev

Starting the Servers

Open a Terminal and start the following servers:

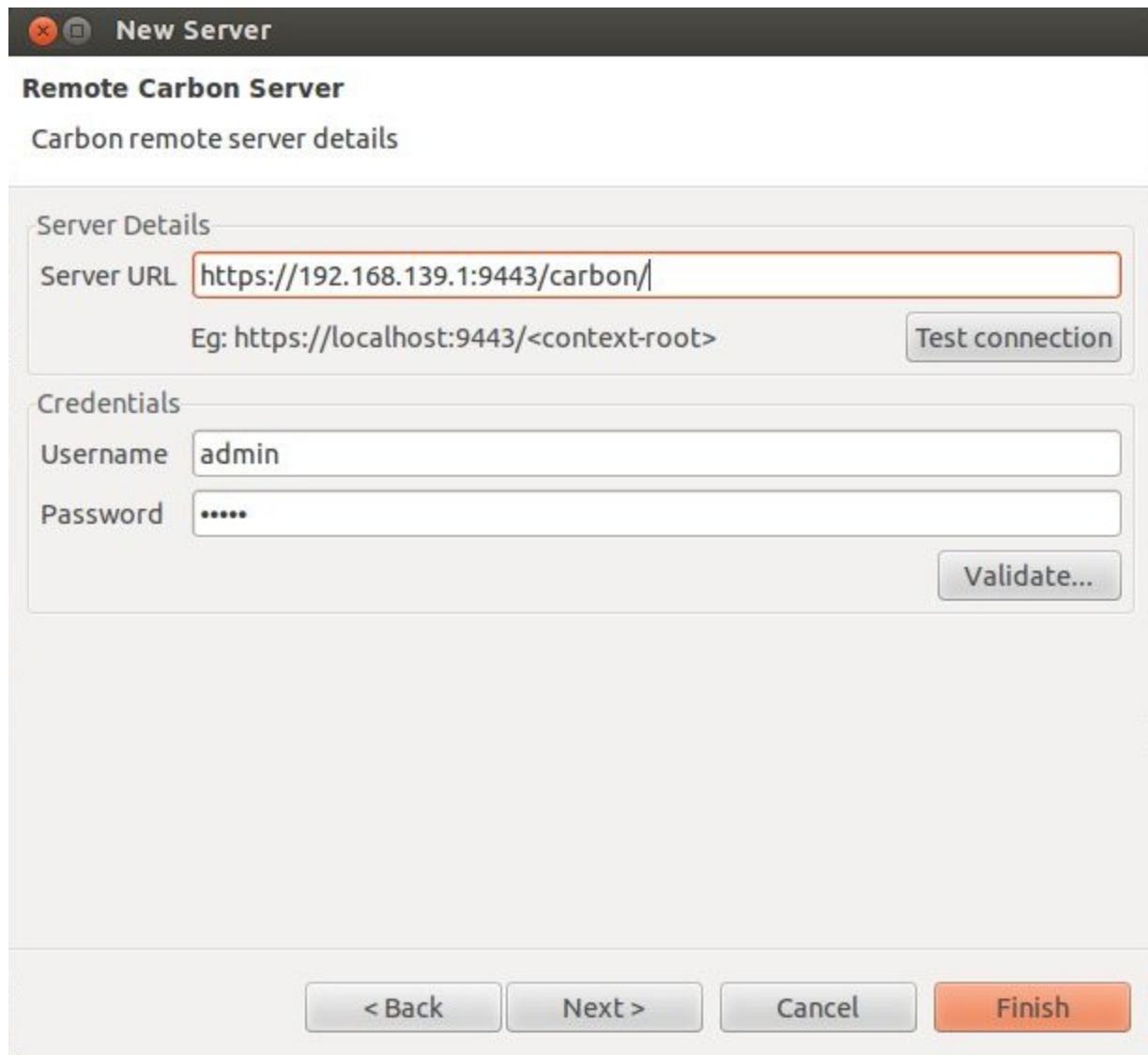
- WSO2 Application Server
- WSO2 Governance Registry
- WSO2 Enterprise Service Bus (relevant server for the Dev environment)

Deploying from Developer Studio

To deploy via Developer Studio, the ESB and G-Reg servers need to be added in Developer Studio.

To add the servers in Developer Studio:

1. Open the Servers view (if you can't see it, go to **Window > Show View**).
2. Right-click on the Server's view window at the bottom of the page and select **New > Server** and select the relevant Carbon server in the WSO2 group.
3. Specify the server name and server URL along with the credentials.
4. The server is then added in Developer Studio.



To deploy the `HelloWorldResourcesDevCApp` to WSO2 Governance Registry server instance, drag and drop the `HelloWorldResourcesDevCApp` onto the Governance Registry server that is listed in the Servers view. The application should be automatically deployed. You can monitor the trace in the Terminal from which you started the server.

You can repeat the above steps to deploy the `HelloWorldResourcesDevCApp` to the Governance Registry and `HelloWorldCApp` to the ESB servers.

Deploying from the Server Admin Console

Building the CAR Files

To deploy via the server admin console or via Maven, you need to first build the CAR file. This can be built using Developer Studio or either using command line, via Maven.

Do not set the media type for WSDLs and WSDL resources. This is done to ensure that the WSDL handler does not intercept the Registry Put operation.

1. To build via Developer Studio: right-click on a CApp project and choose **Export Carbon Application Project**.

2. To build via Maven: once a project has been checked-out from SVN for example, you can navigate to the HelloWorldCApp project and enter: **mvn clean install**.

Deploying the CAR Files

To deploy the CAR file, you can either use the administration console or the Maven WSO2 deploy plugin. From the Administration Console

You can deploy CAR files to any WSO2 server using the administration console. This also applies to the Governance Registry. For example, to deploy the dev resources to the registry, you need to:

1. Open the administration console.
2. Login as administrator.
3. Go to **Manage -> Carbon Applications**.
4. Click on **Add**.
5. Pick the CAR file you want to deploy and upload the file.

The application is deployed asynchronously (i.e., multiple deployment functions can occur at the simultaneously). Refresh the user interface regularly to check whether it has been uploaded successfully.

Using Maven Deploy

You can use the Maven deploy plugin to push CAR files to servers. To use the Maven deploy plugin, you must first define deployment profiles within the project **pom.xml** file. This is described in more detail [here](#).

Note: This plugin is new and upgraded often. Make sure to refresh your Maven repository to ensure that you are using the latest version.

Testing the Service

Testing Sample 1: Pass through Proxy Service

1. Open the Dev ESB Server admin console from the bookmarks by navigating to the admin console URL that is listed in the terminal.
2. Login as an admin.
3. Go to **Services > List**.
4. Click on **Try This Service** for the HelloWorldServiceProxy (which you just deployed via the admin console).

Home > Manage > Services > List

Deployed Services

4 active services. 4 deployed service group(s).

Service Type **ALL**

Service



Select all in this page | Select none

Delete

Services

<input type="checkbox"/>	echo		axis2		Unsecured		WSDL1.1		WSDL2.0		Try this service
<input type="checkbox"/>	HelloWorldServiceProxy		proxy		Unsecured		WSDL1.1		WSDL2.0		Try this service

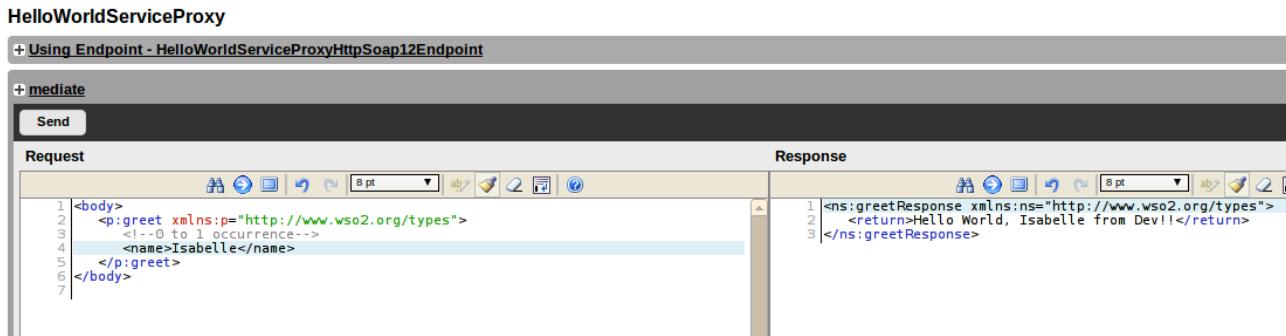
5. In the dialog that appears, use the following request:

```

<body>
  <p:greet xmlns:p="http://www.wso2.org/types">
    <!-- 0 to 1 occurrence-->
    <name>YourNameGoesHere</name>
  </p:greet>
</body>

```

6. Click **Send**. You should see a response from the application server as seen in the image below:



Testing Sample 2: Secured Proxy Service

Secured proxy services (not including user name token) cannot be tested using the tryit tool available in the management console.

For this we need to create a Soap UI project to test the proxy service. Soap UI is shipped by default with WSO2 Developer Studio. Click on **Window > Open Perspective > Other** and select **SoapUI** from the list in the pop-up window that appears to start using SoapUI.

Details on how to test the proxy service are available at: <http://blog.thilinamb.com/2011/02/invoke-secured-web-service-with.html>

Editing Resources for QA

To deploy in the QA environment, you can modify the URI and generate the CAR file from Developer Studio or from the command line.

Option 1: Using Developer Studio

1. Open the HelloWorldServiceEP.xml file under HelloWorldResources/QA and replace the URL with the QA URL:
2. Right-click on the HelloWorldResourcesQACApp project and choose **Export Carbon Application Project**.
3. Provide a location (e.g. /home/wso2user/Desktop) for the output file (the file name is composed of the artifactID combined with the version number).
4. Follow steps 1 and 2 above for editing the WSDL file for the QA environment.

Option 2: From Command Line

1. Open a Terminal window and locate the HelloWorldServiceEP.xml file from the Eclipse workspace.
 2. Edit the HelloWorldServiceEP.xml (e.g. using gedit or vi) under HelloWorldResources/QA and replace the URL with the QA one.
- ```
... uri="http://192.168.139.1:9765/services/t/test-as.com>HelloService/" ...
```
3. Go to the root of the HelloWorldResourcesQACApp and type mvn clean install (the first time this is run, you need Internet access to download the Maven repository bundles). This creates the CAR file to be deployed to

the Registry.

**Note:** This step requires an active network connection.

4. The resulting CAR file can be deployed directly to the registry as mentioned above.

#### Updating the Endpoint URI Value

Once the endpoint has been created as a registry resource, you can update it in two ways; using the administration console or using the check-in/check-out client. You can also redeploy the CAR file, which overrides the current value.

#### Using the Administration Console

1. To edit the QA endpoint, navigate to the QA registry resource (**Resources > Browse**) in the management console: `/_system/governance/branches/qa/endpoints/HelloWorld>HelloWorldServiceEP.xml`.
2. Click on '**Edit as text**' and modify the contents of the registry resource by replacing the value of the URI.

The screenshot shows the 'Browse' interface of the WSO2 ESB Administration Console. The URL in the address bar is `/_system/governance/branches/qa/endpoints/HelloWorld>HelloWorldServiceEP.xml`. The left sidebar shows navigation paths: Root, /\_system/governance/branches/qa/endpoints/HelloWorld/HelloWorldServiceEP.xml, and /\_system/governance/branches/qa/endpoints/HelloWorld/HelloWorldServiceEP.xml. The main content area is divided into several tabs: **Metadata**, **Properties**, **Content**, **Dependencies**, **Associations**, **Retention**, **Lifecycle**, **Comments**, **Ratings**, and **My Rating:**. The **Content** tab is active, displaying XML code for the endpoint configuration. The XML code includes the endpoint's name, address, and various timeout and retry settings. A red box highlights the `uri` attribute, which is set to `http://10.0.2.15:9766/services/t/qa-as.com>HelloService>HelloServiceHttpSoap12Endpoint/`. Below the XML editor are buttons for **Save Content** and **Cancel**.

3. To edit the QA WSDL, go to the QA registry resource (**Resources > Browse**) in the management console: /\_system/governance/branches/qa/wsdl/Helloworld/HelloworldServiceWSDL.xml
4. Click on '**Edit as text**' and modify the contents of the registry resource by replacing the content with the QA WSDL content.

## Using the Check-in Client

You can use the check-in client to check-in/check-out resources from the governance registry, like this:

```
sh checkin-client.sh checkout
https://localhost:9443/registry/_system/governance/branches/qa/endpoints/Helloworld -u admin -p admin
```

The check-in client only supports check-out of collections (i.e. folders, not of individual resources)

To check the file back in after modifying it, use the following command:

```
sh checkin-client.sh checkin -u admin -p admin -l ./HelloWorldServiceEP.xml
https://localhost:9443/registry/_system/governance/qa/prod/endpoints/Helloworld/HelloworldServiceEP.xml
```

Make sure to use the full path, including the filename when checking-in the file!

**Note:** The check-in client is not available by default with WSO2 servers, only with the WSO2 Governance Registry full product. Please request it from the WSO2 support team if you need to use this tool against an embedded registry.

## Using a Script

Alternatively you can have a CAR file with dummy values for the endpoint URLs and use a customized shell script or batch script. The script created would need to do the following:

1. Extract the CAR file.
2. Edit the URL values.
3. Re-create the CAR file with new values.

## Pass-through Proxy Service with an Endpoint Reference

This section describes how the developer creates the registry resource and proxy configuration projects and submits all projects to a SVN repository.

### To create the project:

1. In Developer Studio, choose **File > New > Project**, expand the **WSO2 > Maven Project** category, and create a Maven Multi Module Project called HelloWorldApp.

#### Maven Information settings



Specify the maven information

Group Id

Artifact Id

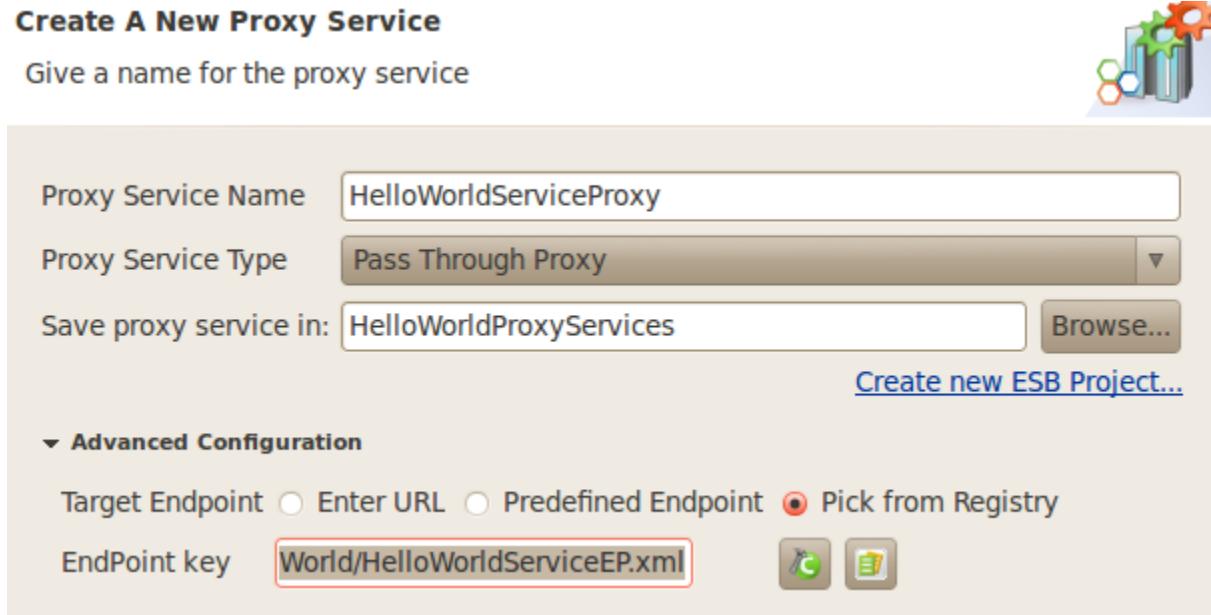
Version

### To create and work with dynamic resources:

1. Select the HelloWorldApp project in the explorer, choose **File > New > Project**, expand the **WSO2 > Repository** category, and create a **Registry Resource Project** called HelloWorldDynamicResources
2. Create one folder inside the HelloWorldDynamicResources project for each environment you need to manage, namely Development and QA.
3. Create a registry resource for the endpoint in the Development environment:
  - a. Right-click the HelloWorldDynamicResources project and choose **New > Registry Resource**.
  - b. Choose **From existing template** and click **Next**.
  - c. Choose the **Address Endpoint Template**.
  - d. Save the resource in the dev folder of the resources project.
  - e. Select '**gov**' as the Registry.
  - f. Specify a registry path where the resource needs to be deployed, i.e. trunk/endpoints/HelloWorld.
  - g. Append the environment to the artifact name (e.g. HelloWorldServiceEP-Dev).
4. Click HelloWorldDynamicResources in the project explorer, choose **File > New > Project**, expand the **WSO2 > Repository** category, and create a **Registry Resource Project** for QA.
5. Create a registry resource for the endpoint as you did in step 4, but this time specify the following settings:
  - a. Name the resource HelloWorldServiceEP-QA (the same resource name as the Dev resource) but append "-QA" to the end of the artifact name.
  - b. Specify the registry path as \_system/governance/branches/qa.
6. Navigate to HelloWorldServiceEP.xml inside the Development resources project folder, double-click the file, click the **Source** tab, and then make it point to HelloWebService in the Dev environment by changing the URL to: `http://10.200.3.92:9764/services/t/dev-as.com/HelloService/` .
7. Navigate to HelloWorldServiceEP.xml inside the QA resources project folder, double-click the file, click the **Source** tab, and then make it point to HelloWebService in the QA environment by changing the URL to: `http://10.200.3.92:9764/services/t/qa-as.com/HelloService/` .
8. Edit the HelloServiceWSDL.xml inside the Development folder and input the contents of the HelloService WSDL in the Dev environment.

### To create and work with static resources:

1. Right-click the HelloWorldApp project in the project explorer, choose **New > Proxy Service**, click **Next**.
2. Select **Create a New Proxy Service** from the options available. The following window appears:



3. Create a **Pass Through Proxy** service using the endpoint reference from the registry.
4. Click on **Create new ESB Project** and set the name as HelloWorldProxyServices.
5. In the **Advanced Configuration** section, select **Pick from Registry**.
6. Set the **EndPoint key** to `gov:/endpoints/HelloWorld/HelloWorldServiceEP.xml`.

Note that `gov:` indicates that we are using the governance section of the registry, and the remaining part of the path is what is exposed to the ESB. This URL remains the same across all environments. Therefore, this one proxy service can be used in all environments and it calls the correct endpoint for the current environment.

7. Edit the proxy service to define the publishing WSDL for the service. Select **Design View**.
  - a. Right-click on the proxy service in the graphical editor and select '**Show Properties view**'.
  - b. Scroll down to the WSDL category and select **REGISTRY\_KEY** as the WSDL Type.
  - c. Click on the icon in the WSDL key row and specify the registry path as `gov:wsdl/HelloWorld/Hel`  
`lloWorldServiceWSDL.xml`.

The projects setup is now complete. You now need to create CApp projects for each of the Composite Applications you want to generate. The ESB code must go in its own CApp, and you create a separate CApp for the resources of each environment.

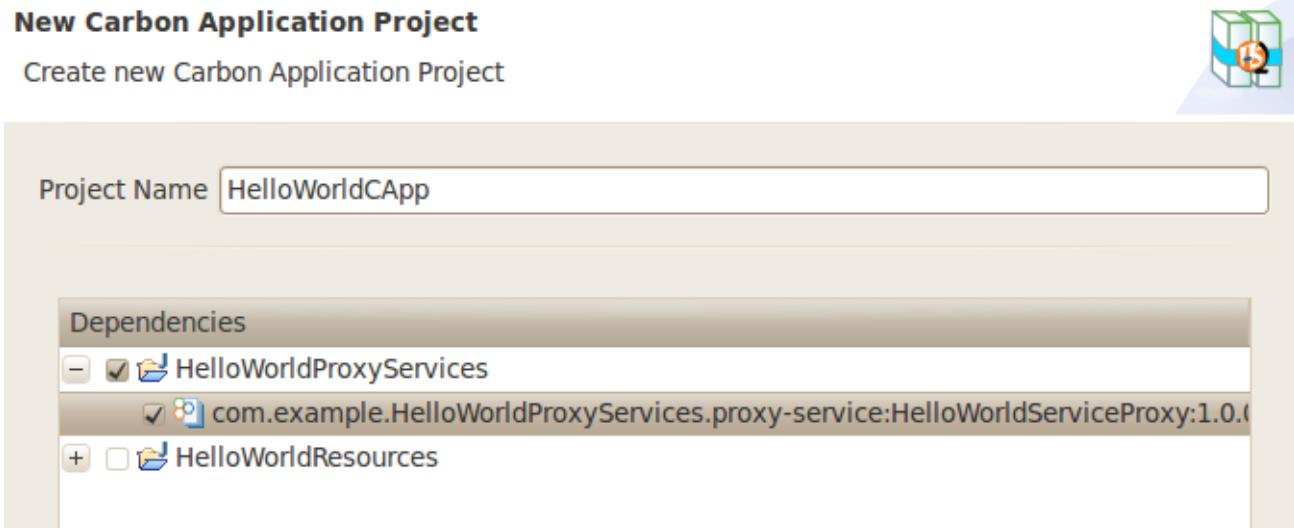
The proxy service needs to go in its own CApp. For resources, you have the choice of:

- Packaging all resources in a single CApp: in this case, when you deploy the application to the registry, all the resources you have defined (Dev and QA in this case) are created/updated in the registry.
- Packaging resources for a single environment in their own CApp - In this case, you need to deploy each application separately.

This choice depends essentially on the amount of control you want over deployment and management of the resources. In the next steps, we show you how to create one CApp per environment.

#### To create a HelloWorldCApp Carbon Application project:

1. Select the HelloWorldApp multi-module project.
2. Choose **File > New > Carbon Application Project** from the main **Eclipse** menu.



3. Name the project HelloWorldCApp.
4. Select HelloWorldProxyServices in the dependencies list.
5. Click **Next**.
6. Set the **artifactID** to HelloWorld (this ensures a `HelloWorld-<versionNumber>.car` file is created).
7. Click **Finish**.
8. Like above, create the HelloWorldResourcesDevCApp Carbon Application and add the HelloWorldResources Project Dev resources to it.

New Carbon Application Project  
Create new Carbon Application Project

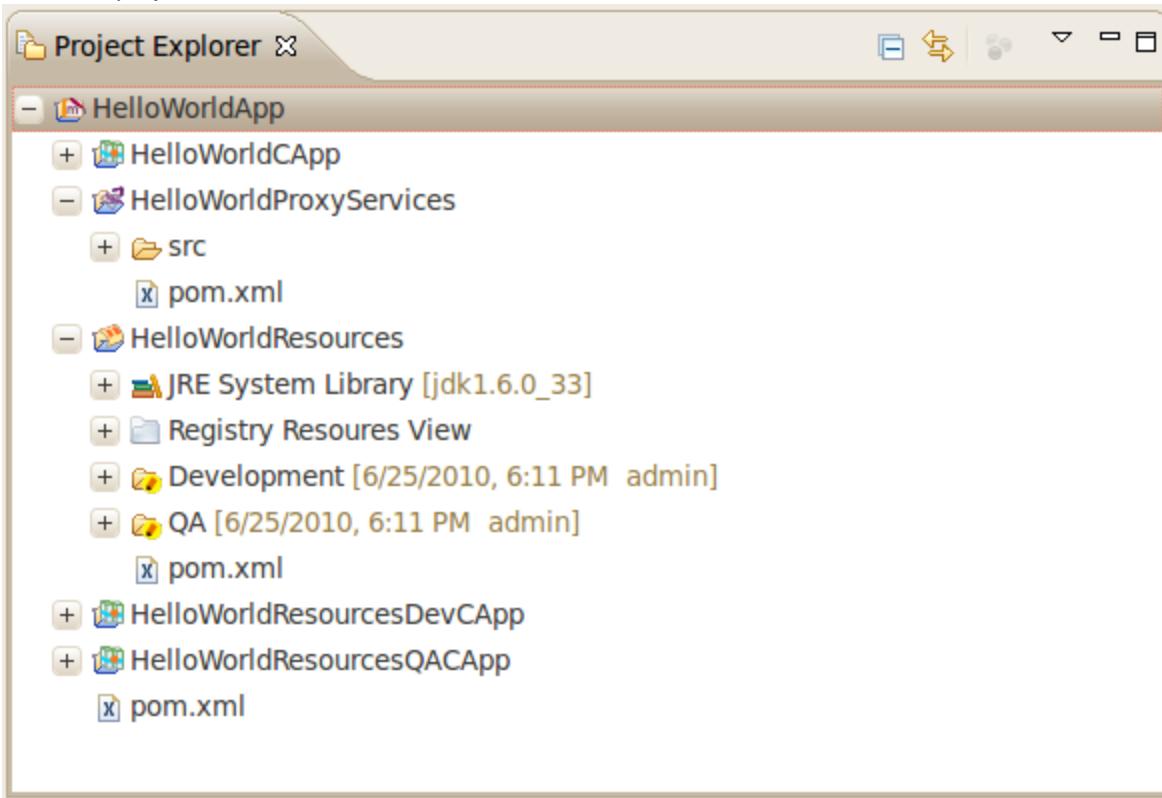


Project Name **HelloWorldResourcesDevCApp**

#### Dependencies

- +  HelloWorldProxyServices
- HelloWorldResources
  - com.example.HelloWorldResources.resource:HelloWorldService-Dev:1.0.0
  - com.example.HelloWorldResources.resource:HelloWorldService-QA:1.0.0

9. Repeat the step above for the QA environment, pointing to the QA resources.
10. Your final project structure should now look like this:



11. Right-click the top project (HelloWorldApp) and invoke Generate POM - Select all the HelloWorld projects, which are then added as modules to the top multi-module Maven project. If you look at the top project pom.xml, you should see the following entries inside:

```
<modules>
 <module>HelloWorldProxyServices</module>
 <module>HelloWorldResources</module>
 <module>HelloWorldCApp</module>
 <module>HelloWorldResourcesDevCApp</module>
 <module>HelloWorldResourcesQACApp</module>
</modules>
```

Your projects are now ready to be built and deployed.

## Secure Proxy Service with a Policy and Endpoint Referenced

In this section we look at a sample on how to create the static security policy resource and the proxy service. These are packed into a single CApp and deployed in the ESB server.

#### Pre-requisites:

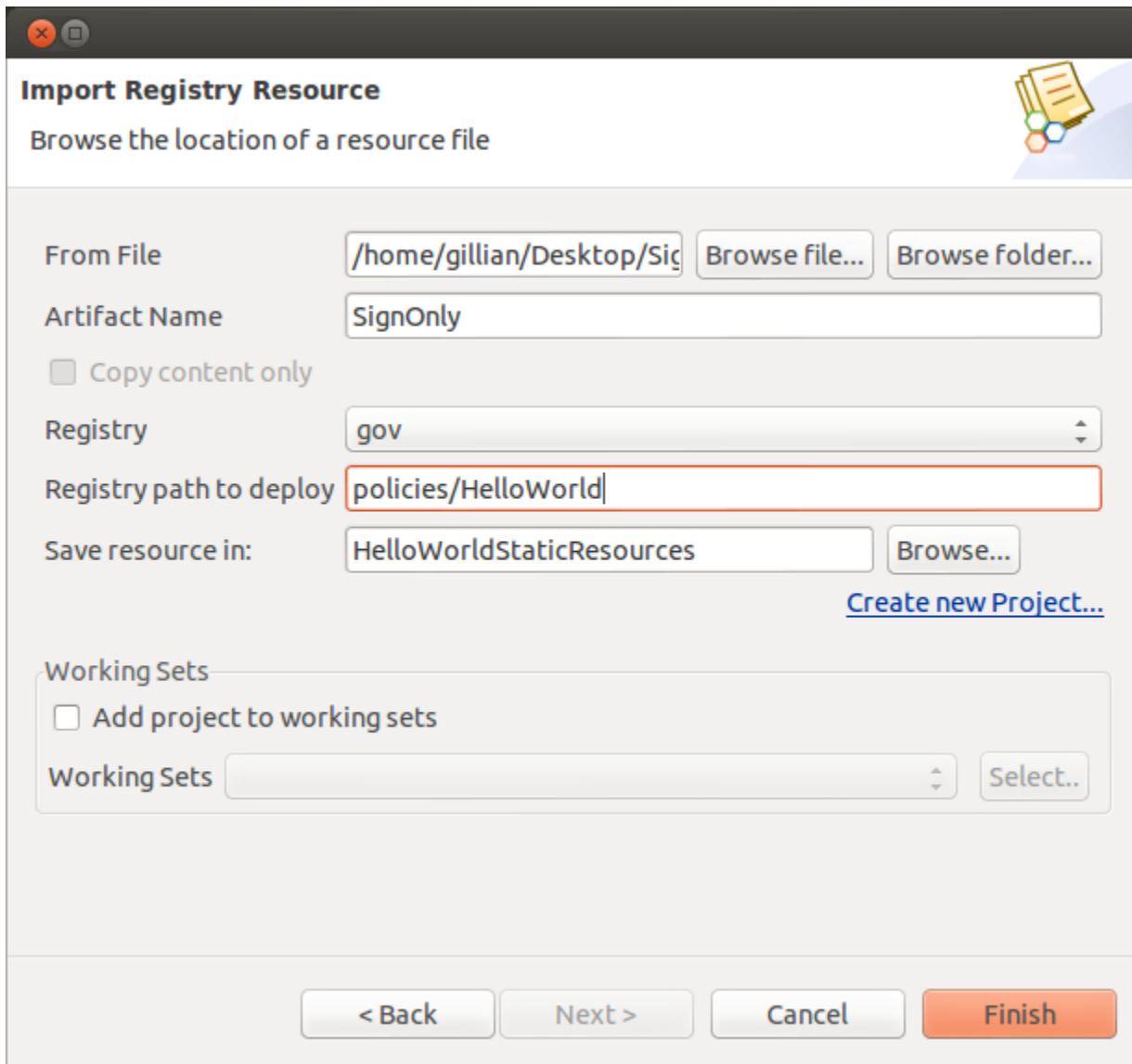
- An XML file containing the security policy for the proxy service needs to be created. Sample content of a simple user name-token security policy is available [here](#).
- Registry Resources with the endpoint URLs to be created as in the previous sample.

#### To create the security policy and proxy service projects:

1. Create an XML file containing the security policy. For this sample, we look at a simple authentication policy using X509 certificate. The XML configuration details are available [here](#).

**Note:** Currently Developer Studio tooling has a limitation where it does not support security policies for user name token authentication.

2. Create a **Registry Resources Project** named 'HelloWorldStaticResources'.
3. To create a new **Registry Resource** for the policy, right-click the project and select **New > Registry Resource**.



4. Choose **Import from file system**.

5. Browse and select the XML file containing the policy from the file system.
6. Select 'gov' as the Registry.
7. Specify 'policies/HelloWorld' as the Registry path to deploy.

**To create an ESB proxy service:**

1. Select the HelloWorldApp project, right-click and select **New > Proxy Service**.
2. Select **Create a New Proxy Service**.
3. Select **Secure Proxy**.
4. Click on **Create new ESB Project** and set the name as HelloWorldProxyServices.
5. In the **Advanced Configuration** section, select **Pick from Registry**.
6. Set the endpoint key to `gov:/endpoints/HelloWorld/HelloWorldServiceEP.xml`.
7. Set the security policy key as `gov:/policies/HelloWorld/SignOnly.xml`.

Note that this indicates that we are using the governance section of the registry, and what is visible from the ESB is the `/policies/HelloWorld/SignOnly.xml` URL. This URL remains the same across all environments.

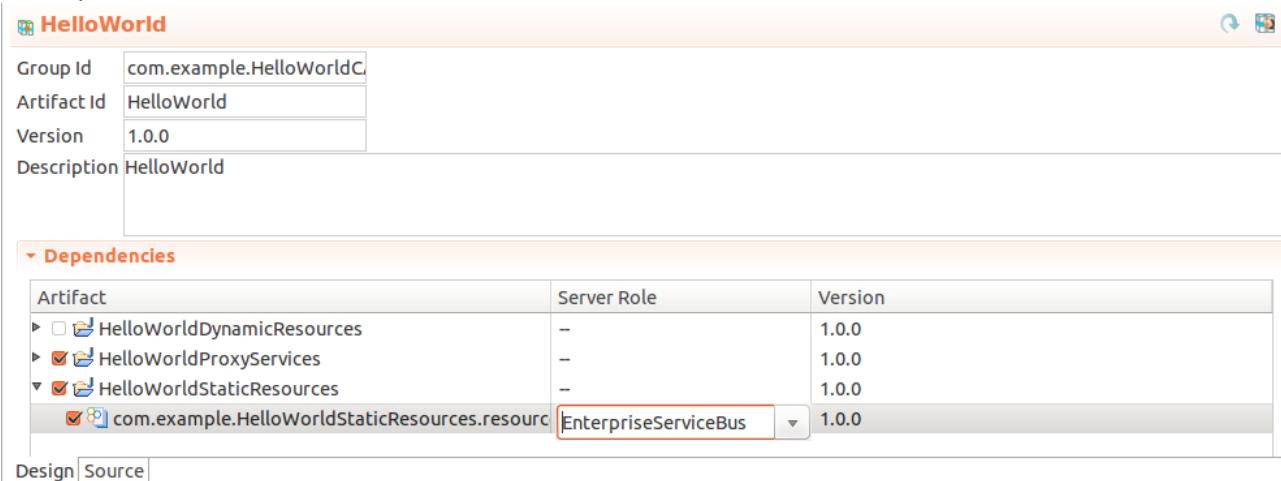
8. The projects setup is now complete.

**To create CApp projects for each CApp archive:**

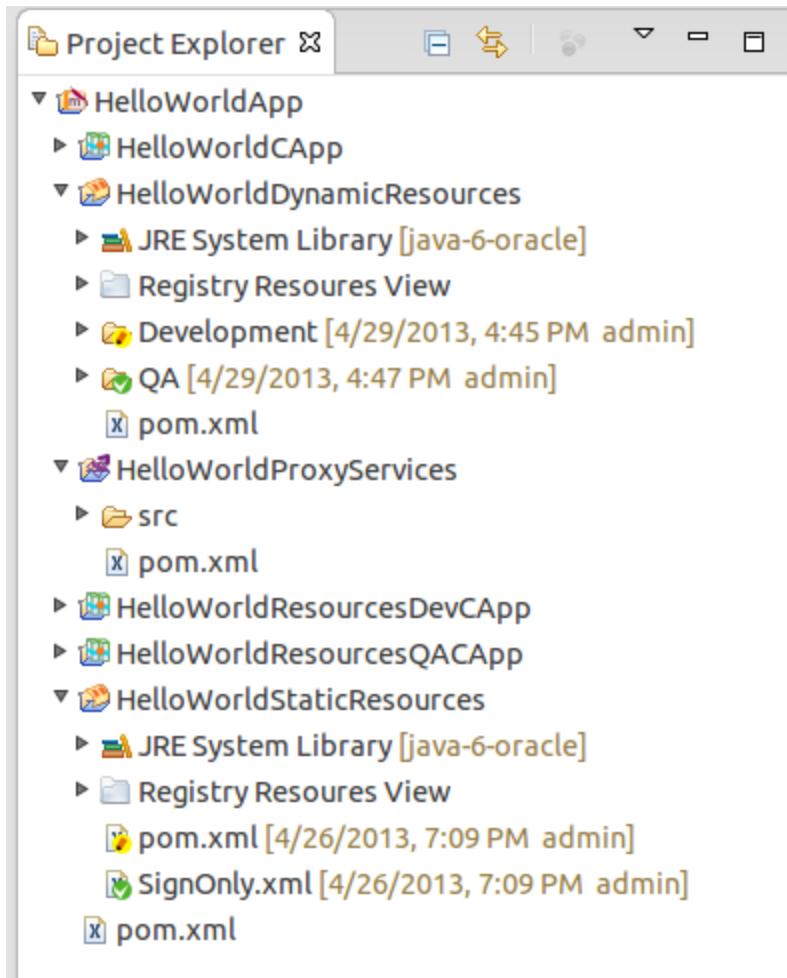
1. Create a resource CApp with the dynamic resources created in the [previous sample](#).
2. Create a CApp with the static resource and proxy service we created in the steps above.
3. The next step is to create the CApp to contain the static resource (security policy) and the proxy service.

**To create and use a HelloWorldCApp Carbon Application project:**

1. Select the HelloWorldApp multi-module project.
2. Choose **File > New > Carbon Application Project** from the main **Eclipse** menu.
3. Name the project HelloWorldCApp.
4. Select the HelloWorldProxyServices and HelloWorldStaticResources in the dependencies list.
5. Click **Next**.
6. Set the **artifactID** to HelloWorld (this ensures a `HelloWorld-<versionNumber>.car` file is created).
7. Click **Finish**.
8. When the CApp is created, expand the CApp in the Project Explorer window and select **pom.xml**.
9. Select HelloWorldStaticResources and expand.
10. Change the server role to ESB since we are deploying this CApp in the ESB. Select **Enterprise Service** from the drop down list.



11. Your final project structure should now look like this:



12. Right-click the top project (HelloWorldApp) and select **Generate POM**. Select all the HelloWorld projects, which are then added as modules to the top multi-module Maven project. If you look at the top project pom.xml, you should see the following entries inside:

```
<modules>
 <module>HelloWorldProxyServices</module>
 <module>HelloWorldDynamicResources</module>
 <module>HelloWorldStaticResources</module>
 <module>HelloWorldCApp</module>
 <module>HelloWorldResourcesDevCApp</module>
 <module>HelloWorldResourcesQACApp</module>
</modules>
```

Your projects are now ready to be built and deployed.

#### Server Profiles and Sample Security Policy

Server profiles can be used to manage deployment information for the Maven deploy plugin. The profiles define the name and server URL of the deployment environment where the project needs to be deployed. A profile needs to be defined for each deployment environment with the following parameters:

Tag	Description
<id>	The unique id used for the profile

<name>	The parameter passed when deploying the project with Maven from the command line
<value>	The value passed with -Denvironment when deploying from the command line. This value defines the server the project needs to be deployed on
<trustStorePath>	The path to the wso2carbon.jks key store
<trustStorePassword>	The password to the Keystore
<serverURL>	URL of the server where the project will be deployed
<username>	Server username
<password>	Server password

To define server profiles, you must edit the project pom.xml file and insert the profile as follows:

```

<?xml version="1.0"?>
<profiles>
 <profile>
 <id>devdefault</id>
 <activation>
 <property>
 <name>environment</name>
 <value>Dev</value>
 </property>
 </activation>
 <build>
 <plugins>
 <plugin>
 <groupId>org.wso2.maven</groupId>
 <artifactId>maven-car-deploy-plugin</artifactId>
 <version>1.0.0</version>
 <extensions>true</extensions>
 <configuration>
 <carbonServers>
 <CarbonServer>

 <trustStorePath>/devesb_home/repository/resources/security/wso2carbon.jks</trustStorePath>
 <trustStorePassword>wso2carbon</trustStorePassword>
 <trustStoreType>JKS</trustStoreType>
 <serverUrl>https://esb_ip_address:port</serverUrl>
 <username>admin</username>
 <password>xxxxxx</password>
 <operation>deploy</operation>
 </CarbonServer>
 </carbonServers>
 </configuration>
 </plugin>
 </plugins>
 </build>
 </profile>
 ...
</profiles>

```

To deploy, use the following commands:

```
maven deploy -Denvironment=Dev (For Dev ESB)
maven deploy -Denvironment=QA (For QA ESB)
```

### Sample Security Policy:

```

<wsp:Policy wsu:Id="SigOnly" xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd"><wsp:ExactlyOne><wsp:All><sp:AsymmetricBinding
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"><wsp:Policy><sp:Initia-
torToken><wsp:Policy><sp:X509Token
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Alw-
aysToRecipient"><wsp:Policy><sp:RequireThumbprintReference/><sp:WssX509V3Token10/></ws-
p:Policy></sp:X509Token></wsp:Policy></sp:InitiatorToken><sp:RecipientToken><wsp:Pol-
icy><sp:X509Token
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Nev-
er"><wsp:Policy><sp:RequireThumbprintReference/><sp:WssX509V3Token10/></wsp:Policy></s-
p:X509Token></wsp:Policy></sp:RecipientToken><sp:AlgorithmSuite
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"><wsp:Policy><sp:Basic2-
56/></wsp:Policy></sp:AlgorithmSuite><sp:Layout><wsp:Policy><sp:Strict/></wsp:Policy><
/sp:Layout><sp:IncludeTimestamp/><sp:OnlySignEntireHeadersAndBody/></wsp:Policy></sp:A-
symmetricBinding><sp:Wss10
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"><sp:Policy><sp:MustSup-
portRefKeyIdentifier/><sp:MustSupportRefIssuerSerial/></sp:Policy></sp:Wss10><sp:Signe-
dParts
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"><sp:Body/></sp:SignedP-
arts></wsp:All></wsp:ExactlyOne><rampart:RampartConfig
xmlns:rampart="http://ws.apache.org/rampart/policy"><rampart:user>wso2carbon</rampart:-
user><rampart:encryptionUser>useReqSigCert</rampart:encryptionUser><rampart:timestampP-
recisionInMilliseconds>true</rampart:timestampPrecisionInMilliseconds><rampart:timesta-
mpTTL>300</rampart:timestampTTL><rampart:timestampMaxSkew>300</rampart:timestampMaxSke-
w><rampart:timestampStrict>false</rampart:timestampStrict><rampart:tokenStoreClass>org-
.wso2.carbon.security.util.SecurityTokenStore</rampart:tokenStoreClass><rampart:nonceL-
ifeTime>300</rampart:nonceLifeTime><rampart:encryptionCrypto><rampart:crypto
provider="org.wso2.carbon.security.util.ServerCrypto"
cryptoKey="org.wso2.carbon.security.crypto.privatestore"><rampart:property
name="org.wso2.carbon.security.crypto.alias">wso2carbon</rampart:property><rampart:pro-
perty
name="org.wso2.carbon.security.crypto.privatestore">wso2carbon.jks</rampart:property><
rampart:property
name="org.wso2.stratos.tenant.id">-1234</rampart:property><rampart:property
name="org.wso2.carbon.security.crypto.truststores">wso2carbon.jks,</rampart:property><
rampart:property
name="rampart.config.user">wso2carbon</rampart:property></rampart:crypto></rampart:enc-
ryptionCrypto><rampart:signatureCrypto><rampart:crypto
provider="org.wso2.carbon.security.util.ServerCrypto"
cryptoKey="org.wso2.carbon.security.crypto.privatestore"><rampart:property
name="org.wso2.carbon.security.crypto.alias">wso2carbon</rampart:property><rampart:pro-
perty
name="org.wso2.carbon.security.crypto.privatestore">wso2carbon.jks</rampart:property><
rampart:property
name="org.wso2.stratos.tenant.id">-1234</rampart:property><rampart:property
name="org.wso2.carbon.security.crypto.truststores">wso2carbon.jks,</rampart:property><
rampart:property
name="rampart.config.user">wso2carbon</rampart:property></rampart:crypto></rampart:sig-
natureCrypto></rampart:RampartConfig></wsp:Policy>
```

## Using the VM Image for Governing External References Across Environments

The [VM image from WSO2Con](#) aides you in developing and deploying a simple ESB application to both Dev and QA environments. The ESB hosts a pass-through proxy to the HelloWorld service which comes packaged as a sample

with the WSO2 Application Server. In order to use this VM, you need to know the following:

### Installing VirtualBox

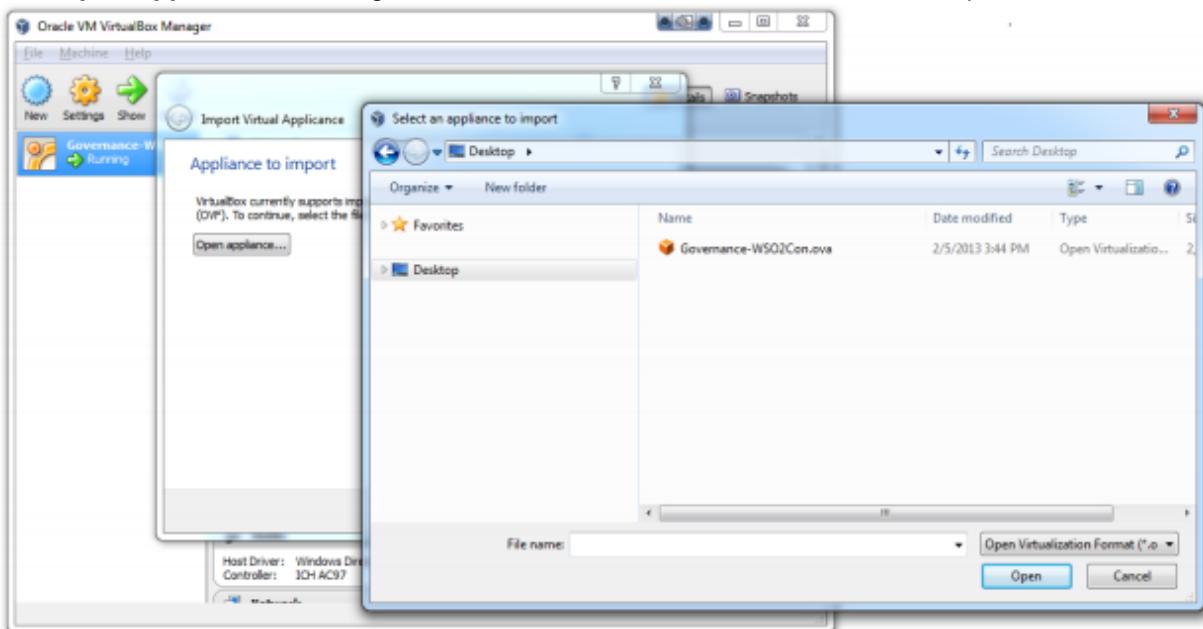
- **Windows:** Run the VirtualBox-4.2.6-82870-Win.exe file
- **Mac:** Run the VirtualBox-4.2.6-82870-OSX.dmg file
- **Linux:** Run the VirtualBox-4.2.6-82870-Linux\_x86 file or VirtualBox-4.2.6-82870-Linux\_amd64.run file

### Setting up the VM

1. Download the VM from [here](#).
2. Start Oracle VM VirtualBox.

See [here](#) for more information on Oracle VM VirtualBox. You can download it [here](#).

3. Import and run the VM.
  - a. Navigate to **File > Import Appliance**.
  - b. Click **Open Appliance** and navigate to the **WSO2Con-Governance.ova** file and open it.



- c. Click **Next** and wait until the VM import completes.
- d. Click on the **Start** button on VirtualBox to boot-up the instance.
- e. Provide the username **wso2user** and password **wso2** to login to the instance.

### Enterprise Integration Patterns

Over the years, architects have invented a blend of integration patterns for connecting various business applications within enterprise systems. Most of these architectures have similarities, initiating a set of widely accepted standards in integration patterns. These standards are described in the **Enterprise Integration Patterns Catalog** available at: <http://www.eaipatterns.com/toc.html>.

You can simulate most of the patterns in the catalog using various constructs of WSO2 ESB. For ideas on how to design your integrations, and to get step-by-step instructions using example scenarios, refer to [Enterprise Integration Patterns with WSO2 ESB](#). You can then refer back to this guide for details on the mediators used in the scenarios, how to work with endpoints and proxy services, and so on.

### WSO2 ESB Tooling

This section describes how you can use the tooling support provided via [WSO2 ESB tooling](#) to create and manage ESB artifacts.

See the following topics for detailed information on how to install WSO2 ESB tooling, how to create ESB artifacts via tooling, and how to package created artifacts into archives that you can deploy to the ESB.

- [Installing WSO2 ESB Tooling](#)
- [Working with ESB Artifacts](#)
- [Packaging your Artifacts into Composite Applications](#)
- [Importing Existing Projects into Workspace](#)

## Installing WSO2 ESB Tooling

WSO2 ESB tooling provides capabilities of a complete Eclipse-based development environment for the ESB. You can develop services, features and artifacts as well as manage their links and dependencies through a simplified graphical editor via WSO2 ESB tooling.

There are 3 possible methods you can follow to install WSO2 ESB tooling.

- [Install WSO2 ESB tooling with pre-packaged Eclipse](#) - This method uses a complete plug-in **installation with pre-packaged Eclipse**, so that you do not have to install Eclipse separately.
- [Install WSO2 ESB tooling on Eclipse Mars using the P2 URL](#)
- [Install WSO2 ESB tooling on Eclipse Mars using the P2 .zip file](#)

The [Install WSO2 ESB tooling on Eclipse Mars using the P2 URL](#) and [Install WSO2 ESB tooling on Eclipse Mars using the P2 .zip file](#) methods require you to **install Eclipse Mars separately** in your system, if you do not have it already

### Install WSO2 ESB tooling with pre-packaged Eclipse

Download the distribution based on your operating system from [here](#), then extract and run the distribution.

If you get an error message about the file being damaged or that you cannot open the file when you try to start Eclipse on a Mac, change the Mac security settings as described below. Alternatively, you can run Eclipse from the command line instead of double-clicking the Eclipse icon.

These instructions are for Mac OS El Capitan. If you are running Mac OS Sierra, the Anywhere security setting is not available, so you must run Eclipse each time from the command line instead of double-clicking the Eclipse icon.

1. Go to **System Preferences**, click **Security & Privacy**, and then click the **General** tab.
2. Under **Allow apps downloaded from**, click **Anywhere**.

Thereafter, you should be able to start Eclipse by double-clicking the Eclipse icon and can set your security settings back to the previous setting.

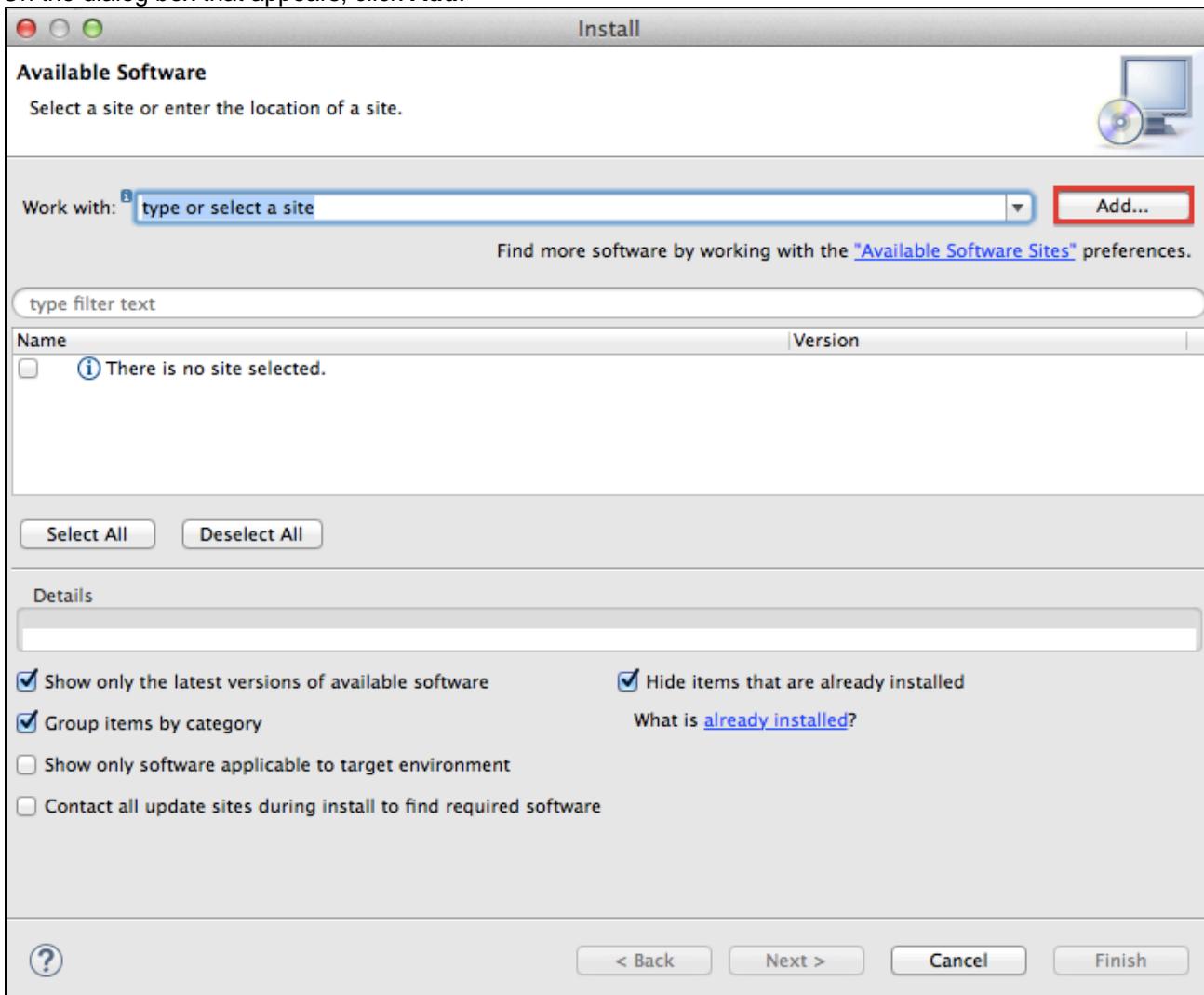
If you are running WSO2 ESB tooling on Ubuntu 16.04, be sure to add the following lines of code in the `ecl ipse.ini` file before the line `--launcher.appendVmargs`, so that you can work with WSO2 ESB tooling smoothly:

```
--launcher.GTK_version
2
```

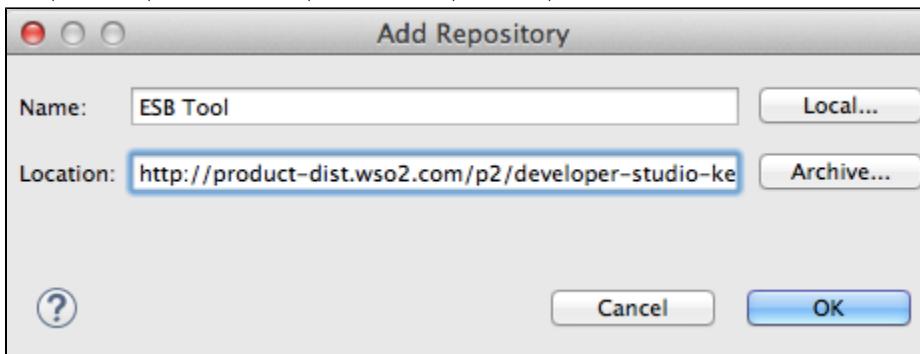
This fix is required due to a known issue with the GTK version shipped with Ubuntu 16.04.

### Install WSO2 ESB tooling on Eclipse Mars using the P2 URL

1. Make sure you have a compatible **Eclipse IDE for Java EE Developers (Mars 2)** distribution installed.
2. Open Eclipse and click **Help > Install New Software**.
3. On the dialog box that appears, click **Add**.



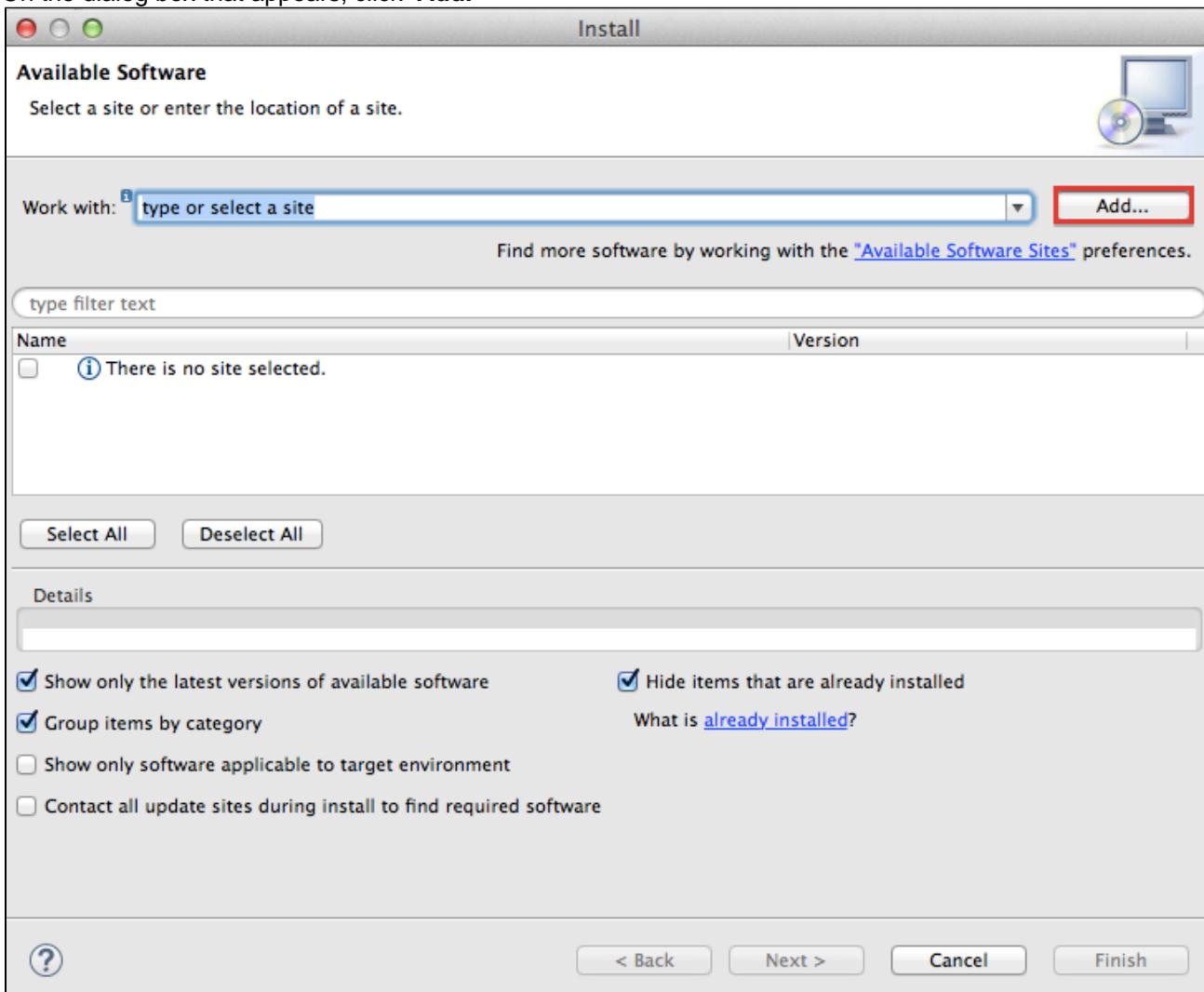
4. Specify **ESB Tool** as the **Name** and **http://product-dist.wso2.com/p2/developer-studio-kernel/4.1.0/esb-tools/releases/5.0.0/** as the **Location** and click **OK**.



5. Select all the check boxes and click **Next**.
6. Read and accept the license agreements and click **Finish**.
7. If a security warning appears saying that the authenticity or validity of the software cannot be established, click **OK**.
8. Restart Eclipse to complete the installation.

## Install WSO2 ESB tooling on Eclipse Mars using the P2.zip file

1. Make sure you have a compatible Eclipse IDE for Java EE Developers (Mars 2) distribution installed.
2. Download the [P2.zip](#) file.
3. Open Eclipse and click **Help > Install New Software**.
4. On the dialog box that appears, click **Add**.



5. Specify ESB Tool as the **Name** and click **Archive**.
6. Navigate to the downloaded .zip file and click **OK**.
7. Select all the check boxes and click **Next**.
8. Read and accept the license agreements and click **Finish**.
9. If a security warning appears saying that the authenticity or validity of the software cannot be established, click **OK**.
10. Restart Eclipse to complete the installation.

## Working with ESB Artifacts

The topics in this section provide information on how you can use [WSO2 ESB Tooling](#) to create an ESB Config project as well as information on how to create various ESB artifacts that you can build and deploy to the ESB in order to process requests. The tooling functionality is available from the Developer Studio dashboard in Eclipse.

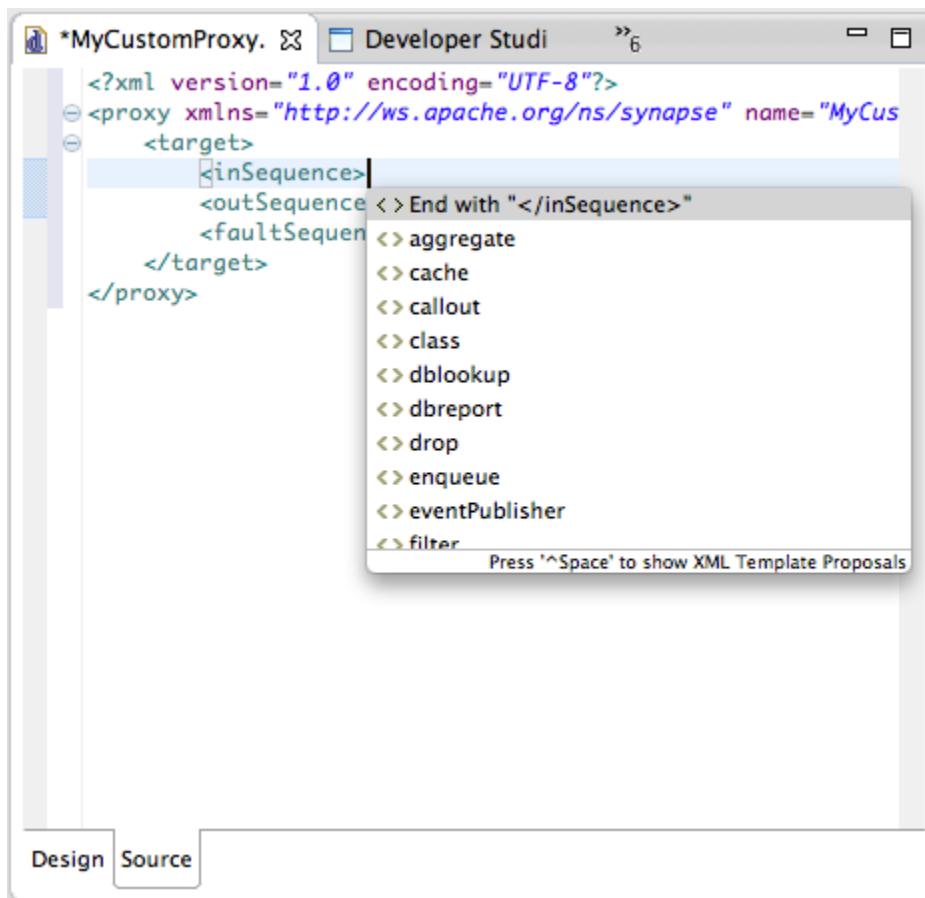
- [Getting Help in the Source View](#)
- [Creating an ESB Config Project](#)
- [Creating an ESB Solution Project](#)
- [Working with Inbound Endpoints](#)

- Working with Endpoints
- Working with Local Registry Entries
- Working with Sequences
- Working with Connectors
- Working with Proxy Services
- Working with APIs
- Working with Tasks
- Working with Message Stores
- Working with Message Processors
- Working with Templates
- Working with Mediators
- Creating a Smooks Configuration Artifact
- Moving ESB artifacts between projects
- Deploying the ESB Config project

#### **Getting help in the source view**

When working with your ESB configurations in the **Source** view instead of the **Design** view, you can hover your mouse over any element (such as the definitions tag) to see the syntax of that element. You can also press **Ctrl + Space** to get suggested auto-completion text.

For example, if you view a proxy service in the **Source** view, and you remove the / from the `<inSequence/>` element and press **Ctrl + space**, the Content Assist appears and lists all the options you can add at this point, including the closing `</inSequence>` element.

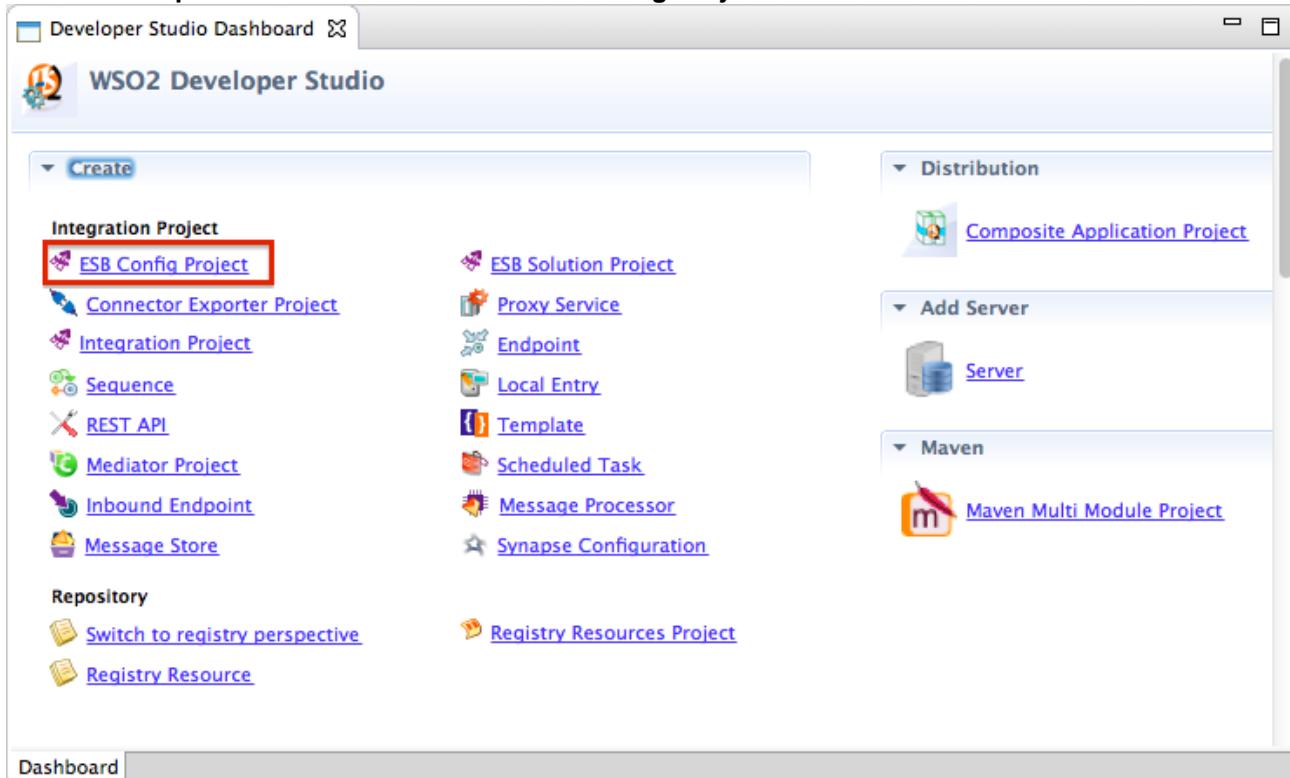


#### **Creating an ESB Config Project**

You can create an ESB Config Project to save all the ESB related artifacts such as proxy services, endpoints, sequences, and synapse configurations.

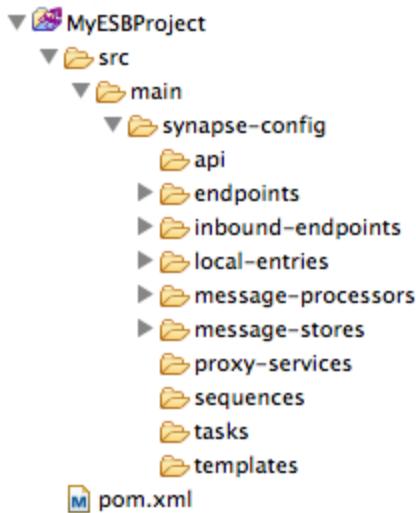
## To create an ESB Config Project:

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. On the **Developer Studio Dashboard** click **ESB Config Project**.



3. If you want to create this ESB project from existing configuration files, select **Point to Existing Synapse-configs Folder**. Otherwise, leave **New ESB Config Project** selected. Click **Next**.
4. Do the following:
  - a. Type a unique name for the project.
  - b. If you selected the option to point to an existing Synapse-configs folder in the previous step, click **Browse** and navigate to the folder containing the configuration files.
  - c. Optionally, specify the location where you want to save the project (or leave the default location specified).
  - d. Optionally, specify the working set, if any, that you want to include in this project.
5. A Maven POM file will be generated automatically for this project. If you want to include parent POM information in the file from another project in this workspace, click **Next**, select the **Specify Parent from Workspace** check box, and then select the required parent project.
6. Click **Finish**.
7. If you specified a folder with existing configuration files, specify whether you want to open those files now.

The new project has now been created in the workspace. If you browse inside the project, you will see a project structure as follows, with folders created for different resources such as endpoints, inbound endpoints, local entries, message processors, message stores, proxy services, and sequences etc.

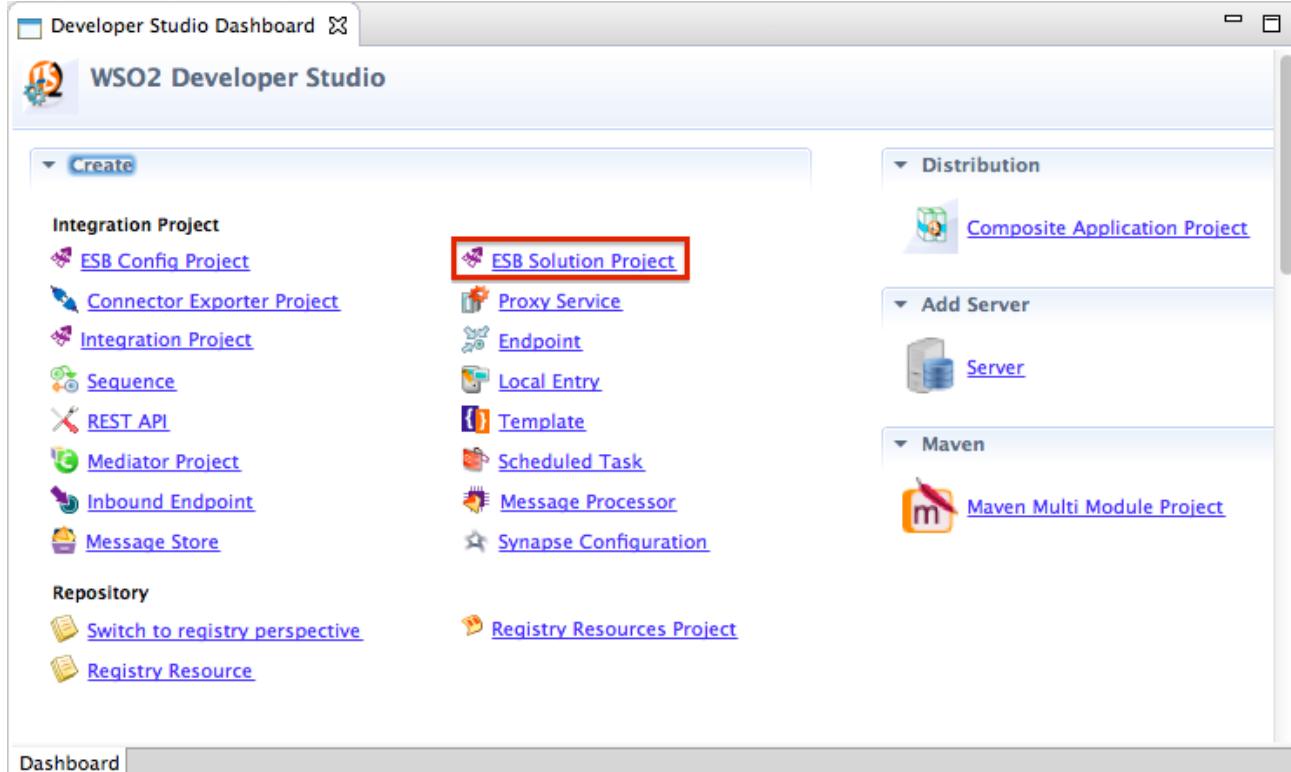


### **Creating an ESB Solution Project**

You can create an ESB Solution Project to save all required project files you need for a particular configuration in one go. For example, if you need to create a project that requires an ESBCConfig project, registry resource project, connector exporter project and composite application project, you can create an ESB Solution Project instead of creating each required project separately.

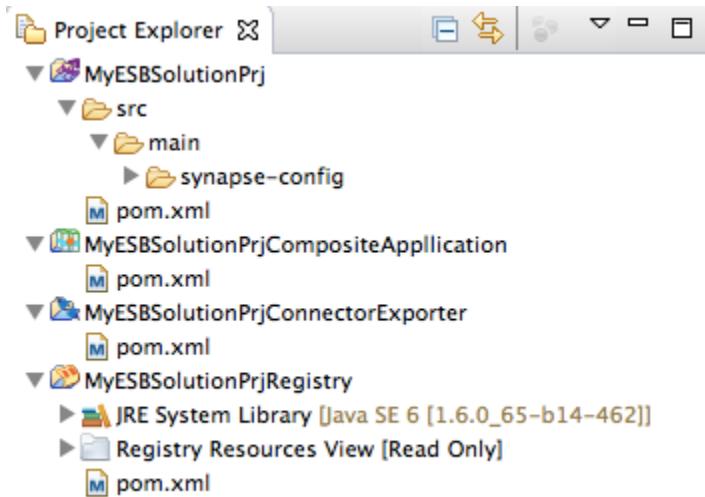
#### **To create an ESB Solution Project:**

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. On the **Developer Studio Dashboard** click **ESB Solution Project**.



3. Do the following:
  - a. Enter a unique name for the project.
  - b. Select the projects you want to create and specify names for each project that you selected (or leave the default project names).

- c. Optionally, specify the location where you want to save the project (or leave the default location specified).
  - d. Optionally, specify the working set, if any, that you want to include in this project.
- A Maven POM file will be generated automatically for this project.
4. If you want to include parent POM information in the file from another project in this workspace, click **Next**, select the **Specify Parent from Workspace** check box, and then select the required parent project.
  5. Click **Finish**. The new projects will be created in the workspace. If you browse inside each project, you will see each project structure as follows:



### ***Importing a Synapse configuration***

A Synapse configuration file contains one or more ESB artifacts, such as endpoints and sequences. You can import an existing Synapse configuration to import all its ESB artifacts in one step.

Follow the steps below to import an existing Synapse configuration into an ESB Config project.

1. Open the Developer Studio Dashboard (click **Developer Studio > Open Dashboard**) and click **Synapse** in the Enterprise Service Bus area.
2. Specify the Synapse configuration file by typing its full path name or clicking **Browse** and navigating to the file.
3. In the **Save Synapse Configuration in** field, specify an existing ESB Config project in your workspace where you want to save the configuration, or click **Create new ESB Project** to create a new ESB Config project and save the Synapse configuration there.
4. Select the artifacts you want to import.
5. Click **Finish**. The artifacts you selected are created in the subfolders of `src/main/synapse-config` folder under the ESB Config project you specified, and the first artifact appears in the editor.

### ***Creating a Smooks configuration artifact***

Smooks is an extensible framework for building applications that process data, such as binding data objects and transforming data. To create a Smooks configuration artifact, you must first create a registry resource as described in [Creating Governance Registry Artifacts](#). When creating the registry resource, select the **From Existing Template** option and select **Smooks Configuration** as the template.

## Create A New Registry Resource

Give a name for the registry resource



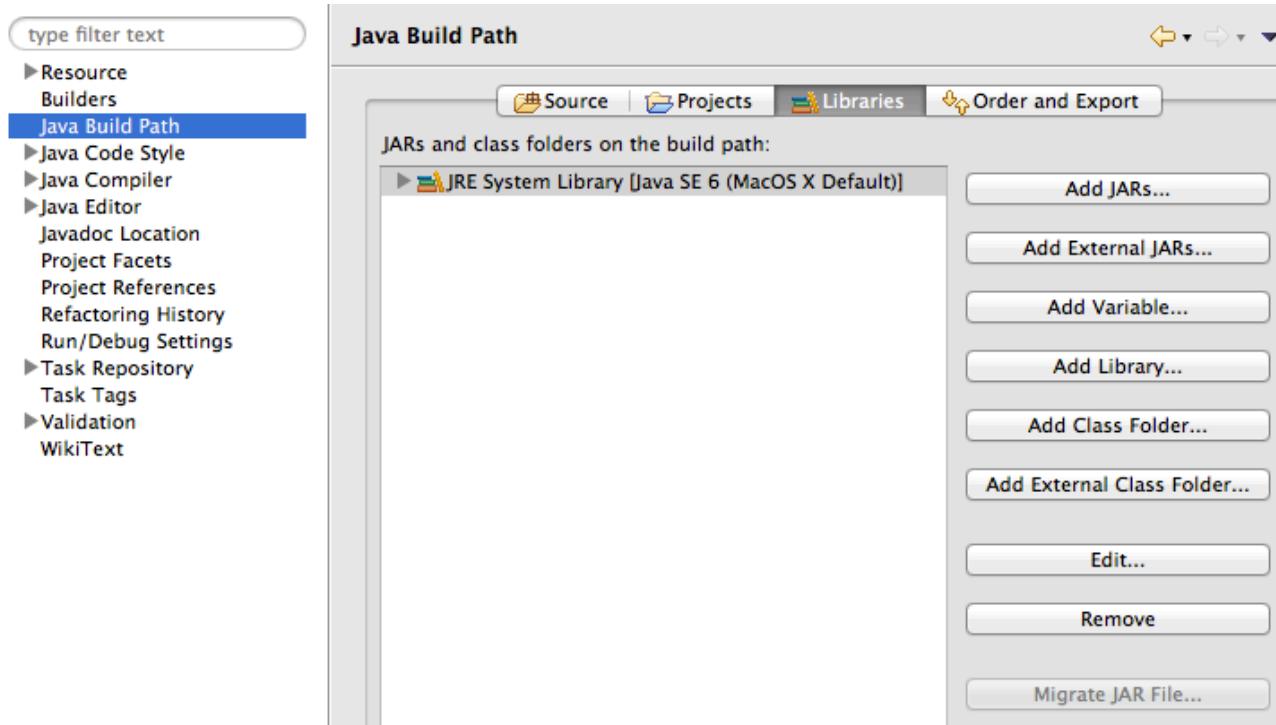
Resource Name	<input type="text" value="smooksconfig"/>
Artifact Name	<input type="text" value="smooksconfig"/>
Template	<input type="text" value="Smooks Configuration"/>
Registry	<input type="text" value="conf"/>
Registry path	<input type="text" value="smooks/"/>
Save resource in	<input type="text" value="MyRegistryResourcesProject"/> <a href="#">Browse...</a>
<a href="#">Create new Project...</a>	

The `smooksconfig.xml` file is created. Double-click it in the Project Explorer to open it in the embedded JBoss Smooks editor.

Click **Input Task**, create the data mapping, and save the configuration file.

Before you can run the Smooks configuration, you must add libraries from the Smooks framework to your registry resources project.

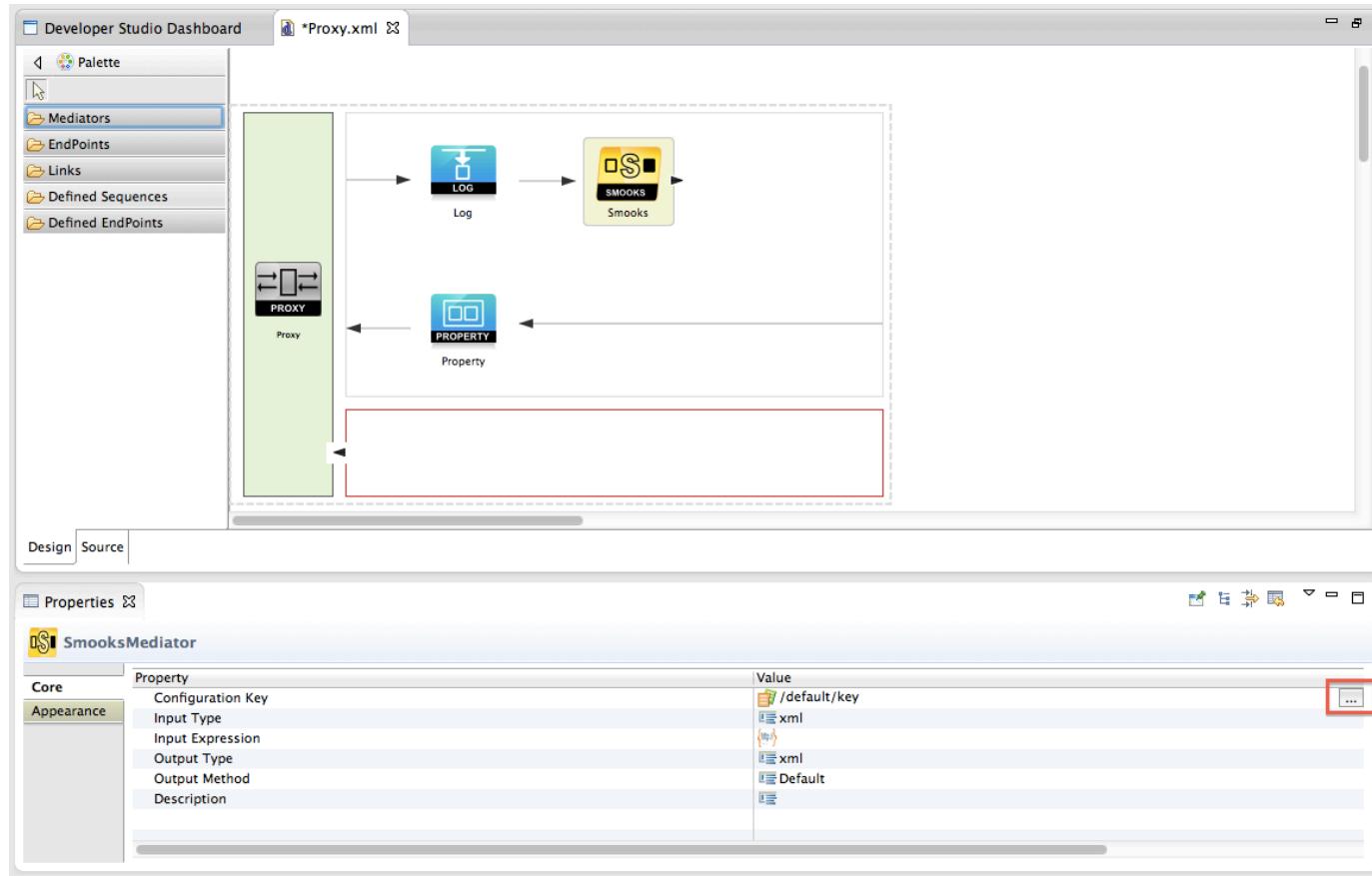
1. Right-click the registry resources project in the Project Explorer and click **Properties**.
2. In the list on the left, click **Java Build Path**, and then click the **Libraries** tab.



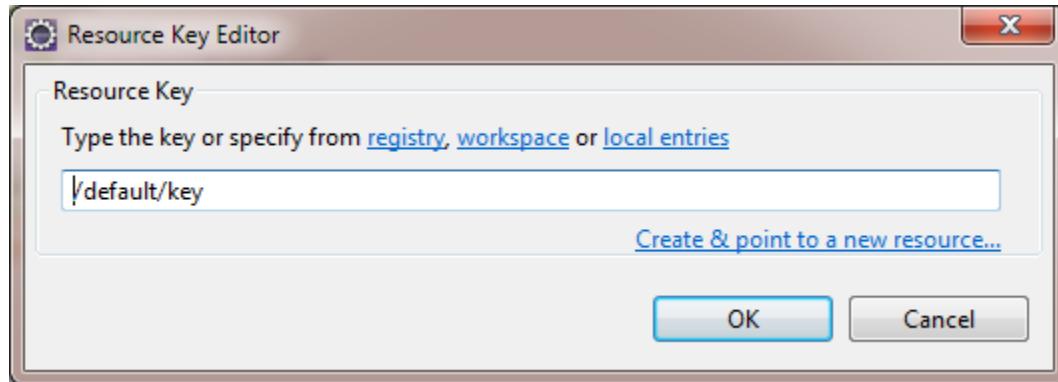
3. Click **Add Library**, click **WSO2 Classpath Libraries**, and then click **Next**.
4. In the **WSO2 Classpath Libraries** dialog box, click the **Smooks** tab, click **Select All**, and click **Finish**.
5. In the **Properties** dialog box, click **OK**.

All the Smooks-related libraries have been added to the project classpath. You can now run the Smooks configuration file by right-clicking the file and choosing **Run As > Smooks Run Configuration**. If your Smooks configuration is correct, the console displays the results according to the input model and output model you specified.

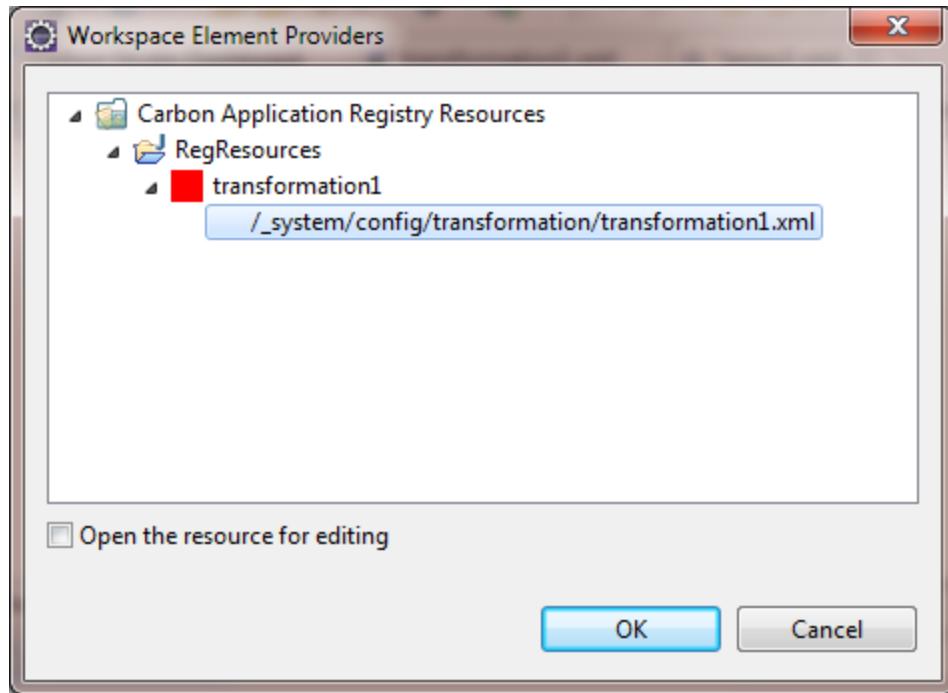
You can now add the Smooks configuration artifact to a proxy service or sequence to use it in the ESB. To do so, create a [proxy service](#). Drag and drop a Log mediator and a Smooks mediator to the **InSequence**. Double-click on the **Smooks** mediator to see the **Property** view. Click the button at the right hand corner of the **Configuration Key** field.



The **Resource Key Editor** dialog box appears. Specify from where you should select the resource file.



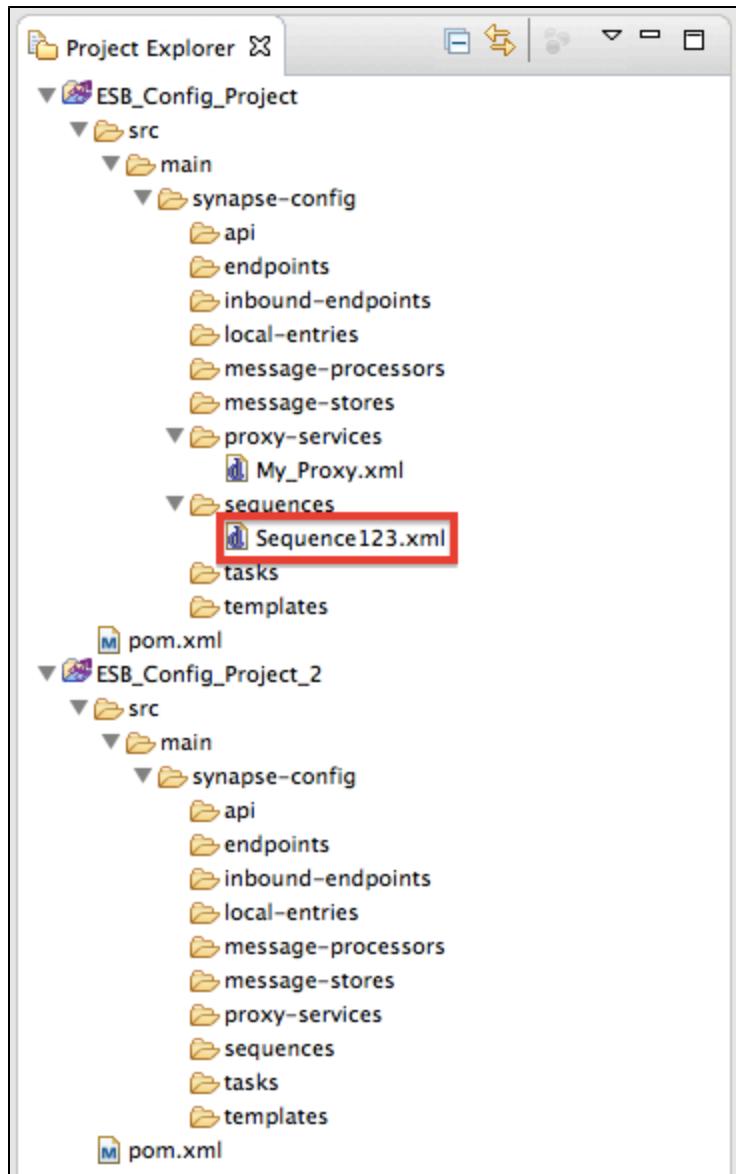
Select **Workspace** option since we have the created **Smooks Configuration** in our workspace. Browse for the **Smooks Configuration** file that we have created and click **OK**.

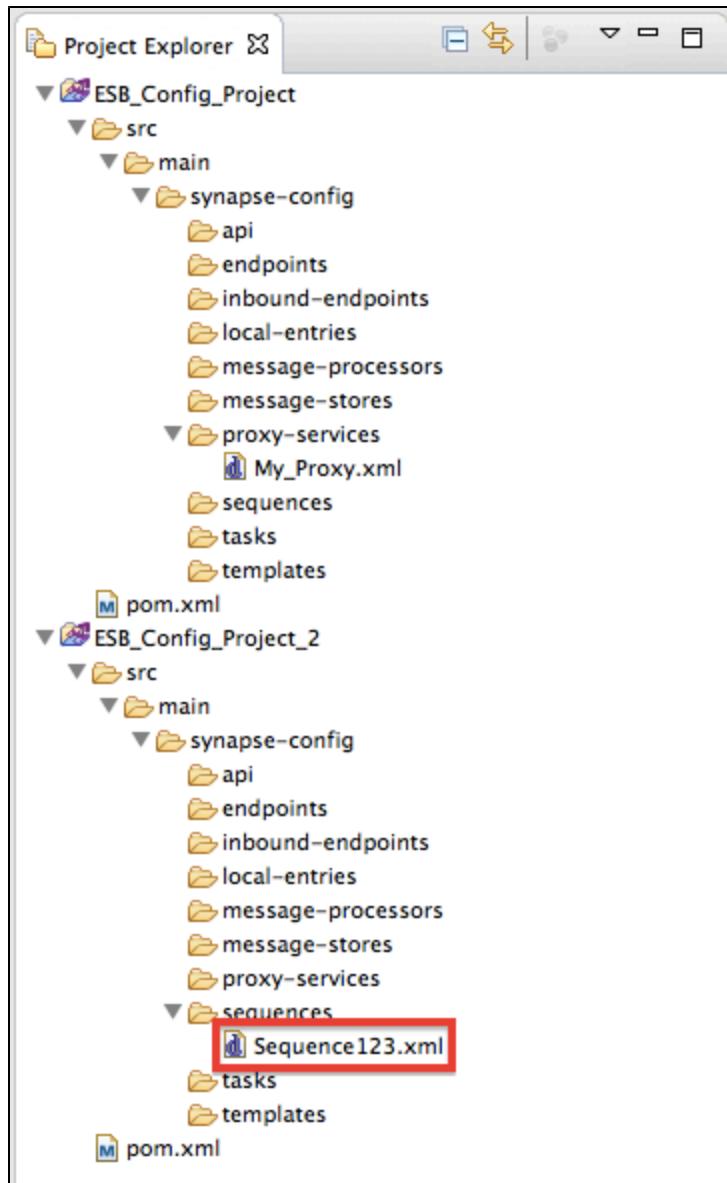


Now you have successfully referred to the Smooks configuration within your proxy service.

#### ***Moving ESB artifacts between projects***

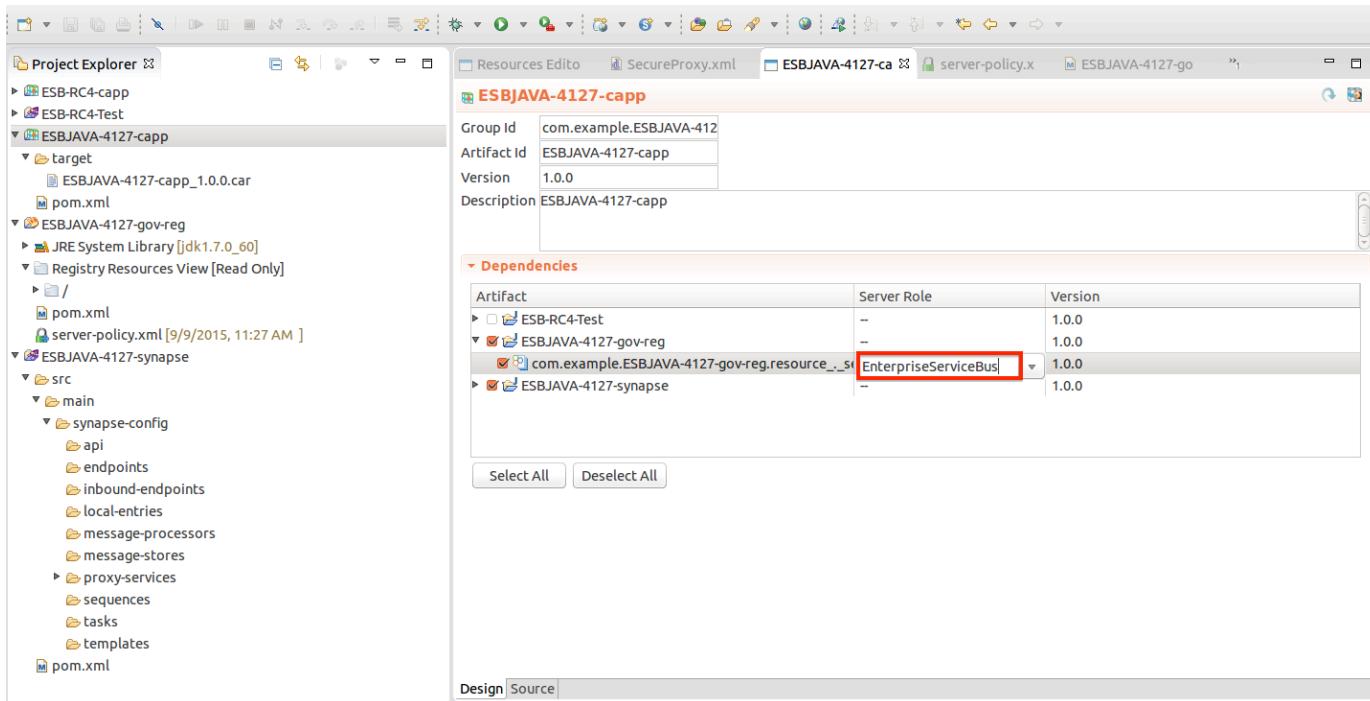
To move an ESB artifact from one project to another, right-click on the artifact (in this instance, Sequence123.xml), click **Move** and then select the folder to which you want to move the artifact. It is also possible to simply drag and drop the artifact.





#### Deploying the ESB Config project

Once you create all the ESB components such as sequences, proxy services, endpoints, local-entries, you can create a Composite Application Project to group them and create a Composite Application Archive. Before creating the **CAR** file, make sure that the server role is set to **EnterpriseServiceBus**.



For more information, see [Packaging Artifacts into Composite Applications](#).

### Packaging your Artifacts into Composite Applications

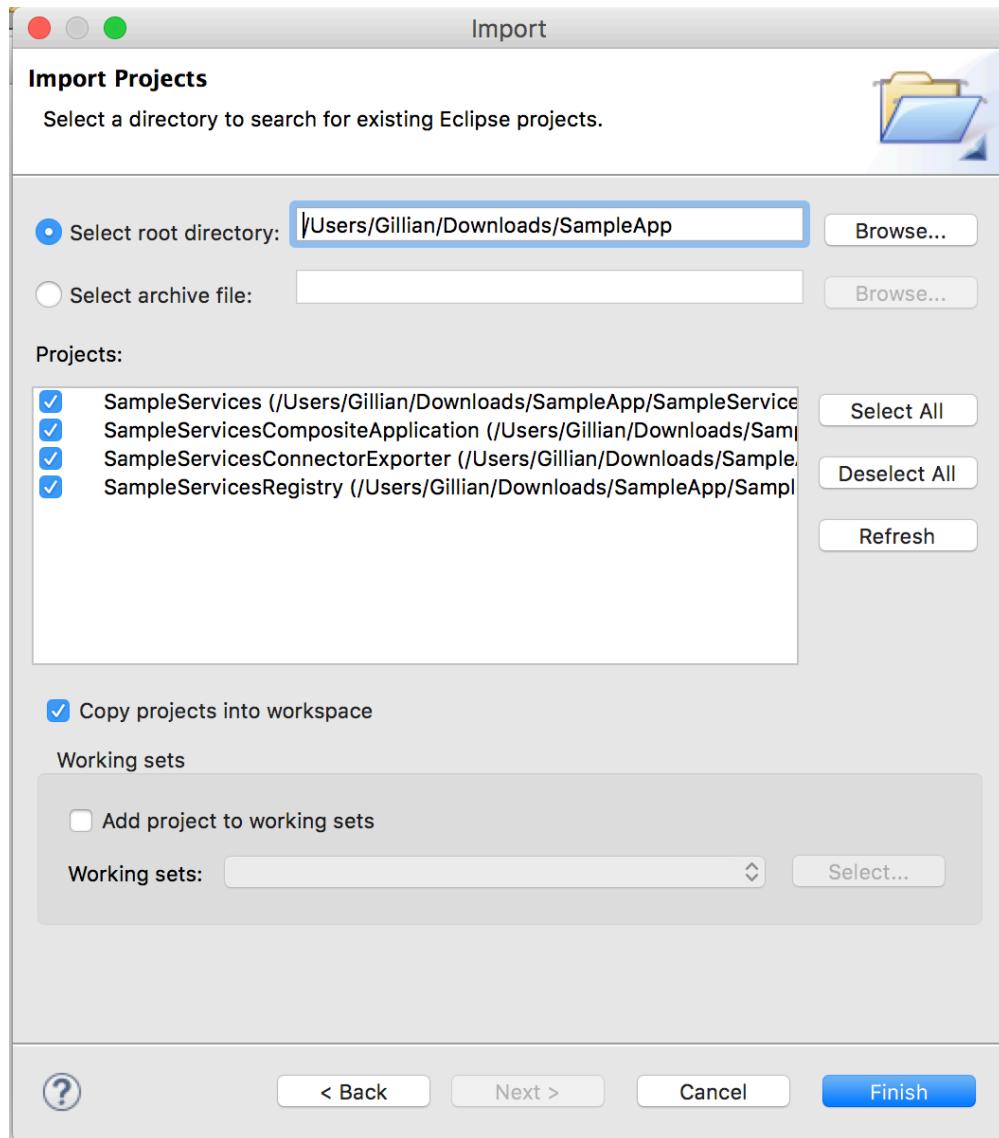
A Tooling project can contain multiple ESB artifacts such as endpoints, mediators, registry resources, etc. When your Tooling project is done, you can bundle all the configuration files and artifacts that are in the tooling environment to a Composite Application (C-App) and move them to the ESB server by deploying the C-App in the server.

To learn more about C-Apps and how to package and deploy artifacts into C-Apps, see [Working with Composite Applications](#) in the WSO2 Admin Guide.

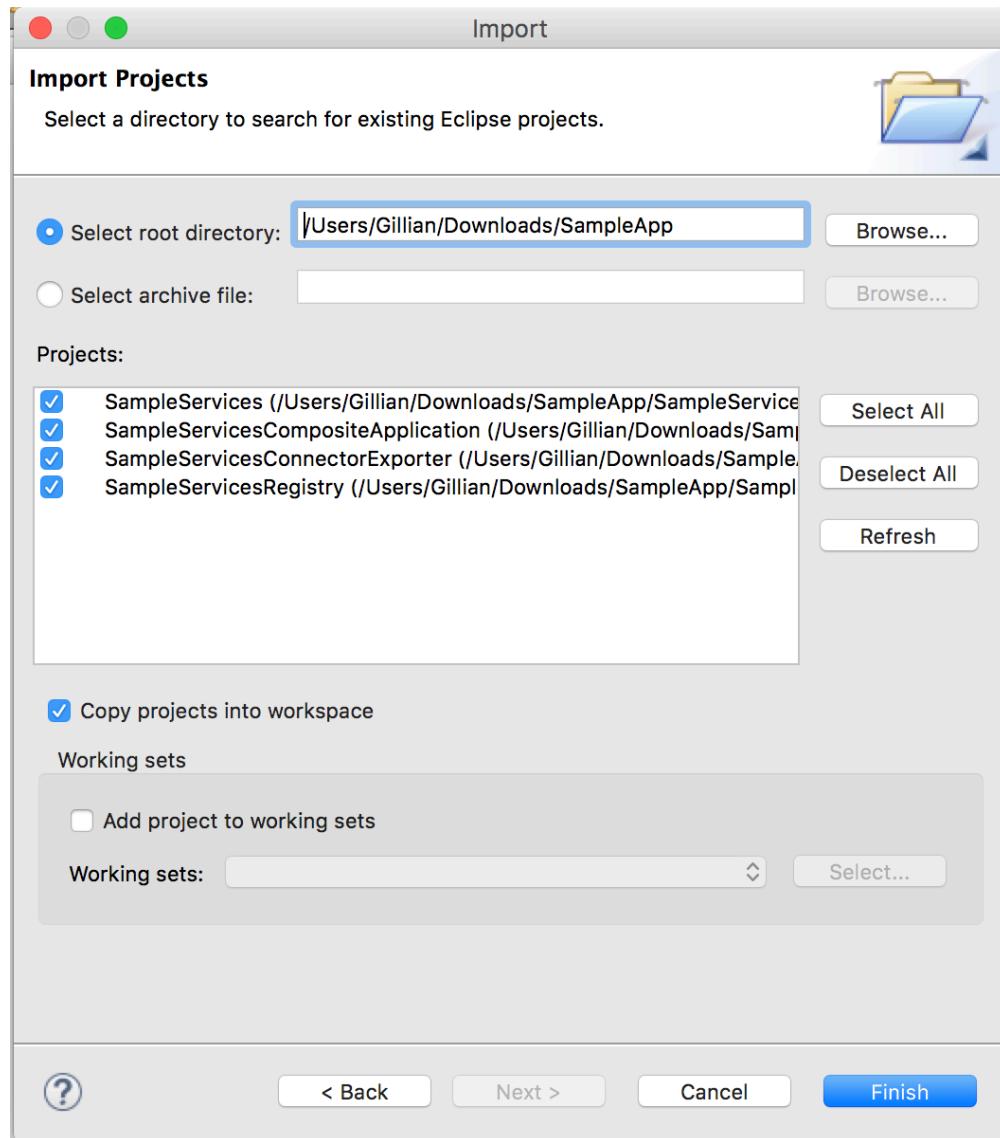
### Importing Existing Projects into Workspace

This page provides instructions on how to import existing WSO2 ESB tooling projects into your Eclipse workspace.

1. Suppose you already have a zip file containing ESB artifact projects, unzip the file.
2. Go to your Eclipse workspace and navigate to **File -> Import**. Select **Existing WSO2 Projects into workspace** under WSO2 category and click **Next**.

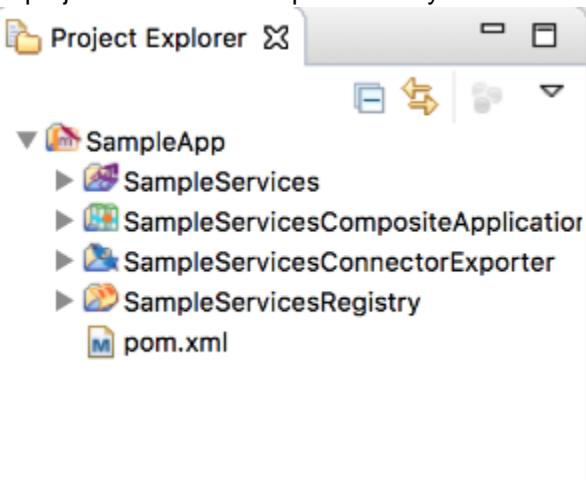


3. Click on **Browse...** and select the unzipped folder containing ESB artifacts projects. All projects in the folder will then be listed. Click **Finish**.



Select **Copy projects into workspace** checkbox if you want to save the project in Eclipse workspace.

4. All projects will now be imported and you can view them in the project explorer.



## Triggering Messages

Messages come into the ESB through the following message entry points:

- **APIs** - APIs accept REST messages that allow clients to provide additional information on how to manage the message in the ESB.
- **Proxy Services** - A proxy service receives messages that are sent to a specific [endpoint](#).
- **Inbound Endpoints** - An inbound endpoint injects a message directly from the transport layer to the mediation layer, without going through the Axis engine.
- **Tasks** - A task injects a message into the ESB at a scheduled interval.

When the ESB receives a message, it sends it either to a proxy service or to the [Main sequence](#) for handling. The proxy or sequence is configured with [message mediation](#), which controls how the message will be transformed, filtered, forwarded, etc.

See the following topics for more information on how you can invoke message flows in the ESB:

- [Working with APIs](#)
- [Working with Proxy Services](#)
- [Working with Inbound Endpoints](#)
- [Working with Scheduled Tasks](#)
- [Using REST](#)

### Working with APIs

This page describes how you can work with APIs that allows messages to be sent directly into the ESB. It contains the following sections:

- [Introduction](#)
  - Defining an API
  - Specifying the context
  - Using patterns with resources
    - URL mappings
    - URI templates
  - Examples
  - Configuring non-HTTP endpoints
- [Basic REST architectural principles](#)
  - Uniform interface
  - Stateless
  - Cacheable
  - Client–server
  - Layered system
- [Best practices for designing APIs for use with REST](#)

#### *Introduction*

An API allows you to send a message directly into the ESB and perform specific logic on it based on the instructions in the HTTP call. This section introduces the fundamentals of creating APIs in the ESB. It illustrates the concepts through the XML configuration.

For information on adding an API via the ESB tooling plug-in, see [Working with APIs via WSO2 ESB Tooling](#).

For information on adding an API through the Management Console UI, see [Working with APIs via the Management Console](#).

#### **Defining an API**

The syntax of a REST API is as follows.

```

<api name="API_NAME" context="URI_PATH_OF_API" [hostname="HOST_NAME_OF_SERVER"] [port="PORT_NUMBER"]>
 <resource [methods="GET|POST|PUT|DELETE|OPTIONS|HEAD|PATCH"] [uri-template="URI_TEMPLATE"] [url-mapping="URL_MAPPING"]>
 <inSequence>
 ...
 </inSequence>?
 <outSequence>
 ...
 </outSequence>?
 <faultSequence>
 ...
 </faultSequence>?
 </resource>
</api>

```

An API definition is identified by the `<api>` tag. Each API must specify a unique name and a unique URL context (see below). An API is made of one or more **resources**, which is a logical component of an API that can be accessed by making a particular type of HTTP call. For example:

```

<api name="API_1" context="/order">
 <resource url-mapping="/list" inSequence="seq1" outSequence="seq2" />
</api>

```

Once a request is dispatched to a resource it will be mediated through the in-sequence of the resource. At the end of the in-sequence the request can be forwarded to a back-end application for further processing. Any responses coming from the back-end system are mediated through the out-sequence of the resource. You can also define a fault-sequence to handle any errors that may occur while mediating a message through a resource.

## Specifying the context

An API in WSO2 ESB is analogous to a web application deployed in the ESB runtime. Each API is anchored at a user-defined URL context, much like how a web application deployed in a servlet container is anchored at a fixed URL context. An API will only process requests that fall under its URL context. For example, if a particular API is anchored at the context `"/test"`, only HTTP requests whose URL path starts with `"/test"` will be handled by that API.

If your ESB is deployed in [multi-tenancy mode](#), you must add the tenant domain to the API context when defining the API in the Source view instead of the Design view in the Management Console. For example, if you are defining the API in the tenant `abc.com`, and the context for the API is `/order`, you would specify the context as: `context="/t/abc.com/order"`. If you define the API in the Design view, however, the tenant domain is automatically prepended to the context, so you would just specify the context as `/order` without the tenant.

It is also possible to bind a given API to a user-defined hostname and/or a port number. For example, if your API is anchored at the URL `http://your.host.name/test`, the following requests will be handled by your API:

```

http://your.host.name/test/getNumbers
http://your.host.name/test/calculation/getRate

```

## Using patterns with resources

A resource can be associated with a user-defined **URL mapping** or **URI template**, which allow us to restrict the type of HTTP requests processed by a particular resource. A resource can also be bound to a specific subset of

HTTP verbs and header values, providing additional control over what requests are handled by a given resource.

For example, consider a resource associated with the URL mapping “/foo/\*” and the HTTP verb “GET”. This approach ensures that the resource will only process GET requests whose URL path matches the pattern “/foo/\*”. Therefore, the following requests would be processed and mediated by the resource:

```
GET /test/foo/bar
GET /test/foo/a?arg1=hello
```

The following HTTP requests would **not** be handled by this resource:

```
GET /test/food/bar (URL pattern does not match)
POST /test/foo/bar (HTTP verb does not match)
```

### ***URL mappings***

When a resource is defined with a URL mapping, only those requests that match the given pattern will be processed by the resource. There are three types of URL mappings:

- Path mappings (/test/\*, /foo/bar/\*)
- Extension mappings (\*.jsp, \*.do)
- Exact mappings (/test, /test/foo)

### ***URI templates***

A URI template represents a class of URIs using patterns and variables. When a resource is associated with a URI template, all requests that match the template will be processed by the resource. Some examples of valid URI templates are as follows:

```
/order/{orderId}
/dictionary/{char}/{word}
```

The identifiers within curly braces are considered variables. For example, a URL that matches the template “/order/{orderId}” is as follows:

```
/order/A0001
```

In the above URL instance, the variable "orderId" has been assigned the value “A0001”. Similarly, the following URL adheres to the template “/dictionary/{char}/{word}”:

```
/dictionary/c/cat
```

In this case, the variable “char” has the value “c” and the variable “word” is given the value “cat”.

The ESB provides access to the exact values of the template variables through message context properties. For example, if you have a resource configured with the URI template “/dictionary/{char}/{word}”, and a request “/dictionary/c/cat” is sent to the ESB, it will be dispatched to this resource, and we will be able to retrieve the exact values of the two variables using the “get-property” XPath extension of WSO2 ESB and prefixing the variable with “uri.var.” as shown in the following example:

```
<log level="custom">
 <property name="Character" expression="get-property('uri.var.char')"/>
 <property name="Word" expression="get-property('uri.var.word')"/>
</log>
```

This log mediator configuration would generate the following output for the request “/dictionary/c/cat”:

```
LogMediator Character = c, Word = cat
```

### ***Examples***

Let's look at some sample API configurations to understand how we use context, resources, and patterns to control how requests are processed.

```

<api name="API_1" context="/order">
 <resource url-mapping="/list" inSequence="seq1" outSequence="seq2" />
</api>

<api name="API_2" context="/user">
 <resource url-mapping="/list" methods="GET" inSequence="seq3" outSequence="seq4" />
 <resource uri-template="/edit/{userId}" methods="PUT POST" inSequence="seq5"
outSequence="seq6" />
 </api>

<api name="API_3" context="/payments">
 <resource url-mapping="/list" methods="GET" inSequence="seq7" outSequence="seq8" />
 <resource uri-template="/edit/{userId}" methods="PUT POST" outSequence="seq9" >
 <inSequence>
 <log/>
 <send>
 <endpoint key="BackendService"/>
 </send>
 </inSequence>
 </resource>
 <resource inSequence="seq10" outSequence="seq11" />
 </api>

```

You can define a URL mapping to a set of operations as shown in the API\_1 definition, or you can define separate mappings for separate operations as shown in API\_2. Also note the last resource definition in API\_3, which does not specify a URL mapping nor a URI template. This is called the default resource of the API. Each API can have at most one default resource. Any request received by the API that does not match any of the enclosed resource definitions will be dispatched to the default resource of the API. In the case of API\_3, a DELETE request on the URL "/payments" will be dispatched to the default resource as none of the other resources in API\_3 are configured to handle DELETE requests.

For a comprehensive example of using APIs with the ESB, see the article [How to GET a Cup of Coffee the WSO2 Way](#).

### Configuring non-HTTP endpoints

When using a non-HTTP endpoint, such as a JMS endpoint, in the API definition, you must remove the REST\_URL\_POSTFIX property to avoid any characters specified after the context (such as a trailing slash) in the request from being appended to the JMS endpoint. For example:

```

<api xmlns="http://ws.apache.org/ns/synapse" name="EventDelayOrderAPI"
context="/orderdelayAPI">
 <resource methods="POST" url-mapping="/">
 <inSequence>
 <property name="REST_URL_POSTFIX" action="remove" scope="axis2"></property>
 <send>
 <endpoint>
 <address uri=
"jms:/DelayOrderTopic?transport.jms.ConnectionFactoryJNDIName=TopicConnectionFactory&
java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory&
java.naming.provider.url=tcp://localhost:61616&transport.jms.DestinationType=topic">
 </address>
 </endpoint>
 </send>
 </inSequence>
 </resource>
 </api>

```

Notice that we have specified the REST\_URL\_POSTFIX property with the value set to "remove". When invoking this API, even if the request contains a trailing slash after the context (e.g., POST `http://127.0.0.1:8287/orderdelayAPI/` instead of POST `http://127.0.0.1:8287/orderdelayAPI`), the endpoint will be called correctly.

### ***Basic REST architectural principles***

The structure of APIs in the ESB is based on REST architectural principles. This section highlights some of the basic architectural principles of REST based on the following resources:

- Architectural Styles and the Design of Network-based Software Architectures
- Restful Service Best Practices

### **Uniform interface**

REST provides a uniform interface between clients and servers, which simplifies and decouples the architecture, allowing both to evolve independently. To achieve a uniform interface, the following constraints are applied:

- Resource-based: Individual resources are identified in requests using URIs as resource identifiers.
- Manipulation of resources through representations: When a client holds a representation of a resource including any metadata attached, it has enough information to modify or delete the resource on the server, provided it has the permission to do so.
- Self-descriptive messages: Each message includes enough information to describe how to process the message.
- Hypermedia as the engine of application state (HATEOAS): Clients deliver state via body contents, query-string parameters, request headers, and the requested URI. Services deliver state to clients via body content, response codes, and response headers. This is technically referred to as hypermedia (or hyperlinks within hypertext).

### **Stateless**

The necessary state to handle the request is contained within the request itself, whether as part of the URI, query-string parameters, body, or headers.

### **Cacheable**

As on the World Wide Web, clients can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or non-cacheable to prevent clients from reusing stale or inappropriate data in response to further requests.

## Client–server

The uniform interface separates clients from servers. For example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface is not altered.

## Layered system

A client cannot ordinarily detect whether it is connected directly to the back-end server or to an intermediary along the way. This allows for load balancing and caching.

### **Best practices for designing APIs for use with REST**

This section highlights some best practices from <https://s3.amazonaws.com/tfpearsoncollege/bestpractices/RESTful+Best+Practices.pdf> to keep in mind when designing your APIs.

- **Use meaningful resource names** to clarify what a given request does. A RESTful URI should refer to a resource that is a thing instead of an action. The name and structure of URIs should convey meaning to those consumers.
- **Use plurals in node names** to keep your API URIs consistent across all HTTP methods.
- **Use HTTP methods appropriately.** Use POST, GET, PUT, DELETE, OPTIONS and HEAD in requests to clarify the purpose of the request. The POST, GET, PUT and DELETE methods map to the CRUD methods Create, Read, Update, and Delete, respectively. Each resource should have at least one method.
- **Create at most only one default resource** (a resource with neither a uri-template nor a url-mapping) for each API.
- **Offer both XML and JSON** whenever possible.
- **Use abstraction when it's helpful.** The API implementation does not need to mimic the underlying implementation.
- **Implement resource discoverability through links (HATEOAS).** As mentioned in the previous section, the application state should be communicated via hypertext. The API should be usable and understandable given an initial URI without prior knowledge or out-of-band information.
- **Version your APIs** as early as possible in the development cycle. At present, the ESB identifies each API by its unique context name. If you introduce a version in the API context (e.g., /Service/1.0.0), you can update it when you upgrade the same API (e.g., /Service/1.0.1).
- **Secure your services** using OAuth2, OpenID, or another authentication/authorization mechanism. See also [Securing APIs](#).

## Working with APIs via WSO2 ESB Tooling

You can create a new API or import an existing API from the file system using WSO2 ESB tooling. You can also add an API resource to an existing API via WSO2 ESB tooling.

You need to have WSO2 ESB tooling installed to create a new API or to import an existing API via ESB tooling. For instructions on installing WSO2 ESB tooling, see [Installing WSO2 ESB Tooling](#).

### **Creating a new API**

Follow these steps to create a new API.

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **REST API** on the **Developer Studio Dashboard**.
3. Leave the first option selected and click **Next**.
4. Type a unique name for the new API and specify the context, hostname, and port. For more information, see [Configuring Endpoints using REST APIs](#).
5. Do one of the following:

- To save the API in an existing ESB Config project in your workspace, click **Browse** and select that project.
  - To save the API in a new ESB Config project, click **Create new ESB Project** and create the new project.
6. Click **Finish**. The REST API is created in the `src/main/synapse-config/api` folder under the ESB Config project you specified. When prompted, you can open the file in the editor, or you can right-click the API in the project explorer and click **Open With > ESB Editor**. Click its icon in the editor to view its properties.

### **Importing an API**

Follow these steps to import an existing API into an ESB Config project.

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **REST API** on the **Developer Studio Dashboard**.
3. Select **Import API Artifact** and click **Next**.
4. Specify the XML file that defines the API by typing its full pathname or clicking **Browse** and navigating to the file.
5. In the **Save API In** field, specify an existing ESB Config project in your workspace where you want to save the API, or click **Create new ESB Project** to create a new ESB Config project and save the API configuration there.
6. If there are multiple API definitions in the file, click **Create ESB Artifacts**, and then select the artifacts you want to import.
7. Click **Finish**. The APIs you selected are created in the subfolders of the `src/main/synapse-config/api` folder under the ESB Config project you specified, and the first API appears in the editor.

### **Adding an API resource**

Follow these steps below to add an API resource to an existing API.

1. Open an existing API in the editor.
2. In the tool palette, click **APIResource** in the API section and simply drag and drop the resource to the REST API.
3. You can now add mediators, endpoints etc to the resource and save.

### **Working with APIs via the Management Console**

You can easily add APIs as well as delete APIs that are no longer required via the Management Console. The following topics describe how you can add APIs and delete APIs:

- [Adding APIs](#)
- [Deleting APIs](#)

### **Adding APIs**

Follow the steps below to add an endpoint

1. Open the ESB management console.
2. Click the **Main** tab, and then click **APIs** to open the **Deployed APIs** page.
3. Click **Add API** to open the **Add API** page.

4. Enter values for the following parameters in the header of the page as required.

Parameter Name	Description	Data Type	Required/Optional	Best Practices
<b>API Name</b>	A unique name for the API.	String	Required	
<b>Context</b>	This parameter specifies a URL context to which the requests processed by the REST API should be limited. See <a href="#">Specifying the Context</a> for further information.	String	Required	Version your APIs as early as possible in the development cycle. At present, the ESB identifies each API by its unique context name. If you introduce a version in the API context (e.g., /Service 1.0.0), you can update it when you upgrade the same API (e.g., /Service 1.0.1)
<b>Host Name</b>	The host at which the API is anchored.	String	Optional	
<b>Port</b>	The port of the REST API.	Integer	Optional	
<b>Add Resource</b>	Click this link to add one or more		At least one resource is required.	Create at most only one default resource (a resource with neither a uri-template nor a url-mapping) for each API.  The parameters that can be configured for each resource is described in detail in the next section.

5. Click **Add Resource** to expand the page and display the **Resource** section.

**Resource**

Methods	<input type="checkbox"/> GET <input type="checkbox"/> POST <input type="checkbox"/> PUT <input type="checkbox"/> DELETE <input type="checkbox"/> OPTIONS <input type="checkbox"/> HEAD
URL Style	<input type="button" value="None"/>
<b>In Sequence</b>	
<input checked="" type="radio"/> None <input type="radio"/> Define Inline <input type="radio"/> Pick From Registry <input type="radio"/> Use Existing Sequence	
<b>Out Sequence</b>	
<input checked="" type="radio"/> None <input type="radio"/> Define Inline <input type="radio"/> Pick From Registry <input type="radio"/> Use Existing Sequence	
<b>Fault Sequence</b>	
<input checked="" type="radio"/> None <input type="radio"/> Define Inline <input type="radio"/> Pick From Registry <input type="radio"/> Use Existing Sequence	
<input type="button" value="Update"/>	

Configure the following parameters for the resource.

Parameter Name	Description	Data Type	Required/Optional
<b>Methods</b>	Select the required check boxes to indicate the HTTP method to be used to invoke the REST API. See <a href="#">Best practices for designing REST APIs</a> for more information.	boolean	Required
<b>URL Style</b>	Select a value for his parameter to indicate whether you are specifying a URL mapping or a URI template. A new data field named <b>URL-Mapping</b> or <b>URI-Template</b> will appear based on the selection. Enter the required pattern in this data field.	string	Required

<b>In Sequence</b>	This specifies the mediation flow for incoming messages. Possible values are as follows. <ul style="list-style-type: none"> <li>• <b>None:</b> Select this to omit the sequence.</li> <li>• <b>Define Inline:</b> Select this if you want to define a new sequence and then define the sequence as described in <a href="#">Adding a Mediation Sequence</a>.</li> <li>• <b>Pick From Registry:</b> Select this to use an existing sequence saved in the <a href="#">Registry</a>.</li> <li>• <b>Use Existing Sequence:</b> Select this to use an existing sequence that already exists in the Synapse Configuration.</li> </ul>	One of the four options should be selected.
<b>Out Sequence</b>	This specifies the mediation flow for outgoing messages. Possible values are as follows. <ul style="list-style-type: none"> <li>• <b>None:</b> Select this to omit the sequence.</li> <li>• <b>Define Inline:</b> Select this if you want to define a new sequence and then define the sequence as described in <a href="#">Adding a Mediation Sequence</a>.</li> <li>• <b>Pick From Registry:</b> Select this to use an existing sequence saved in the <a href="#">Registry</a>.</li> <li>• <b>Use Existing Sequence:</b> Select this to use an existing sequence that already exists in the Synapse Configuration.</li> </ul>	One of the four options should be selected.
<b>Fault Sequence</b>	This specifies the mediation flow for messages with errors. Possible values are as follows. <ul style="list-style-type: none"> <li>• <b>None:</b> Select this to omit the sequence.</li> <li>• <b>Define Inline:</b> Select this if you want to define a new sequence and then define the sequence as described in <a href="#">Adding a Mediation Sequence</a>.</li> <li>• <b>Pick From Registry:</b> Select this to use an existing sequence saved in the <a href="#">Registry</a>.</li> <li>• <b>Use Existing Sequence:</b> Select this to use an existing sequence that already exists in the Synapse Configuration.</li> </ul>	One of the four options should be selected.

6. Click **Update** to update the resource to the configuration. Repeat this step to enter as many resources as required.
7. Click **Save** to save the API. The API would now appear in the list of deployed APIs.

## Deleting APIs

Follow the steps below to delete an API.

1. Open the ESB management console.
2. Click the **Main** tab, and then click **APIs** to open the **Deployed APIs** page. All the available APIs will be listed.
3. Click **Delete** for the relevant API. Then click **Yes** in the message which appears to confirm whether you want the API to be deleted. The API will be removed from the deployed APIs list

## Configuring Specific Use Cases

This page describes how to configure APIs for the following use cases:

- Setting query parameters on outgoing messages

- Content negotiation
- Transforming the content type
  - Setting up the back end
  - Configuring the API
  - Executing the sample
- Enabling REST to SOAP
  - Setting up the back end
  - Configuring the API
  - Executing the sample
  - Setting HTTP status codes
- Enabling REST to JMS
  - Setting up the back end
  - Configuring the API
  - Executing the sample
- Exposing a back-end REST service using a different API
  - Setting up the back end
  - Configuring the API
  - Executing the sample
- Handling non-matching resources
  - Setting up the back end
  - Configuring the API
  - Creating the sequence
  - Executing the sample
- Retrieving metadata associated with the state of the resource using the HEAD request method
  - Setting up the backend
  - Configuring the API
  - Executing the sample
- Sending form data
  - Setting up the backend
  - Configuring the API
  - Executing the sample

### Setting query parameters on outgoing messages

REST clients use query parameters to provide inputs for the relevant operation. These query parameters may be required to carry out the back-end operations either in a REST service or a proxy service. Let's take a sample request and see how these parameters can be set in the outgoing message.

```
curl -v -H "Content-Type: application/xml" -d
"<Customer><id>123</id><name>John</name></Customer>"
http://localhost:8282/stockquote/view/IBM?param1=value1¶m2=value2
```

In this request, there are two query parameters (customer name and ID) that must be set in the outgoing message from the ESB. We can configure the API to set those parameters as follows:

```
<definitions>

<api xmlns="http://ws.apache.org/ns/synapse" name="StockQuoteAPI"
context="/stockquote">

<resource uri-template="/view/{symbol}" methods="GET">

 <inSequence>
 <payloadFactory>
 <format>
 <m0:getQuote xmlns:m0="http://services.samples">
```

```

<m0:request>
 <m0:symbol>$1</m0:symbol>
 <m0:customerName>$2</m0:customerName>
 <m0:customerId>$3</m0:customerId>
</m0:request>
</m0:getQuote>
</format>
<args>
 <arg expression="get-property('uri.var.symbol')"/>
 <arg expression="get-property('query.param.param1')"/>
 <arg expression="get-property('query.param.param2')"/>
</args>
</payloadFactory>
<send>
 <endpoint>
 <address uri=" http://localhost:9000/services/SimpleStockQuoteService "
format="soap11"/>
 </endpoint>
</send>
</inSequence>
<outSequence>
 <send/>
</outSequence>
</resource>
</api>
</definitions>

```

The query parameter values can be accessed through the "get-property" function by specifying the parameter number as highlighted in the above request.

## Content negotiation

Content negotiation is a mechanism by which a client and a server communicate with each other and decide on a content format to use for data transfer. In our samples so far, we used XML (application/xml) as the primary means of data transfer. However we could use other content formats such as plain text and JSON to achieve the same result.

Real-world client applications usually have preferred content types. For example:

- A web browser usually prefers HTML.
- A Java-based desktop application usually prefers plain-old XML (POX).
- A mobile application usually prefers JSON.

To maintain interoperability, the server-side applications should be prepared to serve content using any of these

formats. Using content negotiation, the client can indicate its content type preferences to the server, and the server can serve the requests using a content type preferred by the client.

The HTTP specification provides the basic elements to build a powerful content negotiation framework. The HTTP client can indicate its content type preferences by sending the Accept header along with the requests. First, we need to retrieve the value of the Accept header sent by the client. We do this in the in-sequence using the \$trp XPath variable, as follows:

```
<property name="TEST_CLIENT_ACCEPT" expression="$trp:Accept" />
```

Now in the out-sequence we can run a few comparisons to see which content type to use for sending the response. We use the \$ctx XPath variable to specify the TEST\_CLIENT\_ACCEPT property in the message context.

```
<switch source="$ctx:TEST_CLIENT_ACCEPT">
 <case regex=".*atom.*">
 <send/>
 </case>
 <case regex=".*text/html.*">
 <send/>
 </case>
 <case regex=".*json.*">
 <send/>
 </case>
 <case regex=".*application/xml.*">
 <send/>
 </case>
 <default>
 <send/>
 </default>
</switch>
```

The above configuration results in the following priority order of content types.

1. Atom (Also used as default)
2. HTML
3. JSON
4. POX

To try this out, invoke the following Curl commands and see how the response format changes according to the value of the Accept header.

```
curl -v http://localhost:8281/orders
curl -v -H "Accept: application/xml" http://localhost:8281/orders
curl -v -H "Accept: application/json" http://localhost:8281/orders
curl -v -H "Accept: text/html" http://localhost:8281/orders
```

## Transforming the content type

This section describes how you can transform the content type of a message using an API. In this scenario, the API exposes a REST back-end service that accepts and returns XML and JSON messages for HTTP methods as follows:

- GET - response is in JSON format
- POST - accepts XML request and returns response in JSON format
- PUT - accepts JSON request and returns response in JSON format
- DELETE - empty request body should be sent

Transformation in HTTP GET:



Transformation in HTTP POST:



Transformation in HTTP PUT:



### ***Setting up the back end***

For the REST back-end service, we will use a modified version of the [JAX-RS Advanced sample](#), which is available in the WSO2 Application Server. The sample is modified to avoid complications in content types when integrating with WSO2 ESB. The only difference is that the 'serviceURL' of the sample used in this use case will be '[http://localhost:9763/StarbucksService/services/Starbucks\\_Outlet\\_Service](http://localhost:9763/StarbucksService/services/Starbucks_Outlet_Service)'.

When using Application Server 5.1.0 or later, be sure to add the CXF classloading environment to the <AS\_HOME>/repository/conf/tomcat/webapp-classloading.xml file as in the following configuration (see [Webapp Classloading](#) for details on configuring class loading in Application Server):

```

<Classloading xmlns="http://wso2.org/projects/as/classloading">
 <ParentFirst>false</ParentFirst>
 <Environments>CXF,Carbon</Environments>
</Classloading>

```

Since we are going to run both WSO2 ESB and the WSO2 Application Server on the same machine, offset the ports of the application server by changing the offset value to 1 in <AS\_HOME>/repository/conf/carbon.xml.

To deploy the REST back-end, download the StarbucksService.war from <https://svn.wso2.org/repos/wso2/people/charitham/REST-API/StarbucksService.war>) and [deploy it in Application Server](#).

### ***Configuring the API***

Create an API using the following configuration:

```

<api xmlns="http://ws.apache.org/ns/synapse" name="StarbucksService"
context="/Starbucks_Service">
 <resource methods="POST" url-mapping="/orders/add">
 <inSequence>
 <property name="REST_URL_POSTFIX" scope="axis2" action="remove"/>
 <send>
 <endpoint>
 <address
uri="http://localhost:9764/StarbucksService/services/Starbucks_Outlet_Service/orders/"
/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <log level="full"/>
 <property name="messageType" value="application/xml" scope="axis2"/>
 <send/>
 </outSequence>
 </resource>
 <resource methods="PUT" url-mapping="/orders/edit">
 <inSequence>
 <log level="full"/>
 <property name="REST_URL_POSTFIX" scope="axis2" action="remove"/>
 <property name="messageType" value="application/json" scope="axis2"/>
 <property name="ContentType" value="application/json" scope="axis2"/>
 <send>
 <endpoint>
 <address
uri="http://localhost:9764/StarbucksService/services/Starbucks_Outlet_Service/orders/"
format="rest"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <log level="full"/>
 <property name="messageType" value="application/xml" scope="axis2"/>
 <send/>
 </outSequence>
 </resource>
 <resource methods="DELETE GET" uri-template="/orders/{id}">
 <inSequence>
 <send>
 <endpoint>
 <address
uri="http://localhost:9764/StarbucksService/services/Starbucks_Outlet_Service/"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <log level="full"/>
 <property name="messageType" value="application/xml" scope="axis2"/>
 <send/>
 </outSequence>
 </resource>
</api>

```

### **Executing the sample**

The context of the API is '/Starbucks\_Service'. For every HTTP method, a url-mapping or uri-template is defined, and the URL to call the methods differ with the defined mapping or template.

Following is the CURL command to send a GET request to the API:

```
curl -v -X GET http://localhost:8280/Starbucks_Service/orders/123
```

The response from the back end to the ESB will be:

```
{ "Order":{ "additions": "Milk", "drinkName": "Vanilla Flavored Coffee", "locked":false, "orderId":123 } }
```

The ESB transforms this response to XML and send it back as:

```
<Order>
 <additions>Milk</additions>
 <drinkName>Vanilla Flavored Coffee</drinkName>
 <locked>false</locked>
 <orderId>123</orderId>
</Order>
```

Following is the cURL command to send an HTTP POST request to the API:

```
curl -v -H "Content-Type: application/xml" -X POST -d @placeOrder.xml http://localhost:8280/Starbucks_Service/orders/add
```

where placeOrder.xml has the following content on the order:

```
<Order>
 <drinkName>Mocha Flavored Coffee</drinkName>
 <additions>Caramel</additions>
</Order>
```

This XML request will be sent to the back end, which will send back a JSON response to the ESB:

```
{ "Order":{ "additions": "Caramel", "drinkName": "Mocha Flavored Coffee", "locked":false, "orderId": "d088a289-1be3-453b-ab37-7609828d2197" } }
```

The ESB converts this response to XML and sends it back to the client:

```
<Order>
 <additions>Caramel</additions>
 <drinkName>Mocha Flavored Coffee</drinkName>
 <locked>false</locked>
 <orderId>d088a289-1be3-453b-ab37-7609828d2197</orderId>
</Order>
```

Following is the cURL command for sending an HTTP PUT request:

```
curl -v -H "Content-Type: application/xml" -X PUT -d @editOrder.xml http://localhost:8280/Starbucks_Service/orders/edit
```

where editOrder.xml has the following syntax:

```
<Order>
 <orderId>d088a289-1be3-453b-ab37-7609828d2197</orderId>
 <additions>Chocolate Chip Cookies</additions>
</Order>
```

The ESB will convert this request to JSON and send it to the back end. The response will be in JSON format:

```
{"Order": {"additions": "Chocolate Chip Cookies", "drinkName": "Mocha Flavored Coffee", "locked": false, "orderId": "d088a289-1be3-453b-ab37-7609828d2197"}}
```

The ESB converts this response to XML and sends it back to the client:

```
<Order>
 <additions>Chocolate Chip Cookies</additions>
 <drinkName>Mocha Flavored Coffee</drinkName>
 <locked>false</locked>
 <orderId>d088a289-1be3-453b-ab37-7609828d2197</orderId>
</Order>
```

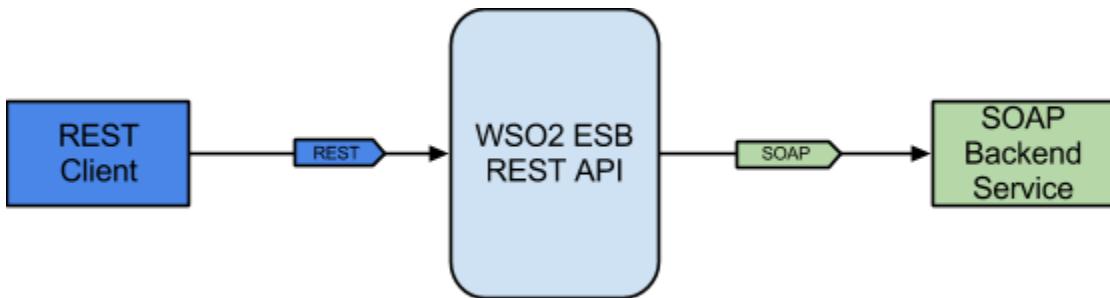
Following is the cURL command for sending an HTTP DELETE request:

```
curl -v -X DELETE http://localhost:8280/Starbucks_Service/orders/d088a289-1be3-453b-ab37-7609828d2197
```

This request will be sent to the back end, and the order with the specified ID will be deleted.

## Enabling REST to SOAP

In this scenario, you expose a SOAP service over REST using an ESB API.



### ***Setting up the back end***

For the SOAP back-end service, we are using the StockQuote Service that is shipped with ESB. Configure and start the back-end service as described in [Starting Sample Back-End Services](#). We will use [cURL](#) as the REST client to invoke the ESB API.

### ***Configuring the API***

Following is a sample API configuration that we can used to implement this scenario.

```

<api xmlns="http://ws.apache.org/ns/synapse" name="StockQuoteAPI"
context="/stockquote">
 <resource uri-template="/view/{symbol}" methods="GET">
 <inSequence>
 <payloadFactory>
 <format>
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>$1</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </format>
 <args>
 <arg expression="get-property('uri.var.symbol')"/>
 </args>
 </payloadFactory>
 <header name="Action" value="urn:getQuote"/>
 <send>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"
format="soap11"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </resource>
 <resource url-mapping="/order/*" methods="POST">
 <inSequence>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
 <property name="OUT_ONLY" value="true"/>
 <header name="Action" value="urn:placeOrder"/>
 <send>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"
format="soap11"/>
 </endpoint>
 </send>
 </inSequence>
 </resource>
</api>

```

### **Executing the sample**

In this API configuration we have defined two resources. One is for the HTTP method GET and the other one is for POST.

In the first resource, we have defined the uri-template as /view/{symbol} so that request will be dispatched to this resource when you invoke the API using the following URI:

`http://127.0.0.1:8280/stockquote/view/IBM`

Following is the cURL command to send this URI:

`curl -v http://127.0.0.1:8280/stockquote/view/IBM`

The context of this API is `stockquote`.

The SOAP payload required for the SOAP back-end service is constructed using the payload factory mediator defined in the inSequence. The value for the <m0:symbol> element is extracted using the following expression:

```
get-property('uri.var.symbol')
```

Here 'symbol' refers to the variable we defined in the uri-template (/view/{symbol}). Therefore, for the above invocation, the 'uri.var.symbol' property will resolve to the value 'IBM'.

After constructing the SOAP payload, the request will be sent to the SOAP back-end service from the <send> mediator, which has an address endpoint defined inline with the format="soap11" attribute in the address element. The response received from the back-end soap service will be sent to the client in plain old XML (POX) format.

In the second resource, we have defined the URL mapping as "/order/\*". Since this has POST as the HTTP method, the client has to send a payload to invoke this. Following is a sample cURL command to invoke this:

```
curl -v -d @placeorder.xml -H "Content-type: application/xml"
http://127.0.0.1:8280/stockquote/order/
```

Save the following sample place order request as placeorder.xml in your local file system and execute the command. This payload is used to invoke a SOAP service.

```
<placeOrder xmlns="http://services.samples">
<order>
 <price>50</price>
 <quantity>10</quantity>
 <symbol>IBM</symbol>
</order>
</placeOrder>
```

You will see something similar to following line printed in the sample axis2 server (back-end server) console.

```
Tue Mar 19 09:30:33 IST 2013 samples.services.SimpleStockQuoteService :: Accepted
order #1 for : 10 stocks of IBM at $ 50.0
```

This SOAP service invocation is an OUT\_ONLY invocation, so the ESB is not expecting any response back from the SOAP service. Since we have set the FORCE\_SC\_ACCEPTED property value to true, the ESB returns a 202 response back to the client.

### **Setting HTTP status codes**

A REST service typically sends HTTP status codes with its response. When you configure an API that send messages to a SOAP back-end service, you can set the status code of the HTTP response within the ESB configuration. To achieve this, set the status code parameter within the out sequence of the API definition:

```
<definitions>
 <api xmlns="http://ws.apache.org/ns/synapse" name="StockQuoteAPI"
context="/stockquote">

 ...
 <outSequence>
 <property name="HTTP_SC" value="201" scope="axis2" />
 </outSequence>
 </resource>
</api>
```

```
</definitions>
```

Now you can try the REST call with the following command.

```
curl -v http://127.0.0.1:8280/stockquote/view/IBM
```

The response message will contain the following response code (201) as configured above.

```
< HTTP/1.1 201 Created
< Content-Type: text/xml; charset=UTF-8
< Server: Synapse-HttpComponents-NIO
< Date: Thu, 21 Mar 2013 09:03:00 GMT
< Transfer-Encoding: chunked
```

## **Enabling REST to JMS**

This section describes how an API can receive an HTTP message sent from a REST client and place it on a JMS queue.

The API receives the HTTP message and sends it to a JMS queue that resides in a message broker. The back-end service listens and picks up the message from this queue.

### ***Setting up the back end***

For the back-end service, we are using the StockQuote Service that is shipped with ESB. Configure and start the back-end service as described in [Starting Sample Back-End Services](#). We will use [cURL](#) as the REST client to invoke the ESB API.

For this sample, we use ActiveMQ 5.5.1 as the message broker. Start ActiveMQ and [configure the JMS transport in ESB to work with ActiveMQ](#) before you proceed.

### ***Configuring the API***

Following is a sample ESB API configuration that we can used to implement this scenario:

```

<api xmlns="http://ws.apache.org/ns/synapse" name="RestToJmsAPI"
context="/stockquote">
 <resource methods="POST" url-mapping="/order/*">
 <inSequence>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
 <property name="OUT_ONLY" value="true"/>
 <send>
 <endpoint>
 <address
uri="jms:/SimpleStockQuoteService?transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory&java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory&java.naming.provider.url=tcp://localhost:61616" format="soap11"/>
 </endpoint>
 </send>
 </inSequence>
 </resource>
</api>

```

In this API configuration, we have set the OUT\_ONLY property to true, as it is a one-way invocation. The message is sent to the JMS queue using the JMS Transport sender. (Note the prefix "jms" in the address endpoint URI.) Since we have set the FORCE\_SC\_ACCEPTED property value to true, the ESB returns a 202 response back to the client.

### **Executing the sample**

Following is a sample cURL command to invoke this API:

```
curl -v -d @placeorder.xml -H "Content-type: application/xml"
http://127.0.0.1:8280/stockquote/order/
```

Save the following sample place order request as "placeorder.xml" in your local file system and execute the above command.

```

<placeOrder xmlns="http://services.samples">
 <order>
 <price>50</price>
 <quantity>10</quantity>
 <symbol>IBM</symbol>
 </order>
</placeOrder>

```

You will see something similar to the following printed in the sample axis2 server (back-end server) console:

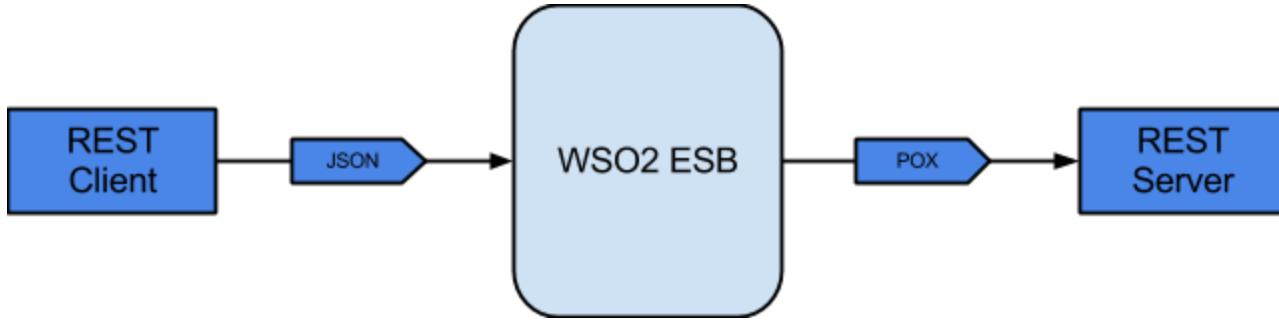
```
Tue Mar 19 09:30:33 IST 2013 samples.services.SimpleStockQuoteService :: Accepted
order #1 for : 10 stocks of IBM at $ 50.0
```

Shut down the back-end server and execute the command again. If you go to the ActiveMQ Console and inspect the SimpleStockQuoteService queue, you will see a message queued there.

For information on enabling a JMS client to send messages to a REST back-end service, see [Front-End JMS Client and Back-End REST Service](#).

### **Exposing a back-end REST service using a different API**

In this scenario, we are exposing a back-end REST service using an ESB API. That means we are exposing a different API to the client for a REST back-end service.



### **Setting up the back end**

For the REST back-end service, we are going to use the [JAX-RS Basics](#) sample service available in the WSO2 Application Server 5.0.1. Configure and deploy the JAX-RS Basics sample by following the instructions for building and running the sample on the [JAX-RS Basics](#) page. Since we are going to run both WSO2 ESB and the WSO2 Application Server on the same machine, we have to offset the ports of Application Server by changing the offset value to 1 in <AS\_HOME>/repository/conf/carbon.xml.

### **Configuring the API**

Following is a sample configuration we can use to configure this scenario:

```

<api xmlns="http://ws.apache.org/ns/synapse" name="ClientServiceAPI"
context="/clientservice">
 <resource methods="OPTIONS DELETE GET" uri-template="/clients/{id}">
 <inSequence>

 <property name="REST_URL_POSTFIX"
expression="fn:concat('/customers/',get-property('uri.var.id'))"
scope="axis2"/>
 <send>
 <endpoint>
 <address
uri="http://localhost:9764/jaxrs_basic/services/customers/customerservice/" />
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </resource>
 <resource methods="POST PUT" uri-template="/clients">
 <inSequence>
 <property name="REST_URL_POSTFIX" value="/customers" scope="axis2" />
 <switch source="$ctx:REST_METHOD">
 <case regex="POST">
 <payloadFactory>
 <format>
 <Customer xmlns="">
 <name>$1</name>
 </Customer>
 </format>
 </payloadFactory>
 </case>
 </switch>
 </inSequence>
 </resource>
 </resource>
</api>

```

```
 <arg expression="//addClient/name" />
 </args>
</payloadFactory>
</case>
<case regex="PUT">
 <payloadFactory>
 <format>
 <Customer xmlns="">
 <id>$1</id>
 <name>$2</name>
 </Customer>
 </format>
 <args>
 <arg expression="//updateClient/id" />
 <arg expression="//updateClient/name" />
 </args>
 </payloadFactory>
</case>
</switch>
<send>
 <endpoint>
 <address
uri="http://localhost:9764/jaxrs_basic/services/customers/customerservice/" />
 </endpoint>
</send>
</inSequence>
<outSequence>
 <send/>
```

```

</outSequence>
</resource>
</api>

```

The service URL for the back-end REST service is:

`http://localhost:9764/jaxrs_basic/services/customers/customerservice/`

The service URL for the REST API (ClientServiceAPI) is:

`http://localhost:8280/clientservice/`

As you can see, we have exposed the back-end REST service in a different context using this API. In addition to altering the context, the API is accepting the payload in a different format.

For the HTTP POST method, the back-end REST service accepts the payload in the following format:

```

<Customer>
 <name>Jack</name>
</Customer>

```

The ClientServiceAPI accepts the payload in the following format:

```

<addClient>
 <name>Jack</name>
</addClient>

```

For the HTTP PUT method, the back-end REST service accepts the payload in the following format:

```

<Customer>
 <id>123</id>
 <name>John</name>
</Customer>

```

The ClientServiceAPI accepts the payload in the following format:

```

<updateClient>
 <id>123</id>
 <name>John</name>
</updateClient>

```

Conversion of the payload format is done using the `<payloadFactory>` mediator.

We have now exposed a different REST API for the REST back-end services using an ESB REST API. In the ClientServiceAPI, there are two resources. The first resource is for the GET, OPTIONS, and DELETE HTTP methods. URL postfix is needed for the REST back-end service and is calculated using the following expression:

`fn:concat(' /customers / ', get-property('uri.var.id'))`

The calculated URL postfix is appended to the back-end service URL, as we have set that value to the `REST_URL_POSTFIX` property. Therefore, the final request URL will be something similar to the following:

`http://localhost:9764/jaxrs_basic/services/customers/customerservice/customers/123`

### **Executing the sample**

Following is the cURL command to send a GET request to the API:

```
curl -v http://localhost:8280/clientservice/clients/123
```

You will see the following line printed as the response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><Customer><id>123</id><name>John</name></Customer>
```

The second resource is for the POST and PUT HTTP methods. The incoming REST request is filtered using the `HTTP_METHOD`, and the appropriate payload required for the back-end REST service is constructed using the `<payloadFactory>` mediator.

Following is the cURL command to send a POST request to the API:

```
curl -v -d @addClient.xml -H "Content-type: application/xml" http://localhost:8280/clientservice/clients
```

Content of the `addClient.xml` file should be in the following format:

```
<addClient>
 <name>WSO2</name>
</addClient>
```

When you execute the above command in the application server console, the following line will be printed:

```
----invoking addCustomer, Customer name is: WSO2
```

Following is the cURL command to send a PUT request to the API:

```
curl -v -d @updateClient.xml -X PUT -H "Content-type: application/xml" http://localhost:8280/clientservice/clients
```

The content of the `updateClient.xml` file should be in the following format:

```
<updateClient>
 <id>123</id>
 <name>ESB</name>
</updateClient>
```

When you execute the above command in the application server console, the following line will be printed:

```
----invoking updateCustomer, Customer name is: ESB
```

### **Handling non-matching resources**

In this scenario, we are defining a sequence to be invoked if the WSO2 ESB is unable to find a matching resource definition for a specific API invocation. This sequence generates a response indicating an error when no matching resource definition is found.

### **Setting up the back end**

For the back-end service, we are using the StockQuote Service that is shipped with ESB. Configure and start the back-end service as described in [Starting Sample Back-End Services](#). We will use [cURL](#) as the REST client to invoke the ESB API.

### **Configuring the API**

Create an API using the following configuration:

```
<api xmlns="http://ws.apache.org/ns/synapse" name="jaxrs" context="/jaxrs">
 <resource methods="GET" uri-template="/customers/{id}">
 <inSequence>
 <send>
 <endpoint>
 <address
uri="http://localhost:9764/jaxrs_basic/services/customers/customerservice"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </resource>
</api>
```

### ***Creating the sequence***

Create a new sequence with the following configuration:

```
<sequence xmlns="http://ws.apache.org/ns/synapse" name="_resource_mismatch_handler_">
 <payloadFactory>
 <format>
 <tp:fault xmlns:tp="http://test.com">
 <tp:code>404</tp:code>
 <tp:type>Status report</tp:type>
 <tp:message>Not Found</tp:message>
 <tp:description>The requested resource (/${1}) is not
available.</tp:description>
 </tp:fault>
 </format>
 <args>
 <arg xmlns:ns="http://org.apache.synapse/xsd"
expression="$axis2:REST_URL_POSTFIX"/>
 </args>
 </payloadFactory>
 <property name="RESPONSE" value="true" scope="default"/>
 <property name="NO_ENTITY_BODY" action="remove" scope="axis2"/>
 <property name="HTTP_SC" value="404" scope="axis2"/>
 <header name="To" action="remove"/>
 <send/>
 <drop/>
</sequence>
```

### ***Executing the sample***

Send an invalid request to the back end as follows:

```
curl -X GET http://localhost:8280/jaxrs/customers-wrong/123
```

You will get the following response:

```

<tp:fault xmlns:tp="http://test.com">
<tp:code>404</tp:code>
<tp:type>Status report</tp:type>
<tp:message>Not Found</tp:message>
<tp:description>The requested resource (//customers-wrong/123) is not
available.</tp:description>
</tp:fault>

```

## Retrieving metadata associated with the state of the resource using the HEAD request method

In this scenario, we are using the http request method HEAD to retrieve the metadata associated with the state of a resource. This is useful in situations where you want to check the content length of the response, last modified date etc. When the HEAD request method is used, the ESB accepts the request and sends it back with only the headers and no body.

### ***Setting up the backend***

For the back-end service, we are using the StockQuote Service that is shipped with ESB. [cURL](#) is used as the REST client to invoke the ESB API. Run [Sample 800](#) as follows.

For further information on how to run an ESB sample, see [Setting Up the ESB Samples](#).

1. Execute one of the following commands to start the ESB with the configuration for sample 800.
  - On Windows: `wso2esb-samples.bat -sn 800`
  - On Linux/Solaris: `./wso2esb-samples.sh -sn 800`
2. Execute one of the following commands to start the the Axis2 server.
  - On Windows: `axis2server.bat`
  - On Linux/Solaris: `./ axis2server.sh`
3. Run `ant` from the `<ESB_HOME>/samples/axis2Server/src/SimpleStockQuoteService` directory.

### ***Configuring the API***

Create an API using the following configuration:

```

<api xmlns="http://ws.apache.org/ns/synapse" name="TestAPI" context="/test">
 <resource methods="POST GET DELETE OPTIONS HEAD">
 <inSequence>
 <property name="AM_REQUEST_METHOD" expression="get-property('axis2',
'HTTP_METHOD')"/></property>
 <send>
 <endpoint>
 <http method="get"
uri-template="http://localhost:9764/StarbucksService/services/Starbucks_Outlet_Service
/orders/"></http>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <filter xpath="get-property('AM_REQUEST_METHOD')='HEAD'">
 <property name="NO_ENTITY_BODY" value="true" scope="axis2"
type="BOOLEAN"/></property>
 </filter>
 <send></send>c
 </outSequence>
 </resource>
</api>

```

### ***Executing the sample***

Send a request to the backend as follows:

```

curl -v -X HEAD
http://localhost:9764/StarbucksService/services/Starbucks_Outlet_Service/orders/

```

To send the above request to the API, the starbucks service should be deployed in WSO2 Application server. See [Building and Running JAX-RS Samples](#) for further information.

You would get the following response:

```

< HTTP/1.1 302 Found
< Set-Cookie: JSESSIONID=32C2FF28247F1826FB0F21B68099D3D1; Path=/; HttpOnly
< Location: https://10.100.5.73:9444/carbon/
< Content-Type: text/html; charset=UTF-8
< Content-Length: 0
< Date: Mon, 09 Jun 2014 05:26:54 GMT
* Server WSO2 Carbon Server is not blacklisted
< Server: WSO2 Carbon Server

```

### ***Sending form data***

In this scenario, an API is used to send form data to a REST service which accepts data of the `x-www-form-urlencoded` content type.

### ***Setting up the backend***

Configure an endpoint as follows for the REST back end service to which the form data should be sent. Note that the endpoint has to be an HTTP endpoint. See [Adding an Endpoint](#) for further instructions.

```
<endpoint xmlns="http://ws.apache.org/ns/synapse" name="FormDataReceiver">
 <http uri-template="http://www.eaipatterns.com/MessageEndpoint.html" method="post">
 <suspendOnFailure>
 <progressionFactor>1.0</progressionFactor>
 </suspendOnFailure>
 <markForSuspension>
 <retriesBeforeSuspension>0</retriesBeforeSuspension>
 <retryDelay>0</retryDelay>
 </markForSuspension>
 </http>
</endpoint>
```

### ***Configuring the API***

Configure the API as follows:

```

<api xmlns="http://ws.apache.org/ns/synapse" name="FORM" context="/Service">
 <resource methods="POST">
 <inSequence>
 <log level="full"></log>
 <property name="name" value="Mark" scope="default" type="STRING"></property>
 <property name="company" value="wso2" scope="default"
type="STRING"></property>
 <property name="country" value="US" scope="default" type="STRING"></property>
 <payloadFactory media-type="xml">
 <format>
 <soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
 <soapenv:Body>
 <root xmlns="">
 <name>$1</name>
 <company>$2</company>
 <country>$3</country>
 </root>
 </soapenv:Body>
 </soapenv:Envelope>
 </format>
 <args>
 <arg evaluator="xml" expression="$ctx:name"></arg>
 <arg evaluator="xml" expression="$ctx:company"></arg>
 <arg evaluator="xml" expression="$ctx:country"></arg>
 </args>
 </payloadFactory>
 <log level="full"></log>
 <property name="messageType" value="application/x-www-form-urlencoded"
scope="axis2" type="STRING"></property>
 <property name="DISABLE_CHUNKING" value="true" scope="axis2"
type="STRING"></property>
 <call>
 <endpoint key="FormDataReceiver"></endpoint>
 </call>
 <respond></respond>
 </inSequence>
</resource>
</api>

```

In this configuration, `name`, `company` and `country` are defined as properties to be sent as form data using the [Property mediator](#). These properties are set as key value pairs by the [PayloadFactory mediator](#) and sent to the ESB.

Then the `messageType` property is set to `application/x-www-form-urlencoded` to enable the ESB to identify these key value pairs as form data. The `DISABLE_CHUNKING` property is set to `true` to remove the chunking from the outgoing message. Then the [Call mediator](#) is used to send the message to the endpoint defined for the REST backend service.

### ***Executing the sample***

Send a request to the backend as follows:

```
curl -v -X POST -d @request -H "Content-Type: text/xml" http://localhost:8280/Service
```

## **Securing APIs**

- Using a Basic Auth handler
- Using an OAuth base security token
  - Creating the custom handler
  - Creating the API
  - Getting the OAuth token
- Using a policy file to authenticate with a secured back-end service
- Transforming Basic Auth to WS-Security

## Using a Basic Auth handler

In most of the real-world use cases of REST, when a consumer attempts to access a privileged resource, credentials must be provided in an Authorization header or the consumer will be refused access. In WSO2 ESB, when we want to secure an API, we can simply make it available via HTTPS and let the security handlers validate the credentials. By default, the ESB does not include any REST Security Handlers, but the following example of a primitive security handler serves as a template that can be used to write your own security handler to secure an API in the ESB.

You add the REST request handler to the `<handlers>` element of the API configuration, as follows (you cannot add it through the API UI):

```

<api name="StockQuoteAPI" context="/stockquotenew"
xmlns="http://ws.apache.org/ns/synapse">
 <resource uri-template="/view/{symbol}" methods="GET" protocol="https">
 <inSequence>
 <payloadFactory>
 <format>
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>$1</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </format>
 <args>
 <arg expression="get-property('uri.var.symbol')"/>
 </args>
 </payloadFactory>
 <header name="Action" value="urn:getQuote"/>
 <send>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"
format="soap11"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </resource>
 <handlers>
 <handler class="org.wso2.rest.BasicAuthHandler"/>
 </handlers>
</api>

```

The custom Basic Auth handler in this sample simply verifies whether the request uses username: admin and password: admin. Following is the code for this handler:

```
package org.wso2.rest;
```

```

import org.apache.commons.codec.binary.Base64;
import org.apache.synapse.MessageContext;
import org.apache.synapse.core.axis2.Axis2MessageContext;
import org.apache.synapse.core.axis2.Axis2Sender;
import org.apache.synapse.rest.Handler;

import java.util.Map;

public class BasicAuthHandler implements Handler {
 public void addProperty(String s, Object o) {
 //To change body of implemented methods use File | Settings | File Templates.
 }

 public Map getProperties() {
 return null; //To change body of implemented methods use File | Settings | File Templates.
 }

 public boolean handleRequest(MessageContext messageContext) {

 org.apache.axis2.context.MessageContext axis2MessageContext
 = ((Axis2MessageContext) messageContext).getAxis2MessageContext();
 Object headers = axis2MessageContext.getProperty(
 org.apache.axis2.context.MessageContext.TRANSPORT_HEADERS);

 if (headers != null && headers instanceof Map) {
 Map headersMap = (Map) headers;
 if (headersMap.get("Authorization") == null) {
 headersMap.clear();
 axis2MessageContext.setProperty("HTTP_SC", "401");
 headersMap.put("WWW-Authenticate", "Basic realm=\"WSO2 ESB\"");
 axis2MessageContext.setProperty("NO_ENTITY_BODY", new
Boolean("true"));
 messageContext.setProperty("RESPONSE", "true");
 messageContext.setTo(null);
 Axis2Sender.sendBack(messageContext);
 return false;
 } else {
 String authHeader = (String) headersMap.get("Authorization");
 String credentials = authHeader.substring(6).trim();
 if (processSecurity(credentials)) {
 return true;
 } else {
 headersMap.clear();
 axis2MessageContext.setProperty("HTTP_SC", "403");
 axis2MessageContext.setProperty("NO_ENTITY_BODY", new
Boolean("true"));
 messageContext.setProperty("RESPONSE", "true");
 messageContext.setTo(null);
 Axis2Sender.sendBack(messageContext);
 }
 }
 }
 }
}

```

```

 return false;
 }
}
return true;
}

public boolean handleResponse(MessageContext messageContext) {
 return true;
}

public boolean processSecurity(String credentials) {
 String decodedCredentials = new String(new
Base64().decode(credentials.getBytes()));
 String userName = decodedCredentials.split(":")[0];
 String password = decodedCredentials.split(":")[1];
 if ("admin".equals(userName) && "admin".equals(password)) {
 return true;
 } else {
 return false;
 }
}
}

```

You can build the project (mvn clean install) for this handler by accessing its source here:

[https://github.com/wso2/product-esb/tree/v5.0.0/modules/samples/integration-scenarios/starbucks\\_sample/BasicAuth-handler](https://github.com/wso2/product-esb/tree/v5.0.0/modules/samples/integration-scenarios/starbucks_sample/BasicAuth-handler)

When building the sample using the source ensure you update `pom.xml` with the online repository. To do this, add the following section before `<dependencies>` tag in `pom.xml`:

```

<repositories>
 <repository>
 <id>wso2-nexus</id>
 <name>WSO2 internal Repository</name>
 <url>http://maven.wso2.org/nexus/content/groups/wso2-public/</url>
 <releases>
 <enabled>true</enabled>
 <updatePolicy>daily</updatePolicy>
 <checksumPolicy>ignore</checksumPolicy>
 </releases>
 </repository>
 <repository>
 <id>wso2-maven2-repository</id>
 <name>WSO2 Maven2 Repository</name>
 <url>http://dist.wso2.org/maven2</url>
 <snapshots>
 <enabled>false</enabled>
 </snapshots>
 <releases>
 <enabled>true</enabled>
 <updatePolicy>never</updatePolicy>
 <checksumPolicy>fail</checksumPolicy>
 </releases>
 </repository>
</repositories>

```

Alternatively, you can download the JAR file from the following location, copy it to the repository/component/lib directory, and restart the ESB:

[https://github.com/wso2/product-esb/blob/v5.0.0/modules/samples/integration-scenarios/starbucks\\_sample/bin/WSO2-REST-BasicAuth-Handler-1.0-SNAPSHOT.jar](https://github.com/wso2/product-esb/blob/v5.0.0/modules/samples/integration-scenarios/starbucks_sample/bin/WSO2-REST-BasicAuth-Handler-1.0-SNAPSHOT.jar)

You can now send a request to the secured API. For example, you can send it using [cURL](#) as the REST client:

```
curl -v -k -H "Authorization: Basic YWRtaW46YWRtaW4=" https://localhost:8243/stockquote/new/IBM
```

### Using an OAuth base security token

You can generate an OAuth base security token using WSO2 Identity Server, and then use that token when invoking your API to connect to a REST endpoint. This approach involves the following tasks:

1. Create a custom handler that will validate the token
2. Create an API that points to the REST endpoint and includes the custom handler
3. Create an OAuth application in Identity Server and get the access token
4. Invoke the API with the access token

#### ***Creating the custom handler***

The custom handler must extend AbstractHandler and implement ManagedLifecycle as shown in the following example. You can download the Maven project for this example at [https://github.com/wso2-docs/ESB/tree/master/ESB-Artifacts/OAuthHandler\\_Sample](https://github.com/wso2-docs/ESB/tree/master/ESB-Artifacts/OAuthHandler_Sample)

```
package org.wso2.handler;
```

```

import org.apache.axis2.AxisFault;
import org.apache.axis2.client.Options;
import org.apache.axis2.client.ServiceClient;
import org.apache.axis2.context.ConfigurationContext;
import org.apache.axis2.context.ConfigurationContextFactory;
import org.apache.axis2.transport.http.HTTPConstants;
import org.apache.axis2.transport.http.HttpTransportProperties;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.http.HttpHeaders;
import org.apache.synapse.core.axis2.Axis2MessageContext;
import org.wso2.carbon.identity.oauth2.stub.OAuth2TokenValidationServiceStub;
import org.wso2.carbon.identity.oauth2.stub.dto.OAuth2TokenValidationRequestDTO;
import org.apache.synapse.ManagedLifecycle;
import org.apache.synapse.MessageContext;
import org.apache.synapse.core.SynapseEnvironment;
import org.apache.synapse.rest.AbstractHandler;
import
org.wso2.carbon.identity.oauth2.stub.dto.OAuth2TokenValidationRequestDTO_OAuth2AccessT
oken;

import java.util.Map;

public class SimpleOauthHandler extends AbstractHandler implements ManagedLifecycle {

 private static final String CONSUMER_KEY_HEADER = "Bearer";
 private static final String OAUTH_HEADER_SPLITTER = ",";
 private static final String CONSUMER_KEY_SEGMENT_DELIMITER = " ";
 private static final String OAUTH_TOKEN_VALIDATOR_SERVICE =
"oauth2TokenValidationService";
 private static final String IDP_LOGIN_USERNAME = "identityServerUserName";
 private static final String IDP_LOGIN_PASSWORD = "identityServerPw";
 private ConfigurationContext configContext;
 private static final Log log = LogFactory.getLog(SimpleOauthHandler.class);

 @Override
 public boolean handleRequest(MessageContext msgCtx) {
 if (this.getConfigContext() == null) {
 log.error("Configuration Context is null");
 return false;
 }
 try{
 //Read parameters from axis2.xml
 String identityServerUrl =
 msgCtx.getConfiguration().getAxisConfiguration().getParameter(
 OAUTH_TOKEN_VALIDATOR_SERVICE).getValue().toString();
 String username =
 msgCtx.getConfiguration().getAxisConfiguration().getParameter(
 IDP_LOGIN_USERNAME).getValue().toString();
 String password =
 msgCtx.getConfiguration().getAxisConfiguration().getParameter(
 IDP_LOGIN_PASSWORD).getValue().toString();
 OAuth2TokenValidationServiceStub stub =
 new OAuth2TokenValidationServiceStub(this.getConfigContext(),
identityServerUrl);
 ServiceClient client = stub._getServiceClient();
 Options options = client.getOptions();
 HttpTransportProperties.Authenticator authenticator = new
HttpTransportProperties.Authenticator();
 }
 }
}

```

```

 authenticator.setUsername(username);
 authenticator.setPassword(password);
 authenticator.setPreemptiveAuthentication(true);
 options.setProperty(HTTPConstants.AUTHENTICATE, authenticator);
 client.setOptions(options);
 OAuth2TokenValidationRequestDTO dto =
this.createOAuthValidatorDTO(msgCtx);
 return stub.validate(dto).getValid();
 }catch(Exception e){
 log.error("Error occurred while processing the message", e);
 return false;
 }
}
private OAuth2TokenValidationRequestDTO createOAuthValidatorDTO(MessageContext msgCtx) {
 OAuth2TokenValidationRequestDTO dto = new OAuth2TokenValidationRequestDTO();
 Map headers = ((Map) ((Axis2MessageContext) msgCtx).getAxis2MessageContext().

getProperty(org.apache.axis2.context.MessageContext.TRANSPORT_HEADERS);
 String apiKey = null;
 if (headers != null) {
 apiKey = extractCustomerKeyFromAuthHeader(headers);
 }
 OAuth2TokenValidationRequestDTO_OAuth2AccessToken token =
 new OAuth2TokenValidationRequestDTO_OAuth2AccessToken();
 token.setTokenType("bearer");
 token.setIdentifier(apiKey);
 dto.setAccessToken(token);
 return dto;
}
private String extractCustomerKeyFromAuthHeader(Map headersMap) {
 //From 1.0.7 version of this component onwards remove the OAuth authorization
header from
 // the message is configurable. So we dont need to remove headers at this
point.
 String authHeader = (String) headersMap.get(HttpHeaders.AUTHORIZATION);
 if (authHeader == null) {
 return null;
 }
 if (authHeader.startsWith("OAuth ") || authHeader.startsWith("oauth ")) {
 authHeader = authHeader.substring(authHeader.indexOf("o"));
 }
 String[] headers = authHeader.split(OAUTH_HEADER_SPLITTER);
 if (headers != null) {
 for (String header : headers) {
 String[] elements = header.split(CONSUMER_KEY_SEGMENT_DELIMITER);
 if (elements != null && elements.length > 1) {
 boolean isConsumerKeyHeaderAvailable = false;
 for (String element : elements) {
 if (!"".equals(element.trim())) {
 if (CONSUMER_KEY_HEADER.equals(element.trim())) {
 isConsumerKeyHeaderAvailable = true;
 } else if (isConsumerKeyHeaderAvailable) {
 return removeLeadingAndTrailing(element.trim());
 }
 }
 }
 }
 }
 }
}

```

```
 }
 return null;
 }
 private String removeLeadingAndTrailing(String base) {
 String result = base;
 if (base.startsWith("\\")) || base.endsWith("\\")) {
 result = base.replace("\\", "");
 }
 return result.trim();
 }
 @Override
 public boolean handleResponse(MessageContext messageContext) {
 return true;
 }
 @Override
 public void init(SynapseEnvironment synapseEnvironment) {
 try {
 this.configContext =
ConfigurationContextFactory.createConfigurationContextFromFileSystem(null, null);
 } catch (AxisFault axisFault) {
 log.error("Error occurred while initializing Configuration Context",
axisFault);
 }
 }
 @Override
 public void destroy() {
 this.configContext = null;
 }
 private ConfigurationContext getConfigContext() {
```

```

 return configContext;
 }
}

```

## **Creating the API**

You will now create an API named `TestGoogle` that connects to the following endpoint: `https://www.google.lk/search?q=wso2`

1. In the ESB Management Console, go to Manage -> Service Bus and click **Source View**.
2. Insert the following XML configuration into the source view before the closing `</definitions>` tag to create the `TestGoogle` API:

```

<api xmlns="http://ws.apache.org/ns/synapse"
 name="TestGoogle"
 context="/search">
 <resource methods="GET">
 <inSequence>
 <log level="full">
 <property name="STATUS" value="***** REQUEST HITS IN SEQUENCE *****"/>
 </log>
 <send>
 <endpoint>
 <http method="get"
uri-template="https://www.google.lk/search?q=wso2"/>
 </endpoint>
 </send>
 </inSequence>
 </resource>
 <handlers>
 <handler class="org.wso2.handler.SimpleOAuthHandler"/>
 </handlers>
</api>

```

Notice that the `<handlers>` tag contains the reference to the custom handler class.

3. Copy the custom `handler.jar` to the `<ESB_HOME>/repository/components/libs` directory.
4. Open `<ESB_HOME>/repository/conf/axis2/axis2.xml` and add the following parameters:

```

<!-- OAuth2 Token Validation Service -->
<parameter
name="oauth2TokenValidationService">https://localhost:9444/services/OAuth2TokenValidationService</parameter>
<!-- Server credentials -->
<parameter name="identityServerUserName">admin</parameter>
<parameter name="identityServerPw">admin</parameter>

```

5. Restart the ESB.

## **Getting the OAuth token**

You will now use Identity Server to create an OAuth application and generate the security token.

1. Start WSO2 Identity Server and log into the management console.
2. On the **Main** tab, click **Add** under **Service Providers**, and then add a service provider.

3. Note the access token URL and embed it in a cURL request to get the token. For example, use the following command and replace <client-id> and <client secret> with the actual values:

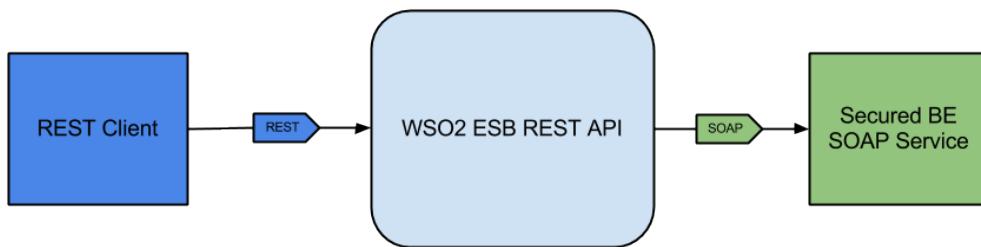
```
curl -v -k -X POST --user <client-id>:<client secret> -H "Content-Type: application/x-www-form-urlencoded; charset=UTF-8" -d 'grant_type=password&username=admin&password=admin' https://localhost:9444/oauth2/token
```

Identity Server returns the access token, which you can now use when invoking the API. For example:

```
curl -v -X GET -H "Authorization: Bearer ca1799fc84986bd87c120ba499838a7" http://localhost:8280/search
```

### **Using a policy file to authenticate with a secured back-end service**

You can connect a REST client to a secured back-end service (such as a SOAP service) through an API that reads from a policy file.



First, you configure the ESB to expose the API to the REST client. For example:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <localEntry key="sec_policy"
 src="file:repository/samples/resources/policy/policy_3.xml"/>
 <api name="StockQuoteAPI" context="/stockquote">
 <resource methods="GET" uri-template="/view/{symbol}">
 <inSequence trace="enable">
 <header name="Action" value="urn:getQuote"/>
 <payloadFactory>
 <format>
 <m0:quote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>$1</m0:symbol>
 </m0:request>
 </m0:quote>
 </format>
 <args>
 <arg expression="get-property('uri.var.symbol')"/>
 </args>
 </payloadFactory>
 <send>
 <endpoint name="rest">
 <address
uri="http://localhost:9000/services/SecureStockQuoteService"
 format="soap11">
 <enableAddressing/>
 <enableSecurity policy="sec_policy" />
 </address>
 </endpoint>
 </send>
 </inSequence>
 <outSequence trace="enable">
 <header
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
 name="wsse:Security"
 action="remove"/>
 <send/>
 </outSequence>
 </resource>
 <resource methods="POST">
 <inSequence>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
 <property name="OUT_ONLY" value="true"/>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"
 format="soap11"/>
 </endpoint>
 </send>
 </inSequence>
 </resource>
 </api>
</definitions>

```

The policy file stores the security parameters that are going to be authenticated by the back-end service. You specify

the policy file with the localEntry property, which in this example we've named sec\_policy:

```
<localEntry key="sec_policy"
 src="file:repository/samples/resources/policy/policy_3.xml" />
```

You use then reference the policy file by its localEntry name when enabling the security policy for the end point:

```
<address uri="http://localhost:9000/services/SecureStockQuoteService"
 format="soap11">
 <enableAddressing/>
 <enableSec policy="sec_policy" />
</address>
```

In the outSequence property, the security header must be removed before sending the response back to the REST client.

```
<outSequence trace="enable">
 <header
 xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
 name="wsse:Security"
 action="remove" />
```

To test this API configuration, you must run the SecureStockQuoteService, which is bundled in the WSO2 ESB samples folder, as the back-end server. Start this sample as described in [Setting Up the ESB Samples](#). Because this sample uses Apache Rampart for the back-end security implementation, you might also need to download and install the unlimited strength policy files for your JDK before using Apache Rampart.

#### To download the unlimited strength policy files:

1. Go to <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.
2. Scroll to the bottom of the page and download the file called Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files x (x is the JDK version) for your JDK version.
3. Extract the downloaded ZIP. You'll now have two JAR files: `local_policy.jar` and `US_export_policy.jar`.
4. In your Java installation directory, go to the `jre/lib/security` directory, such as: `/usr/java/jdk1.6.0_38/jre/lib/security`
5. Make a backup of the files `local_policy.jar` and `US_export_policy.jar` and then replace them with the ones from the JCE ZIP file.

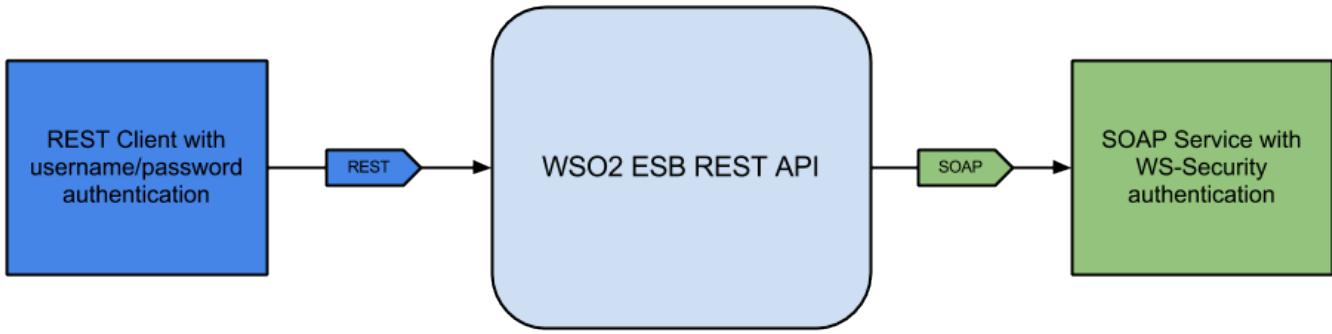
Now that you have set up the API and the secured back-end SOAP service, you are ready to test this configuration with the following curl command.

```
curl -v http://127.0.0.1:8280/stockquote/view/IBM
```

Observe that the REST client is getting the correct response (the `wsse:Security` header is removed from the decrypted message) while going through the secured back-end service and the API implemented in the ESB. You can use a TCP monitoring tool such as `tcpmon` to monitor the message sent from the ESB and the response message received by the ESB. For a tutorial on using `tcpmon`, see: <http://technonstop.com/tcpmon-tutorial>

#### Transforming Basic Auth to WS-Security

REST clients typically use Basic Auth over HTTP to authenticate the user name and password, but if the back-end service uses WS-Security, you can configure the API to transform the authentication from Basic Auth to WS-Security.



To achieve this transformation, you configure the ESB to expose the API to the REST client as shown in the [previous example](#), but you add the **HTTPS** protocol and **Basic Auth handler** to the configuration as shown below:

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <localEntry key="sec_policy"
src="file:repository/samples/resources/policy/policy_3.xml"/>
 <api name="StockQuoteAPI" context="/stockquote">
 <resource methods="GET" uri-template="/view/{symbol}" protocol="https" >
 ...
 </resource>
 <handlers>
 <handler class="org.wso2.rest.BasicAuthHandler"/>
 </handlers>
 </api>
</definitions>

```

This configuration allows two-fold security: the client authenticates with the API using Basic Auth over HTTPS, and then the API sends the request to the back-end service using the security policy.

You can test this configuration using the following command:

```

curl -v -k -H "Authorization: Basic YWRtaW46YWRtaW4="
https://localhost:8243/stockquote/view/IBM

```

## Unusual Scenarios for HTTP Methods in REST

When sending REST messages to an API, you typically use POST or PUT to send a message and GET to request a message. However, there are some unusual scenarios you might want to support, which are described in the following sections:

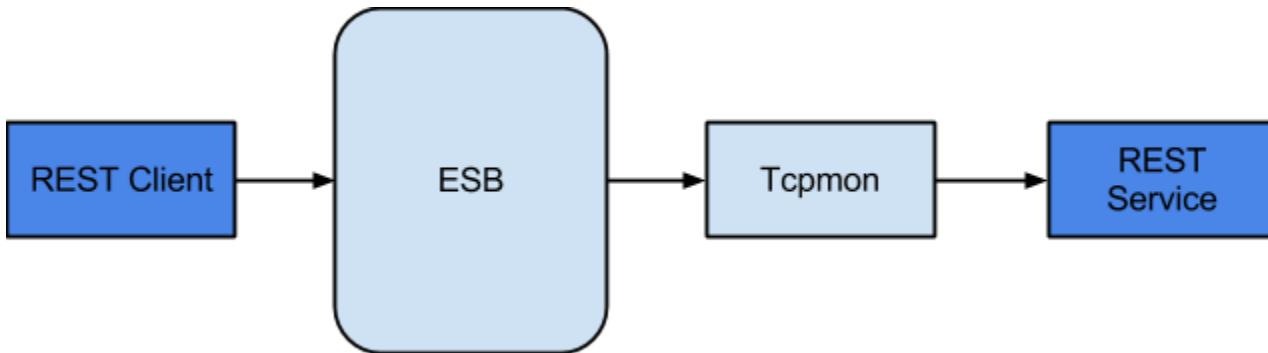
- Using POST with No Body
- Using POST with Query Parameters

- Using GET with a Body

### Using POST with No Body

Typically, POST is used to send a message that has data enclosed as a payload inside an HTML body. However, you can also use POST without a payload. WSO2 ESB treats it as a normal message and forwards it to the endpoint without any extra configuration.

The following diagram depicts a scenario where a REST client communicates with a REST service using WSO2 ESB. Apache Tcmpon is used solely for monitoring the communication between the ESB and the back-end service and has no impact on the messages passed between the ESB and back-end service. For this particular scenario, the cURL client is used as the REST client, and the [basic JAX-RS sample](#) (available with [WSO2 Application Server](#)) is used as the back-end REST service.



To implement this scenario:

1. Configure and deploy the JAX-RS Basics sample by following the instructions for building and running the sample on the [JAX-RS Basics](#) page.
2. Since we are going to run both WSO2 ESB and the WSO2 Application Server on the same machine, offset the ports of the application server by changing the offset value to 1 in `<AS_HOME>/repository/conf/carbon.xml`.
3. Start the ESB and change the configuration as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default sequence"/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE"
expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
 </sequence>
 <sequence name="main">
 <log/>
 <drop/>
 </sequence>
 <api name="testAPI" context="/customerservice">
 <resource methods="POST" url-mapping="/customers">
 <inSequence>
 <send>
 <endpoint>
 <address
uri="http://localhost:8282/jaxrs_basic/services/customers/customerservice"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </resource>
 </api>
</definitions>

```

In this proxy configuration, testAPI intercepts messages that are sent to the relative URL /customerservice/customers and sends them to the relevant endpoint by appending the url-mapping of the resource tag to the end of the endpoint URL.

4. Start tcpmon and make it listen to port 8282 of your local machine. It is also important to set the target host name and the port as required.

We will now test the connection by sending a POST message that includes a payload inside an HTML body.

5. Go to the terminal and issue the following command:

```
curl -v -H "Content-Type: application/xml" -d
"<Customer><id>123</id><name>John</name></Customer>" http://localhost:8280/customerservice/customers
```

6. The following reply message appears in the console:

```

<Customer>
 <id>132</id>
 <name>John</name>
</Customer>

```

7. Now send the same POST message but without the enclosed data:

```
curl -v -H "Content-Type: application/xml" -d "" http://localhost:8280/customerservice/customer
```

The tcpmon output shows the same REST request that was sent by the client, demonstrating that the ESB handled

the POST message regardless of whether it included a payload. However, you would need to configure the back-end service to handle such requests, or the application server will throw exceptions.

### Using POST with Query Parameters

Sending a POST message with query parameters is an unusual scenario, but the ESB supports it with no additional configuration. The ESB forwards the message like any other POST message and includes the query parameters.

To test this scenario, use the same setup as above, but instead of removing the data part from the request, add some query parameters to the URL as follows:

```
curl -v -H "Content-Type: application/xml" -d
"<Customer><id>123</id><name>John</name></Customer>" 'http://localhost:8280/customers
ervice/customers?param1=value1¶m2=value2'
```

When you execute this command, you can see the following output in tcpmon:

```
POST
/jaxrs_basic/services/customers/customerservice/customers?param1=value1¶m2=value2
HTTP/1.1
Content-Type: application/xml; charset=UTF-8
Accept: /*
Transfer-Encoding: chunked
Host: 127.0.0.1:8282
Connection: Keep-Alive

32
<Customer><id>123</id><name>John</name></Customer>
0
```

As you can see, the query parameters are present in the REST request, demonstrating that the ESB sent them along with the message. You could write resource methods to support this type of a request. In this example, the resource method accessed by this request simply ignores the parameters.

### Using GET with a Body

Typically, a GET request does not contain a body, and WSO2 ESB does not support these types of requests. When it receives a GET request that contains a body, it drops the message body as it sends the message to the endpoint. You can test this scenario using the same setup as above, but this time the client command should look like this:

```
curl -v -H "Content-Type: application/xml" -d
"<Customer><id>123</id><name>John</name></Customer>" 'http://localhost:8280/customers
ervice/customers' -X GET
```

The additional parameter `-X` replaces the original POST method with the specified method, which in this case is GET. This will cause the client to send a GET request with a message similar to a POST request. If you view the output in tcpmon, you will see that there is no message body in the request.

### Using REST with APIs

You can create APIs that allow messages to be sent over the HTTP and HTTPS transports. For more information on creating APIs, see [Working with APIs](#).

You can configure REST endpoints in an API by directly specifying HTTP verbs (such as POST and GET), URI templates, and URL mappings. Alternatively, you can use the [HTTP Endpoint](#) to define REST endpoints using URI templates.

## Per-API Logs in WSO2 ESB

The advantage of having per-API log files is that it is very easy to analyze/monitor what went wrong in a particular REST API defined in the ESB by looking at the log files. The API log is an additional log file, which will contain a copy of the logs to a particular REST API.

Below are the configuration details to configure the logs of a REST API called `TestAPI` using `log4j` properties.

Open `<ESB_HOME>/repository/conf/log4j.properties` file using your favorite text editor to configure `log4j` to log the API specific logs to a file. You can configure the logger for either **INFO level logs** or **DEBUG level logs** as follows:

Add the following section to the end of the file to configure the logger for log messages where the Log Category is **I NFO**.

```
log4j.category.API_LOGGER=INFO, API_APPENDER
log4j.additivity.API_LOGGER=false
log4j.appenders.API_APPENDER=org.apache.log4j.RollingFileAppender
log4j.appenders.API_APPENDER.File=${carbon.home}/repository/logs/${instance.log}/wso2-eb-sb-api${instance.log}.log
log4j.appenders.API_APPENDER.MaxFileSize=1000KB
log4j.appenders.API_APPENDER.MaxBackupIndex=10
log4j.appenders.API_APPENDER.layout=org.apache.log4j.PatternLayout
log4j.appenders.API_APPENDER.layout.ConversionPattern=%d{ISO8601} [%X{ip}-%X{host}]
[%t] %5p %c{1} %m%n</pre>
```

The in-sequence of the REST API will need to contain a **Log mediator** with the Log Category defined as **INF O** to be able to view logs in the log file.

### DEBUG level

Add the following section to the end of the file to configure the logger for log messages where the Log Category is **D EBUG**.

```
log4j.category.API_LOGGER.TestAPI=DEBUG, TEST_API_APPENDER
log4j.additivity.API_LOGGER.TestAPI=false
log4j.appenders.TEST_API_APPENDER=org.apache.log4j.DailyRollingFileAppender
log4j.appenders.TEST_API_APPENDER.File=${carbon.home}/repository/logs/${instance.log}/TestAPI.log
log4j.appenders.TEST_API_APPENDER.datePattern='yyyy-MM-dd-HH-mm
log4j.appenders.TEST_API_APPENDER.layout=org.apache.log4j.PatternLayout
log4j.appenders.TEST_API_APPENDER.layout.ConversionPattern=%d{ISO8601}
[%X{ip}-%X{host}] [%t] %5p %c{1} %m%n
```

The above configuration creates a log file names `TestAPI.log` in the folder `<ESB_HOME>/repository/logs`.

The in-sequence of the REST API will need to contain a **Log mediator** with the Log Category defined as **DE BUG** to be able to view logs in the log file.

## Working with Proxy Services

A **proxy service** is a virtual service that receives messages and optionally processes them before forwarding them to a service at a given [endpoint](#). This approach allows you to perform necessary transformations and introduce additional functionality without changing your existing service. For example, if you want to use WS-Security with an existing, unsecured service, you can create a secure proxy service with WS-Security enabled with a specified security policy. Additionally, if you want a service to handle messages that are in different formats, you can create a Transformer proxy service to transform requests and responses based on specified XSLT configurations. A proxy service can also switch [transports](#), process the messages with [mediation sequences](#) and [tasks](#), and terminate the flow or send a message back to the client even without sending it to the actual service.

## Proxy Service vs. Proxy Server

Do not confuse a proxy **service**, which processes messages on their way to a service, with a proxy **server**, which acts as an intermediary for handling traffic to a server. For information on how to configure WSO2 ESB to route messages through a proxy server, see [Working with Proxy Servers](#).

You can create the following types of proxy services:

- **Pass Through Proxy** - Forwards messages to the endpoint without performing any processing on them. This proxy service is useful as a catch-all, so that messages that do not meet the criteria to be handled by other proxy services are simply forwarded to the endpoint.
- **Secure Proxy** - Uses WS-Security to process incoming requests and forward them to an unsecured backend service.
- **WSDL Based Proxy** - A proxy service that is created from the remotely hosted WSDL of an existing web service. The endpoint information is extracted from the WSDL.
- **Logging Proxy** - Logs all the incoming requests and forwards them to a given endpoint. It can also log responses from the backend service before routing them to the client.
- **Transformer Proxy** - Transforms all the incoming requests using XSLT and then forwards them to a given endpoint. It can also transform responses from the backend service.
- **Custom Proxy** - A custom proxy service in which you customize the sequences, endpoints, transports, and other QoS settings.

The simplest way to create a proxy service is through the management console, as described in [Adding a Proxy Service](#). You can then use the console to manage the proxy service (edit it, redeploy it, or disable its statistics or tracing) as described in [Managing Proxy Services](#).

You can also configure proxy services manually in XML. Following is an example of manually configuring a Synapse proxy service. For additional samples of manual proxy service configurations, see [Proxy Service Samples](#).

The `<proxy>` element defines the proxy service, as shown below.

```

<proxy name="string" [transports="(http | https | jms |..)+|all"] [pinnedServers="(serverName)+"] [serviceGroup="string"]>
 <description>...</description>
 <target [inSequence="name"] [outSequence="name"] [faultSequence="name"] [endpoint="name"]>
 <inSequence>...</inSequence>?
 <outSequence>...</outSequence>?
 <faultSequence>...</faultSequence>?
 <endpoint>...</endpoint>?
 </target>?
 <publishWSDL key="string" uri="string">
 (<wsdl:definition>...</wsdl:definition> |
 <wsdl120:description>...</wsdl120:description>)?
 <resource location="..." key="..."/>*
 </publishWSDL>?
 <enableAddressing/>?
 <enableSec/>?
 <enableRM/>?
 <policy key="string" [type="(in | out)"]/>? // optional service or message
level policies such as (e.g. WS-Security and/or WS-RM policies)
 <parameter name="string"> // optional service parameters such as
(e.g. transport.jms.ConnectionFactory)
 string | xml
 </parameter>
 </proxy>

```

A proxy service is created and exposed on the specified transports through the underlying Axis2 engine. The proxy service can be a SOAP or REST/POX service over HTTP/S or SOAP, POX, Plain Text, or Binary / Legacy service for other transports such as JMS and VFS file systems. It exposes service EPRs as per the standard Axis2 conventions based on the service name.

## Note

Axis2 does not allow custom URIs to be set for services on some transports such as HTTP/S. The proxy service could be exposed over all enabled Axis2 transports such as HTTP, HTTPS, JMS, Mail and File etc. or on a subset of these as specified with the optional transports attribute.

You can use the `pinnedServers` attribute to specify the list of Synapse servers where this proxy service should be deployed. If there is no pinned server list, the proxy service is started in all server instances. The `pinnedServers` attribute takes the Synapse server names separated by commas or spaces. The Synapse server name is specified in the system property `SynapseServerName` or through the `axis2.xml` parameter `SynapseConfig.ServerName`. If neither of these are set, the server hostname is used, or it defaults to `localhost`. You can give a name to a Synapse server instance by specifying the property `-DSynapseServerName=<ServerName>` when you execute the startup script `wso2server.bat` or `wso2server.sh`, or by editing `wrapper.conf` to do this where Synapse is started as a service.

Each service could define the target for received messages as a named sequence or a direct endpoint. Target `inSequence` or `endpoint` is required for the proxy configuration, and a target `outSequence` defines how responses should be handled. Any supplied WS-Policies apply as service-level policies, and any service parameters can be passed into the proxy services' `AxisService` instance using the `parameter` elements (for example, the JMS destination). If the proxy service should enable Security, the appropriate modules could be engaged, and specified service level policies will apply.

To create a dynamic proxy, specify the properties of the proxy as dynamic entries by referring to them with the `key`

attribute. For example, you could specify the `inSequence` or `endpoint` with a remote key instead of defining it in the local configuration. As the remote registry entry changes, the properties of the proxy are updated accordingly.

## Note

The proxy service definition itself cannot be specified dynamically. That is, you cannot specify `<proxy key="string"/>`.

You can publish a WSDL for the proxy service using the `<publishWSDL>` element. The WSDL document can be loaded from the registry by specifying the `key` attribute or from any other location by specifying the `uri` attribute. Alternatively, you can provide the WSDL inline as a child element of `<publishWSDL>`. Artifacts (schemas or other WSDL documents) imported by the WSDL can be resolved from the registry by specifying appropriate `<resource>` elements:

```
<publishWSDL key="my.wsdl">
 <resource location="http://www.standards.org/standard.wsdl" key="standard.wsdl"/>
</publishWSDL>
```

In this example, the WSDL is retrieved from the registry using the key `my.wsdl`. It imports another WSDL from `http://www.standards.org/standard.wsdl`, but instead of loading it from this location, Synapse retrieves the imported WSDL from the registry entry `standard.wsdl`.

Following are additional service parameters you can use:

Parameter	Value	Default	Description
<code>useOriginalwsdl</code>	true / false	false	If this parameter is set to <code>true</code> , the original WSDL published via the <code>publishWSDL</code> parameter is used. This WSDL would not be replaced by a WSDL generated by the proxy service.
<code>modifyUserWSDLPortAddress</code>	true / false	true	(Effective only with <code>useOriginalwsdl=true</code> ) If true (default) modify the port addresses to current host.
<code>ApplicationXMLBuilder.allowDTD</code>	true / false	true	If this parameter is set to <code>true</code> , it enables data type definition processing for the proxy service. Data type definition processing is disabled in the Axis2 engine due to security vulnerability. This parameter enables it for individual proxy services.
<code>enablePublishWSDLSafeMode</code>	true / false	true	When creating a proxy, if this parameter is set to <code>true</code> , it prevents the proxy service deployment from failing when the wsdl is not available at server startup.
<code>showAbsoluteSchemaURL</code>	true/false	true	If this parameter is set to <code>true</code> , the absolute path of the referred schemas of the WSDL is shown instead of the relative paths.
<code>showProxySchemaURL</code>	true/false		If this parameter is set to <code>true</code> , the full proxy URL will be set as the prefix to the schema location of the imports in proxy WSDL.

The following service parameters are for specific transports:

Transport	Require	Parameter	Description
JMS	Optional	transport.jms.ConnectionFactory	The JMS connection factory definition (from axis2.xml) to be used to listen for messages for this service.
	Optional	transport.jms.Destination	The JMS destination name (Defaults to the service name).
	Optional	transport.jms.DestinationType	The JMS destination type. Accept values "queue" or "topic."
	Optional	transport.jms.ReplyDestination	The destination where a reply will be posted.
	Optional	transport.jms.Wrapper	The wrapper element for the JMS message.
	Required	transportvfs.FileURI	The primary File (or Directory) URI in the vfs* transport format, for this service.
VFS	Required	transportvfs.ContentType	<p>The expected content type for files retrieved for this service. The VFS transport uses this information to select the appropriate message builder.</p> <p>Examples:</p> <ul style="list-style-type: none"> <li>text/xml for plain XML or SOAP</li> <li>text/plain; charset=ISO-8859-1 for text files</li> <li>application/octet-stream for binary data</li> </ul>
	Optional	transportvfs.FileNamePattern	A file name regex pattern to match files against a directory specified by the FileURI.
	Optional	transport.PollInterval	The poll interval (in seconds).
	Optional	transportvfs.ActionAfterProcess	DELETE or MOVE.
	Optional	transportvfs.MoveAfterProcess	The directory to move files after processing (i.e. all files process successfully).
	Optional	transportvfs.ActionAfterErrors	DELETE or MOVE.
	Optional	transportvfs.MoveAfterErrors	The directory to move files after errors (i.e. some of the files succeed but some fail).
	Optional	transportvfs.ActionAfterFailure	DELETE or MOVE.
	Optional	transportvfs.MoveAfterFailure	The directory to move after failure (i.e. all files fail).
	Optional	transportvfs ReplyFileURI	Reply file URI.

	Optional transport.vfs. ReplyFileName	Reply file name (defaults to response XML).
	Optional transport.vfs. MoveTimestampFormat	Timestamp prefix format for processed file name. java.text.SimpleDateFormat compatible string. e.g. yyMMddHHmmss'-'.

## Working with Proxy Services via WSO2 ESB Tooling

You can create a new proxy service or import an existing proxy service from an XML file, such as a Synapse configuration file using WSO2 ESB tooling.

You need to have WSO2 ESB tooling installed to create a new inbound endpoint or to import an existing inbound endpoint via ESB tooling. For instructions on installing WSO2 ESB tooling, see [Installing WSO2 ESB Tooling](#).

### ***Creating a new proxy service***

Follow the steps below to create a proxy service.

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Proxy Service** on the **Developer Studio Dashboard**.
3. Select **Create a New Proxy Service** and click **Next**.
4. Type a unique name for the proxy service and specify the proxy type (see below).
5. Do one of the following:
  - To save the proxy service in an existing ESB Config project in your workspace, click **Browse** and select that project.
  - To save the proxy service in a new ESB Config project, click **Create new ESB Project** and create the new project.
6. If you specified a proxy type that requires that you enter the target endpoint (the endpoint that represents the actual service), do one of the following:
  - If you know the URL of the endpoint, select **Enter URL** and type it in the text box.
  - If you want to use an endpoint you've already defined in this workspace, select **Predefined Endpoint** and select it from the list.
  - If you want to use an endpoint in the registry, select **Pick from Registry**, and then either type the endpoint's registry key or click **Browse**, click **Registry**, and navigate to the endpoint in the registry.
7. Fill in the advanced configuration based on the proxy service type you specified:
  - **Transformer Proxy:** Transforms all the incoming requests using XSLT and then forwards them to a given target endpoint. Specify the target endpoint as described in the previous step, and then specify the location of the XSLT you want to use to transform requests, either by typing the path or by clicking **Browse** and navigating to the XSLT, which can be a file in the workspace or registry or can be a local entry. If you also want to transform the responses from the backend service, click **Transform Responses**.
  - **Log Forward Proxy:** Logs all the incoming requests and forwards them to a given endpoint. It can also log responses from the backend service before routing them to the client. Specify the log level for requests and responses, where **Simple** logs To, From, WSAction, SOAPAction, ReplyTo, MessageID, and any properties, and **Full** logs all attributes of the message plus the SOAP envelope information.
  - **Pass Through Proxy:** Forwards messages to the endpoint without performing any processing on them. This proxy service is useful as a catch-all, so that messages that do not meet the criteria to be handled by other proxy services are simply forwarded to the endpoint. When you select this proxy service type, you just specify the target endpoint as described in the previous step.
  - **WSDL Based Proxy:** A proxy service that is created from the remotely hosted WSDL of an existing web service. The endpoint information is extracted from the WSDL. In the **URI** field, enter the URL and URN of the WSDL. The URL defines the host address of the network resource (can be omitted if

resources are not network homed), and the URN defines the resource name in local namespaces. For example, if the URL is `ftp://ftp.dlink.ru` and the URN is `/pub/ADSL/`, you would enter `ftp://ftp.dlink.ru/pub/ADSL/` for the URI. To ensure that the URI is valid, click **Test URI**. You then enter the service name and port of the WSDL. Lastly, if you want to publish this WSDL, click **Publish Same Service Contract**.

- **Secure Proxy:** Uses WS-Security to process incoming requests and forward them to an unsecured backend service. Specify the target endpoint as described in the previous step, and then specify the key of the security policy or click **Browse** and select it from the registry.
  - **Custom Proxy:** A custom proxy service in which you customize all the sequences, endpoints, transports, and other QoS settings by adding them to the mediation workflow after the proxy service is created.
8. Click **Finish**. The proxy service is created in the `src/main/synapse-config/proxy-service` folder under the ESB Config Project you specified, and the proxy service appears in the editor. Click its icon in the editor to view its properties.

### **Importing a proxy service**

Follow these steps to import an existing proxy service from an XML file (such as a Synapse configuration file) into an ESB Config project.

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Proxy Service** on the **Developer Studio Dashboard**.
3. Select **Import Proxy Service** and click **Next**.
4. Specify the proxy service file by typing its full pathname or clicking **Browse** and navigating to the file.
5. In the **Save Proxy Service In** field, specify an existing ESB Config project in your workspace where you want to save the proxy service, or click **Create new ESB Project** to create a new ESB Config project and save the proxy service there.
6. If there are multiple proxy services in the file, in the **Advanced Configuration** section select the proxy services you want to import.
7. Click **Finish**. The proxy services you selected are created in the `src/main/synapse-config/proxy-service` folder under the ESB Config project you specified, and the first proxy service appears in the editor.

### **Working with Proxy Services via the Management Console**

You can add, edit as well as set up specific configurations for a proxy service via the ESB Management Console.

See the following topics for information on how to manage proxy services via the Management Console:

- [Adding a Proxy Service](#)
- [Managing Proxy Services](#)

#### **Adding a Proxy Service**

A proxy service defines virtual services hosted on the ESB that can accept requests, mediate them, and deliver them to an actual service. A proxy service can perform transport or interface switching and expose different semantics like WSDL, policies, and QoS aspects like WS-Security from the actual service. It can mediate the messages before they are delivered to the actual **endpoint** and mediate the responses before they reach the client. You can also list a combination of **tasks** to be performed on the messages received by the proxy service and terminate the flow or you can send a message back to the client even without sending it to the actual service.

#### **To add a new proxy service**

1. Click the **Main** tab on the Management Console. Go to **Manage -> Services -> Add** and then click **Proxy Service**.



The **Create Proxy Service from Template** screen appears.

2. Select a template that suits your mediation requirement. For example, if you want to expose an existing service with WS-Security, you can choose the "Secure Proxy" option to create a proxy service with WS-Security enabled with a specified security policy. If you implement a scenario where an existing service is exposed over a different schema (message format), you can use the "Transformer Proxy" option to easily set up a proxy service that transforms requests and responses based on specified XSLT configurations.

The available templates are as follows:

- **Pass Through Proxy** - Creates a simple proxy service on a specified endpoint. The proxy service does not perform any processing on the messages but simply forwards them to the back-end service.
- **Secure Proxy** - Creates a proxy service with WS-Security engaged. The service will process WS-Security on incoming requests and forward them to an unsecured back-end service.
- **WSDL Based Proxy** - Creates a proxy service based on the WSDL of an existing web service. Endpoint information is extracted from the remotely hosted WSDL of an actual service.
- **Logging Proxy** - Creates a proxy service that logs all the incoming requests and forwards them to a given endpoint. Responses from the back-end service can also be logged before routing them to the client.
- **Transformer Proxy** - Creates a proxy service, which transforms all the incoming requests using XSLT and then forwards them to a given endpoint. Responses from the back-end service can also be transformed before routing them to the client.
- **Custom Proxy** - Launches the proxy service creation wizard, where you create a new proxy service by customising every aspect of the proxy including [sequences](#), [endpoints](#), [transport](#), and other QoS settings.

3. Specify the options for the selected template as described in the following topics:

- [Pass Through Proxy Template](#)
- [Secure Proxy Template](#)
- [WSDL Based Proxy Template](#)
- [Logging Proxy Template](#)
- [Transformer Proxy Template](#)
- [Custom Proxy Template](#)

4. Specify an option for publishing a WSDL for the proxy service.

5. Select the transports that the proxy service will use. Here you will only see the transports that are enabled.

6. Click **Create**. The proxy service will be created and should appear in the **Services** list on the **Deployed Services** screen.

Deployed Services					
5 active services. 5 deployed service group(s).					
Service Type <input type="button" value="ALL"/> Service <input type="text" value=""/> <input type="button" value="Search"/>					
<a href="#">Select all in this page</a>   <a href="#">Select none</a>		<input type="button" value="Delete"/>			
Services					
<input type="checkbox"/>	echo				
<input type="checkbox"/>	Proxy Service				
<input type="checkbox"/>	Version				
	wso2carbon-sts				
	XKMS				
<a href="#">Select all in this page</a>   <a href="#">Select none</a>		<input type="button" value="Delete"/>			

### Custom Proxy Template

The "Custom Proxy" template allows you to create a new proxy service by specifying all the settings through a three-step wizard. You customize every aspect of the proxy including sequences, endpoints, transports, and other QoS settings.

#### Step 1

#### Step 2

#### Step 3

Step 1

Design switch to source view

### Step 1 of 3 - Basic Settings

Proxy Service Name\*

**General Settings**

Publishing WSDL

Service Parameters

Service Group

Do Not load service on startup

Pinned servers (separated by comma or space)

**Transport Settings**

http

https

The Proxy Service settings:

- **Proxy Service Name** - Allows to define the unique name of the Proxy Service.
- **General Settings**
  - **Publishing WSDL**- Allows to set a publishing WSDL as:
    - None
    - **Specify in-line**
    - **Specify source URL**
    - **Pick from registry**
  - **Do Not load service on startup** - Allows to choose whether or not to load the Service on startup.
  - **Pinned servers** - Allows to enter pinned servers. The names of the servers must be separated by comma or space.
  - **Service Group** - Allows to specify the Service Group if it is necessary. See more information about service groups in [Managing Service Groups](#).
- **Transport Settings:**
  - **http/https** - Allows to choose transport to use. Only enabled transports appear in the "Transports" list. See the detailed information in [Working with Transports](#).

Click "Next" to go to the Step 2.

Step 2

In sequence handles the incoming requests to the proxy service before they are dispatched to the target endpoint. A proxy service must contain an in sequence or an endpoint or both.

Design switch to source view

### Step 2 of 3 - In Sequence and Endpoint Options

**Proxy Service Name: PS2**

In sequence handles the incoming requests to the proxy service before they are dispatched to the target endpoint. A proxy service must contain an in sequence or an endpoint or both.

**Define In Sequence**

- None
- Define Inline
- Pick from Registry
- Use Existing Sequence

**Define Endpoint**

- None
- Define Inline
- Pick from Registry
- Use Existing Endpoint

The following options are to be specified:

- **In Sequence Options:**

- **None**
- **Define Inline** - Allows to use the "Create" link to create and add a sequence. See [Adding a Mediation Sequence](#).

None

Define Inline 

Pick from Registry

Use Existing Sequence

- **Pick from registry** - Allows to choose the "In Sequence" path from the Configuration Registry or the Governance Registry. For more information about the Registry Browser, see [Working with the Registry](#)

None

Define Inline

Pick from Registry

Configuration Registry Governance Registry

- **Use Existing Sequence** - Allows to choose an existing sequence from the drop-down menu.

None

Define Inline

Pick from Registry

Use Existing Sequence

- **Endpoint Options:**

- **None**

- **Define Inline-** Allows to use the "Create" link to create and add an endpoint. See [Adding an Endpoint](#).

<input type="radio"/> None	<input checked="" type="radio"/> Define Inline	<b>Create</b>
<input type="radio"/> Pick from Registry		
<input type="radio"/> Use Existing Endpoint		

- **Pick from registry -** Allows to choose the "Endpoint" path from the Configuration Registry or the Governance Registry. For more information about the Registry Browser see [Working with the Registry](#).

<input type="radio"/> None	<input type="radio"/> Define Inline	<input checked="" type="radio"/> Pick from Registry	<input type="radio"/> Use Existing Endpoint	<b>Configuration Registry</b>	<b>Governance Registry</b>
----------------------------	-------------------------------------	-----------------------------------------------------	---------------------------------------------	-------------------------------	----------------------------

- **Use Existing Endpoint -** Allows to choose an existing endpoint from the drop-down menu.

<input type="radio"/> None	<input type="radio"/> Define Inline	<input type="radio"/> Pick from Registry	<input checked="" type="radio"/> Use Existing Endpoint	<b>None</b>
----------------------------	-------------------------------------	------------------------------------------	--------------------------------------------------------	-------------

Click "Next" to go to the Step 3.

<b>&lt;Back</b>	<b>Next&gt;</b>	<b>Cancel</b>
-----------------	-----------------	---------------

Step 3

Out sequence mediates the response messages from the backend service before they are sent back to the client. Fault sequence is invoked in case of any errors while mediating a message through the proxy service. Both out sequence and fault sequence are optional.

Design switch to source view

### Step 3 of 3 - Out Sequence and Fault Sequence Options

**Proxy Service Name: PS2**

Out sequence mediates the response messages from the backend service before they are sent back to the client. Fault sequence is invoked in case of any errors while mediating a message through the proxy service. Both out sequence and fault sequence are optional.

<b>Define Out Sequence</b>
<input checked="" type="radio"/> None
<input type="radio"/> Define Inline
<input type="radio"/> Pick from Registry
<input type="radio"/> Use Existing Sequence

<b>Define Fault Sequence</b>
<input checked="" type="radio"/> None
<input type="radio"/> Define Inline
<input type="radio"/> Pick from Registry
<input type="radio"/> Use Existing Sequence

<b>&lt;Back</b>	<b>Finish</b>	<b>Cancel</b>
-----------------	---------------	---------------

The following options are to be specified:

- **Out Sequence Options:**
  - **None**
  - **Define Inline -** Allows to use the "Create" link to create and add an Out Sequence. See [Adding a](#)

### Mediation Sequence.

<input type="radio"/> None	
<input checked="" type="radio"/> Define Inline	<a href="#">Create</a>
<input type="radio"/> Pick from Registry	
<input type="radio"/> Use Existing Sequence	

- **Pick from registry** - Allows to choose the "Out Sequence" path from the Configuration Registry or the Governance Registry. For more information about the Registry Browser see [Working with the Registry](#).

<input type="radio"/> None	
<input type="radio"/> Define Inline	
<input checked="" type="radio"/> Pick from Registry	<input type="text"/>
<input type="radio"/> Use Existing Sequence	

 Configuration Registry  Governance Registry

- **Use Existing Sequence** - Allows to choose an existing sequence from the drop-down menu.

<input type="radio"/> None	
<input type="radio"/> Define Inline	
<input type="radio"/> Pick from Registry	
<input checked="" type="radio"/> Use Existing Sequence	<input type="button" value="None"/>

- **Fault Sequence Options:**

- **None**
- **Define Inline** - Allows to use the "Add" link to create and add a Fault Sequence. See [Adding a Mediation Sequence](#).

<input type="radio"/> None	
<input checked="" type="radio"/> Define Inline	<a href="#">Create</a>
<input type="radio"/> Pick from Registry	
<input type="radio"/> Use Existing Sequence	

- **Pick from registry** - Allows to choose the "Fault Sequence" path from the Configuration Registry or the Governance Registry. For more information about the Registry Browser see [Working with the Registry](#).

<input type="radio"/> None	
<input type="radio"/> Define Inline	
<input checked="" type="radio"/> Pick from Registry	<input type="text"/>
<input type="radio"/> Use Existing Sequence	

 Configuration Registry  Governance Registry

- **Use Existing Sequence** - Allows to choose an existing sequence from the drop-down menu.

<input type="radio"/> None	
<input type="radio"/> Define Inline	
<input type="radio"/> Pick from Registry	
<input checked="" type="radio"/> Use Existing Sequence	<input type="button" value="None"/>

### Logging Proxy Template

The "**Logging Proxy**" template creates a proxy service that logs all the incoming requests and forwards them to a given [endpoint](#). If necessary, responses coming back from the backend service can be logged before routing them

to the client.

### Logging Proxy

Create a proxy service which logs all the incoming requests and forwards them to a given endpoint. If necessary responses coming back from the backend service can be logged before routing them to the client.

**Proxy Service Settings**

Proxy Service Name*	<input type="text"/>
Target Endpoint*	<input checked="" type="radio"/> Enter URL <input type="radio"/> Pick from Registry
Target URL*	<input type="text"/>
Request Log Level	Simple <input type="button" value="▼"/>
Response Log Level	None <input type="button" value="▼"/>

Publish WSDL Options  
 Transports

**Create** **Cancel**

The Proxy Service settings:

- **Proxy Service Name** - Allows do define the unique name of the Proxy Service.
- **Target Endpoint**- Allows to be selected as:
  - **Enter URL**
    - **Target URL** - Allows to enter a Target URL into the field. The URL should be a string and it should not contain any expressions.

Target Endpoint*	<input checked="" type="radio"/> Enter URL <input type="radio"/> Pick from Registry
Target URL*	<input type="text"/>

- **Pick from registry**
  - **Endpoint Key** - Allows to pick the source from the Configuration Registry or Governance Registry. For more information about the Registry Browser, see [Working with the Registry](#).

Target Endpoint*	<input type="radio"/> Enter URL <input checked="" type="radio"/> Pick from Registry
Endpoint Key*	<input type="text"/>
	<input type="button" value="Configuration Registry"/> <input type="button" value="Governance Registry"/>

- **Request Log Level** - Can be selected from the drop-down menu as "Full" or "Simple."
- **Responce Log Level** - Can be selected from the drop-down menu as "Full" or "Simple."
- **Publish WSDL options**
- **Transports**

### Pass Through Proxy Template

The "Pass Through Proxy" template creates a simple proxy service on a specified endpoint. The proxy service does not perform any processing on the messages that pass through it.

### Pass Through Proxy

Create a simple proxy service on a specified endpoint. The proxy service does not perform any processing on the messages that pass through the proxy.

**Proxy Service Settings**

Proxy Service Name*	<input type="text"/>
Target Endpoint*	<input checked="" type="radio"/> Enter URL <input type="radio"/> Pick from Registry
Target URL*	<input type="text"/>

Publish WSDL Options  
 Transports

**Create** **Cancel**

The Proxy Service settings are as follows:

- **Proxy Service Name** - Enter a unique name for this proxy service.
- **Target Endpoint**- Select one of the following options:
  - **Enter URL**
    - **Target URL** - Type the URL of the target endpoint in the field. The URL should be a string and it should not contain any expressions.

Target Endpoint\*  Enter URL  Pick from Registry  
Target URL\*

- **Pick from registry**
  - **Endpoint Key** - Select the endpoint source from the Configuration Registry or Governance Registry using the Registry Browser. For more information, see [Working with the Registry](#).

Target Endpoint\*  Enter URL  Pick from Registry  
Endpoint Key\*  Configuration Registry Governance Registry

- Publish WSDL options
- Transports

### Secure Proxy Template

The **Secure Proxy** template creates a proxy service with WS-Security engaged. The service processes WS-Security on incoming requests and forwards the requests to an unsecured backend service.

**Secure Proxy**

Create a proxy with WS-Security engaged. The service will process WS-Security on incoming requests and forward them to an unsecured backend service.

Proxy Service Settings

Proxy Service Name*	<input type="text"/>
Target Endpoint*	<input checked="" type="radio"/> Enter URL <input type="radio"/> Pick from Registry
Target URL*	<input type="text"/>
Security Policy	<input type="text"/> Configuration Registry  Governance Registry
<input checked="" type="checkbox"/> Publish WSDL Options	
<input checked="" type="checkbox"/> Transports	
<input type="button" value="Create"/>	<input type="button" value="Cancel"/>

The proxy service settings are as follows:

- **Proxy Service Name** - Specify a unique name for this proxy service.
- **Target Endpoint**- Select one of the following options:
  - **Enter URL**
    - **Target URL** - Enter the URL of the target endpoint in the field. The URL should be a string and it should not contain any expressions.

Target Endpoint\*  Enter URL  Pick from Registry  
Target URL\*

- **Pick from registry**
  - **Endpoint Key** - Select the target endpoint source from the Configuration Registry or Governance Registry. For more information on the registry, see [Working with the Registry](#).

Target Endpoint\*  Enter URL  Pick from Registry  
Endpoint Key\*  Configuration Registry Governance Registry

- **Security Policy** - Select from the Configuration Registry or Governance Registry. For more information on

- the registry, see [Working with the Registry](#).
- [Publish WSDL options](#)
- [Transports](#)

#### **Transformer Proxy Template**

The "**Transformer Proxy**" template creates a proxy service that transforms the incoming requests before forwarding them to a given endpoint. It can also transform the responses before sending them to the client.

Create a proxy service which transforms all the incoming requests using XSLT and then forwards them to a given endpoints. If required responses coming back from the backend service can be transformed as well.

Proxy Service Settings

Proxy Service Name\*

Target Endpoint\*  Enter URL  Pick from Registry

Target URL\*

Request XSLT  Configuration Registry Governance Registry

Transform Responses

Publish WSDL Options

Transports

**Create** **Cancel**

The proxy service settings are as follows:

- **Proxy Service Name** - Enter a unique name for this proxy service.
- **Target Endpoint**- Select one of the following:
  - **Enter URL**
    - **Target URL** - Enter the URL of the target endpoint in the field. The URL should be a string and it should not contain any expressions.

Target Endpoint\*  Enter URL  Pick from Registry

Target URL\*

- **Pick from registry**
  - **Endpoint Key** - Select the target endpoint source from the Configuration Registry or Governance Registry. For more information on the registry, see [Working with the Registry](#).

Target Endpoint\*  Enter URL  Pick from Registry

Endpoint Key\*  Configuration Registry Governance Registry

- **Request XSLT** - Select the XSLT to use for transforming requests from the Configuration Registry or Governance Registry.
- **Transform Responses**- Select the check box if you want to transform responses.
  - **Response XSLT** - If the Transform Responses check box is selected, select the XSLT to use for transforming the responses from the Configuration Registry or Governance Registry.
- [Publish WSDL options](#)
- [Transports](#)

#### **WSDL Based Proxy Template**

The "**WSDL Based Proxy**" template creates a proxy service out of a WSDL of an existing Web Service. Endpoint information is extracted from a remotely hosted WSDL of an actual service.

## WSDL Based Proxy

Create a proxy service out of a WSDL of an existing Web Service. Endpoint information is extracted from a remotely hosted WSDL of an actual service.

Proxy Service Settings

Proxy Service Name\*

WSDL URI\*

WSDL Service\*

WSDL Port\*

Publish Same Service Contract

Service Contract Publication Options

Transports

The Proxy Service settings are as follows:

- **Proxy Service Name** - Enter a unique name for this proxy service.
- **WSDL URI** - Enter the URL and URN of the WSDL. The URL defines the host address of the network resource (can be omitted if resources are not network homed), and the URN defines the resource name in local namespaces. For example, if the URL is `ftp://ftp.dlink.ru` and the URN is `/pub/ADSL/`, you would enter `ftp://ftp.dlink.ru/pub/ADSL/` for the URI. To ensure that the URI is valid, click **Test URI**.
- **WSDL Service**
- **WSDL Port**
- **Publish Same Service Contract** - Check this box if you want to publish the same WSDL
- **Service Contract Publication Options** - Specify the same options as the [Publish WSDL Options](#)
- **Transports**

### **Publishing a WSDL**

When you [create a proxy service](#), a default WSDL is generated for it automatically. However, the default WSDL only shows the `mediate` operation and does not provide the client with any information about the message format or parameters expected by the back-end service. To provide clients with this information, you can publish a custom WSDL based on the back-end service's WSDL, or if you don't want to publish all the operations of the back-end service, you can publish a modified version of the WSDL. For example, if the back-end service expects a message to include the name, department, and permission level, and you want the proxy service to inject the permission level as it processes the message, you could publish a WSDL that includes just the name and department and omits the permission level parameter, since that data will be added by the proxy service.

To publish a custom WSDL for this proxy service, select one of the "Publishing WSDL" options:

- [Specify in-line](#)
- [Specify source URL](#)
- [Pick from registry](#)

If the WSDL definition includes dependencies on other resources that are in the registry, [add those resources](#).  
Specify in-line

Enter the WSDL definition in the text area.

Publishing WSDL

Inline WSDL Definition

WSDL Resources

<input type="text" value="Location:"/>	<input type="text" value="Key:"/>	Configuration Registry	Governance Registry
<input type="button" value="Add Resource"/>			

#### Specify Source URL

Enter the URI of the WSDL in the text box, and then click **Test URI** to ensure it is available. A URI consists of a URL and URN. The URL defines the host address of the network resource (can be omitted if resources are not network homed), and the URN defines the resource name in local "namespaces."

For example: URI = `ftp://ftp.dlink.ru/pub/ADSL`

where: URL = `ftp://ftp.dlink.ru` URN = `pub/ADSL`

Publishing WSDL

WSDL URI

WSDL Resources

<input type="text" value="Location:"/>	<input type="text" value="Key:"/>	Configuration Registry	Governance Registry
<input type="button" value="Add Resource"/>			

#### Pick From Registry

If the WSDL is saved as a registry entry, select this option and choose the reference key of that registry entry from the Governance Registry or Configuration Registry. For more information, see [Working with the Registry](#).

Publishing WSDL

Reference Key

Configuration Registry    Governance Registry

WSDL Resources

<input type="text" value="Location:"/>	<input type="text" value="Key:"/>	Configuration Registry	Governance Registry
<input type="button" value="Add Resource"/>			

#### Adding WSDL resources from the registry

If the WSDL has dependencies on items that you have saved in the registry, you map their location as specified in the WSDL to their key in the registry. For example, assume the WSDL imports the metadata schema as follows:

```
<xsd:import namespace="http://www.wso2.org/test/10" schemaLocation="metadata.xsd" />
```

In this case, the location is specified in the WSDL as `metadata.xsd`, so enter `metadata.xsd` in the Location field and then specify the registry key of the `metadata.xsd` file either by typing the key or clicking the appropriate registry and selecting it. After specifying the location and key, click **Add Resource**.

**WSDL Resources**

Location:	Key:	Action
<a href="#">http://Seq</a>	conf:/Sequence_2	Delete

**Add Resource**

You can delete a WSDL Resource using the "Delete" button in the "Action" column.

Location	Key	Action
<a href="#">http://Seq</a>	conf:/Sequence_2	Delete

### Managing Proxy Services

On the ESB Management Console, you can go to the **Deployed Services** screen to view all existing proxy services. The following icon denotes a proxy services:



Follow the instructions below to set up specific configurations for a proxy service.

1. Log on to the ESB Management Console.
2. On the **Main** tab of Management Console go to **Manage -> Services** and click **List**. You will see the **Deployed Services** screen with details of all existing proxy services displayed in the **Services** list.

**Deployed Services**

6 active services. 6 deployed service group(s).

Service Type	ALL	Service	Search		
Select all in this page   Select none		Delete			
<b>Services</b>					
<input type="checkbox"/>	echo				
<input type="checkbox"/>	Proxy Service				
<input type="checkbox"/>	PS2				
<input type="checkbox"/>	Version				
	wso2carbon-sts				
	XKMS				

3. Click the name of the required proxy service to open the service dashboard of the proxy service.
4. Click on **Deactivate** in the **Operations** panel if you want to de-activate the proxy service.
5. Click the appropriate link on the **Specific Configuration** panel based on your requirement. You can perform the following actions via the **Specific Configuration** panel:
  - [Edit a proxy service](#)
  - [Redeploy a proxy service](#)

### Note

When redeploying a proxy service, there is a two-second delay by default to allow the existing configuration to finish processing the current messages. You can change this delay by adding the `hotupdate.timeout` parameter to the parameters section of `<ESB_HOME>/repository`

y/conf/axis2/axis2.xml. For example, to eliminate the delay, add the following line:  
<parameter name="hotupdate.timeout" locked="false">0</parameter>

- Enable or disable statistics
- Enable or disable tracing

### **Editing a proxy service**

#### **To edit a proxy service**

1. Click **Edit** on the **Specific Configuration** panel. This opens the **Modify Proxy Service** screen.

The screenshot shows the 'Modify Proxy Service' interface. At the top, there's a 'Design' tab and a 'switch to source view' link. Below that, it says 'Step 1 of 3 - Basic Settings'. The 'Proxy Service Name' is set to 'TEST'. In the 'General Settings' section, 'Publishing WSDL' is set to 'None'. There is a green '+' icon followed by 'Add Parameter'. In the 'Transport Settings' section, 'https' and 'http' are checked with checked boxes, while 'jms' and 'local' are unchecked. At the bottom, there are 'Next>' and 'Cancel' buttons.

2. Edit the following proxy service options as required. For more information about Proxy Service options, see [Adding a Proxy Service](#).

- **Publishing WSDL:** The WSDL used to provide information about the message format and parameters of the back end service. See [Publishing a WSDL](#) for further information.
- **Service Parameters:** If you want to add a parameter to the proxy service, click **Add Parameter**. Then add the name and the required value of the relevant parameter.
- **Service Group:** The axis2 service group to which the proxy service is related.
- **Do Not load service on startup:** If this check box is selected, the proxy service will be deactivated at the time the ESB server is started/restarted.
- **Pinned servers:** The servers in which the proxy service should be activated when multiple instances of the ESB exist in a cluster. If no servers are specified in this field, the proxy service will be activated in all the servers.

- **Service Description:** This is a free text field used to enter a description for the proxy service.
- **Transport:** Select a transport for the proxy service by selecting one of the check boxes.

3. Click **Finish** to save your changes.

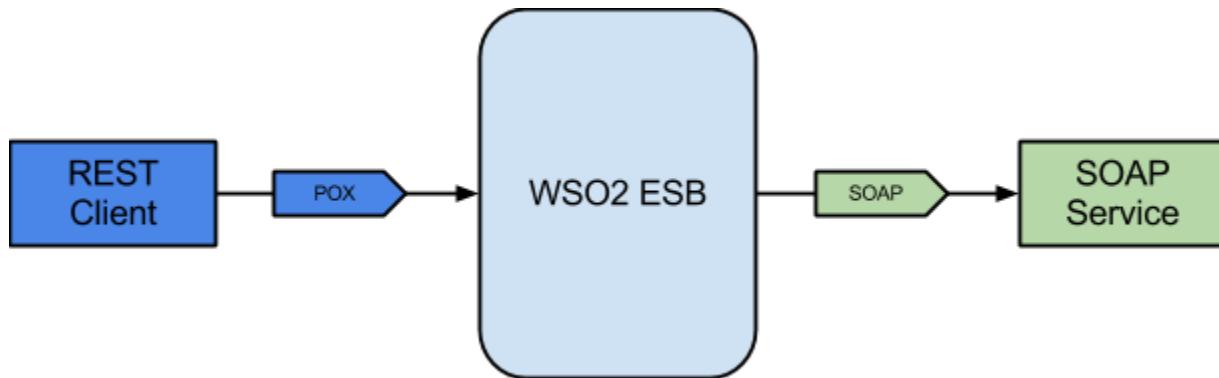
## Using REST with a Proxy Service

If you have a REST front-end client, REST back-end service, or both a REST client and service, you can use a proxy service in the ESB to handle the communication between the front end and back end. This page describes the following scenarios:

- REST Client and SOAP Service
- SOAP Client and REST Service
- REST Client and REST Service
- JMS Client and REST Service

### REST Client and SOAP Service

This usecase represents the scenario where a REST front-end client has to communicate in plain old XML (POX) with a SOAP back-end service. This can be easily achieved by placing the WSO2 ESB in the middle with a proxy service.



Assume the front-end REST client is using http/https as its transport protocol, and the back-end service is expecting SOAP1.1. The proxy service configuration would look like this:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="http https" startOnLoad="true">
 <target>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"
 format="soap11"/>
 </endpoint>
 <outSequence>
 <send/>
 </outSequence>
 <inSequence>
 <header name="Action" value="urn:getQuote"/>
 </inSequence>
 </target>
 </proxy>
</definitions>

```

In this scenario, the client sends a REST message to the ESB, which transforms the message to a SOAP1.1 message and sends it to the back-end service. Once the back-end response is received, the ESB sends the response back to the client as a REST message.

To implement this scenario:

1. Start the ESB and change the configuration as shown above.
2. Deploy the back-end service 'SimpleStockQuoteService' and start the Axis2 server using the instructions in [Starting Sample Back-End Services](#). You will now send a message to the back-end service through the ESB using the sample [Stock Quote Client](#). This client can run in several modes.
3. Run the following ant command from the <ESB\_HOME>/samples/axis2Client directory to trigger a sample message to the back-end service:  

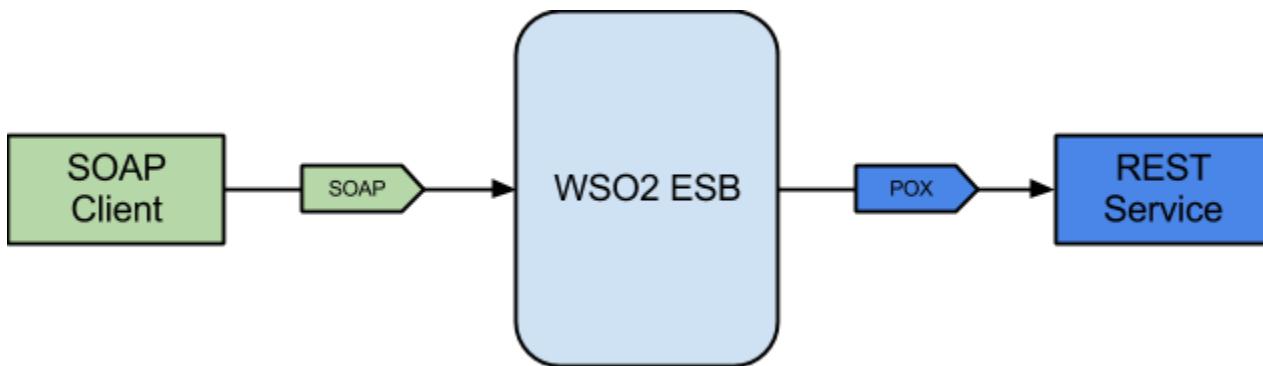
```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy -Drest=true
```

If the message is mediated successfully, it will display an output on the Axis2 server's start-up console along with an output message on the client console. If you want to see exactly how the REST messages are transformed to SOAP messages, you could place an Apache tcpmon application between the client and the ESB and another between the ESB and the back-end service.

For another example of POX to SOAP conversion, see [Sample 50: POX to SOAP Conversion](#).

### **SOAP Client and REST Service**

This usecase represents the scenario where a SOAP front-end client has to communicate with a REST back-end service. This can be easily achieved by placing the WSO2 ESB in the middle with a proxy service.



The proxy service configuration would look like this:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="CustomerServiceProxy"
 transports="http https"
 startOnLoad="true">
 <target>
 <inSequence>
 <filter xpath="//getCustomer">
 <property name="REST_URL_POSTFIX"
 expression="//getCustomer/id"
 scope="axis2"
 type="STRING"/>
 <property name="HTTP_METHOD" value="GET" scope="axis2" type="STRING"/>
 </filter>
 <header name="Accept" scope="transport" value="application/xml"/>
 <send>
 <endpoint>
 <address
 uri="http://localhost:9764/jaxrs_basic/services/customers/customerservice/customers"
 format="pox"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 </proxy>
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default "fault" sequence"/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
 </sequence>
 <sequence name="main">
 <log/>
 <drop/>
 </sequence>
</definitions>

```

In this scenario, the client sends a SOAP message to the ESB, which transforms it to a REST message and sends it to the back-end service. Once the back-end response is received, the ESB sends it back to the client as a SOAP message. The [filter mediator](#) is used to identify the required service. Though this example only has one filter, there could be many filters in the real scenario. The [REST\\_URL\\_POSTFIX property](#) is a well-defined property, and its value is appended to the target URL when sending messages out in a RESTful manner. Similar to [REST\\_URL\\_POSTFIX](#), the [HTTP\\_METHOD](#) property is also well-defined and sets the HTTP verb to GET.

Note that the format of the endpoint is set to "pox". This format works for GET, PUT, and DELETE methods, but if you want to use POST and the URI has a postfix, the format must be "rest".

To implement this scenario:

1. Start the ESB and change the configuration as shown above.
2. Download, install, and start [WSO2 Application Server](#). The application server ships with JAX-RS, which contains a set of pure RESTful services, so we will use this as our back-end service.

3. Send a message to the back-end service through the ESB using [SoapUI](#). The service used in this scenario is the `getCustomer` service, which requires the customer ID to complete its task. A sample SOAP message that is used to achieve this is as follows:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
 <soapenv:Header/>
 <soapenv:Body>
 <getCustomer>
 <id>123</id>
 </getCustomer>
 </soapenv:Body>
</soapenv:Envelope>
```

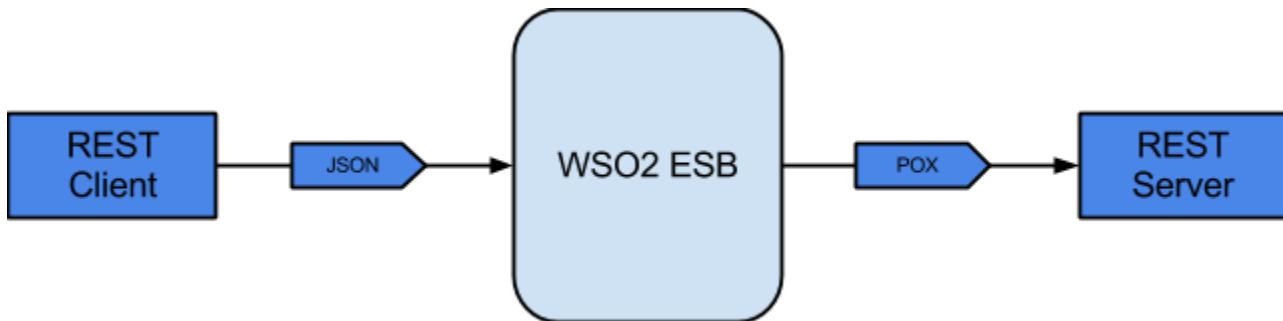
This message simply represents the relevant method to be invoked and its associated value. Upon successful execution, SoapUI should display the following message in response.

```
<Customer>
 <id>123</id>
 <name>John</name>
</Customer>
```

For another example, see [Sample 152: Switching Transports and Message Format from SOAP to REST POX](#).

### REST Client and REST Service

This usecase represents the scenario where a REST front-end client has to communicate with a REST back-end service. Even though they are both using the same architecture, let's assume the front-end client is using JSON whereas the back-end service is using plain old XML (POX). This can be easily achieved by placing the WSO2 ESB in the middle with a proxy service.



The proxy service configuration is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="CustomerServiceProxy"
 transports="http https"
 startOnLoad="true">
 <target>
 <inSequence>
 <property name="ContentType" value="text/xml" scope="axis2"
type="STRING"/>
 <switch source="$axis2:HTTP_METHOD">
 <case regex="GET">
 <property name="HTTP_METHOD" value="GET" scope="axis2"
type="STRING"/>
 </case>
 <case regex="POST">
 <property name="HTTP_METHOD" value="POST" scope="axis2"
type="STRING"/>
 </case>
 <default/>
 </switch>
 <send>
 <endpoint>
 <address
uri="http://localhost:9764/jaxrs_basic/services/customers/customerservice"
format="pox"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 </proxy>
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default "fault" sequence"/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
 </sequence>
 <sequence name="main">
 <log/>
 <drop/>
 </sequence>
</definitions>

```

In this scenario, the ESB simply changes the message type of the client into XML and then passes it to the REST service. Once the ESB has received the XML message, it transforms it back into a JSON message and sends it to the client.

Typically, it's very important to set the `HTTP_METHOD` property, because this represents the required HTTP action to be sent to the back-end service. However, in this scenario we will use a `curl` request, which is done using the HTTP GET method, so by default any request is using GET as its HTTP action, even if the `HTTP_METHOD` property is not set.

To implement this scenario:

1. Start the ESB and change the configuration as shown above.
2. Download, install, and start [WSO2 Application Server](#). The application server ships with JAX-RS, which contains a set of pure RESTful services, so we will use this as our back-end service.
3. Go to the command prompt and issue the following `curl` command to simulate the request of a real REST client:
 

```
curl -v -i -H "Accept: application/json" http://localhost:8280/services/CustomerServiceProxy/customers/123
```

Following is the expected output that should appear on the console upon successful execution of the scenario:

```
{
 "Customer": {
 "id": 123,
 "name": "John"
 }
}
```

## JMS Client and REST Service

This usecase represents the scenario where a JMS front-end client has to communicate with a REST back-end service. This can be easily achieved by placing the WSO2 ESB in the middle with a proxy service.

In this scenario, an ESB proxy service listens to a JMS queue, picks up available messages from that queue, and delivers the messages to the REST back-end service. For the JMS front-end client, we will use the [sample JMSClient](#) that ships with the ESB. For the REST back-end service, we will use the [JAX-RS Basics](#) sample service available in WSO2 Application Server. We will use ActiveMQ 5.5.1 as the message broker.

The proxy service configuration is as follows:

```
<proxy xmlns="http://ws.apache.org/ns/synapse" name="JmsToRestProxy" transports="jms" statistics="disable" trace="disable" startOnLoad="true">
 <target>
 <inSequence>
 <property name="OUT_ONLY" value="true"/>
 <send>
 <endpoint>
 <address
uri="http://localhost:9764/jaxrs_basic/services/customers/customerservice/customers"/>
 </endpoint>
 </send>
 </inSequence>
 </target>
 <parameter name="transport.jms.ContentType">
 <rules>
 <jmsProperty>contentType</jmsProperty>
 <default>application/xml</default>
 </rules>
 </parameter>
 <description></description>
</proxy>
```

To implement this scenario:

1. Start ActiveMQ and configure the JMS transport in ESB to work with ActiveMQ .
2. Configure and deploy the JAX-RS Basics sample by following the [instructions for building and running the sample](#) .
3. Since we are going to run both WSO2 ESB and the WSO2 Application Server on the same machine, offset the ports of the application server by changing the offset value to 1 in `<AS_HOME>/repository/conf/carbon.xml`.
4. Goto `<ESB_HOME>/samples/axis2Client` and execute the following command:  
`ant jmsclient -Djms_type=text -Djms_dest=dynamicQueues/JmsToRestProxy -Djms_payload=<Customer><name>WSO2</name></Customer>`  
This sends `<Customer><name>WSO2</name></Customer>` as the payload.

In the application server console, you will see the following line printed:

```
----invoking addCustomer, Customer name is: WSO2
```

## Applying Security to a Proxy Service

The steps below demonstrate how to apply security for a proxy service via [WSO2 ESB Tooling](#) by creating a security policy, and then deploying it in the server.

- Creating the proxy service in WSO2 ESB Tooling
- Creating the security policy
- Applying security to a proxy service
- Deploying the secured proxy service in WSO2 ESB

### Creating the proxy service in WSO2 ESB Tooling

You can [create a new proxy service](#) or [import an existing proxy service](#) from an XML file, such as a Synapse Configuration file.

#### *Creating a new proxy service*

Follow these steps to create a proxy service. Alternatively, you can [import an existing proxy service](#).

1. In WSO2 ESB Tooling, open the Developer Studio Dashboard (click **Developer Studio > Open Dashboard**) and click **Proxy Service** in the Enterprise Service Bus area.
2. Select **Create a New Proxy Service** and click **Next**.
3. Type a unique name for the proxy service and specify the proxy type (see below).
4. Do one of the following:
  - To save the proxy service in an existing ESB Config project in your workspace, click **Browse** and select that project.
  - To save the proxy service in a new ESB Config project, click **Create new ESB Project** and create the new project.
5. If you specified a proxy type that requires that you enter the target endpoint (the endpoint that represents the actual service), do one of the following:
  - If you know the URL of the endpoint, select **Enter URL** and type it in the text box.
  - If you want to use an endpoint you've already defined in this workspace, select **Predefined Endpoint** and select it from the list.
  - If you want to use an endpoint in the registry, select **Pick from Registry**, and then either type the endpoint's registry key or click **Browse**, click **Registry**, and navigate to the endpoint in the registry.
6. Fill in the advanced configuration based on the proxy service type you specified:
  - **Transformer Proxy:** Transforms all the incoming requests using XSLT and then forwards them to a given target endpoint. Specify the target endpoint as described in the previous step, and then specify the location of the XSLT you want to use to transform requests, either by typing the path or by clicking

**Browse** and navigating to the XSLT, which can be a file in the workspace or registry or can be a local entry. If you also want to transform the responses from the backend service, click **Transform Responses**.

- **Log Forward Proxy:** Logs all the incoming requests and forwards them to a given endpoint. It can also log responses from the backend service before routing them to the client. Specify the log level for requests and responses, where **Simple** logs To, From, WSAction, SOAPAction, ReplyTo, MessageID, and any properties, and **Full** logs all attributes of the message plus the SOAP envelope information.
- **Pass Through Proxy:** Forwards messages to the endpoint without performing any processing on them. This proxy service is useful as a catch-all, so that messages that do not meet the criteria to be handled by other proxy services are simply forwarded to the endpoint. When you select this proxy service type, you just specify the target endpoint as described in the previous step.
- **WSDL Based Proxy:** A proxy service that is created from the remotely hosted WSDL of an existing web service. The endpoint information is extracted from the WSDL. In the **URI** field, enter the URL and URN of the WSDL. The URL defines the host address of the network resource (can be omitted if resources are not network homed), and the URN defines the resource name in local namespaces. For example, if the URL is `ftp://ftp.dlink.ru` and the URN is `/pub/ADSL/`, you would enter `ftp://ftp.dlink.ru/pub/ADSL/` for the URI. To ensure that the URI is valid, click **Test URI**. You then enter the service name and port of the WSDL. Lastly, if you want to publish this WSDL, click **Publish Same Service Contract**.
- **Secure Proxy:** Uses WS-Security to process incoming requests and forward them to an unsecured backend service. Specify the target endpoint as described in the previous step, and then specify the key of the security policy or click **Browse** and select it from the registry.
- **Custom Proxy:** A custom proxy service in which you customize all the sequences, endpoints, transports, and other QoS settings by adding them to the mediation workflow after the proxy service is created.

7. Click **Finish**. The proxy service is created in the `src/main/synapse-config/proxy-service` folder under the ESB Config Project you specified, and the proxy service appears in the editor. Click its icon in the editor to view its properties.

### Importing a proxy service

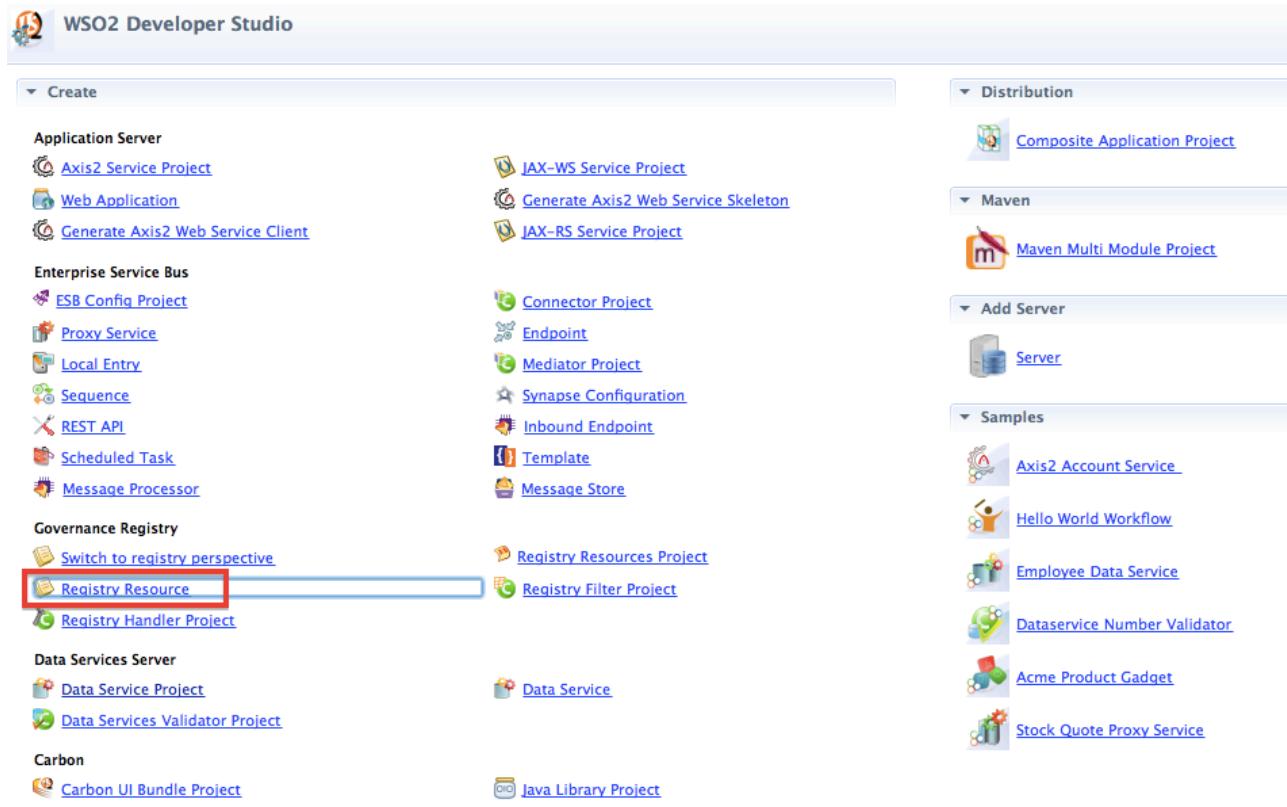
Follow these steps to import an existing proxy service from an XML file (such as a Synapse configuration file) into an ESB Config project. Alternatively, you can [create a new proxy service](#).

1. In WSO2 ESB Tooling, open the Developer Studio Dashboard (click **Developer Studio > Open Dashboard**) and click **Proxy Service** in the Enterprise Service Bus area.
2. Select **Import Proxy Service** and click **Next**.
3. Specify the proxy service file by typing its full pathname or clicking **Browse** and navigating to the file.
4. In the **Save Proxy Service In** field, specify an existing ESB Config project in your workspace where you want to save the proxy service, or click **Create new ESB Project** to create a new ESB Config project and save the proxy service there.
5. If there are multiple proxy services in the file, in the **Advanced Configuration** section select the proxy services you want to import.
6. Click **Finish**. The proxy services you selected are created in the `src/main/synapse-config/proxy-service` folder under the ESB Config project you specified, and the first proxy service appears in the editor.

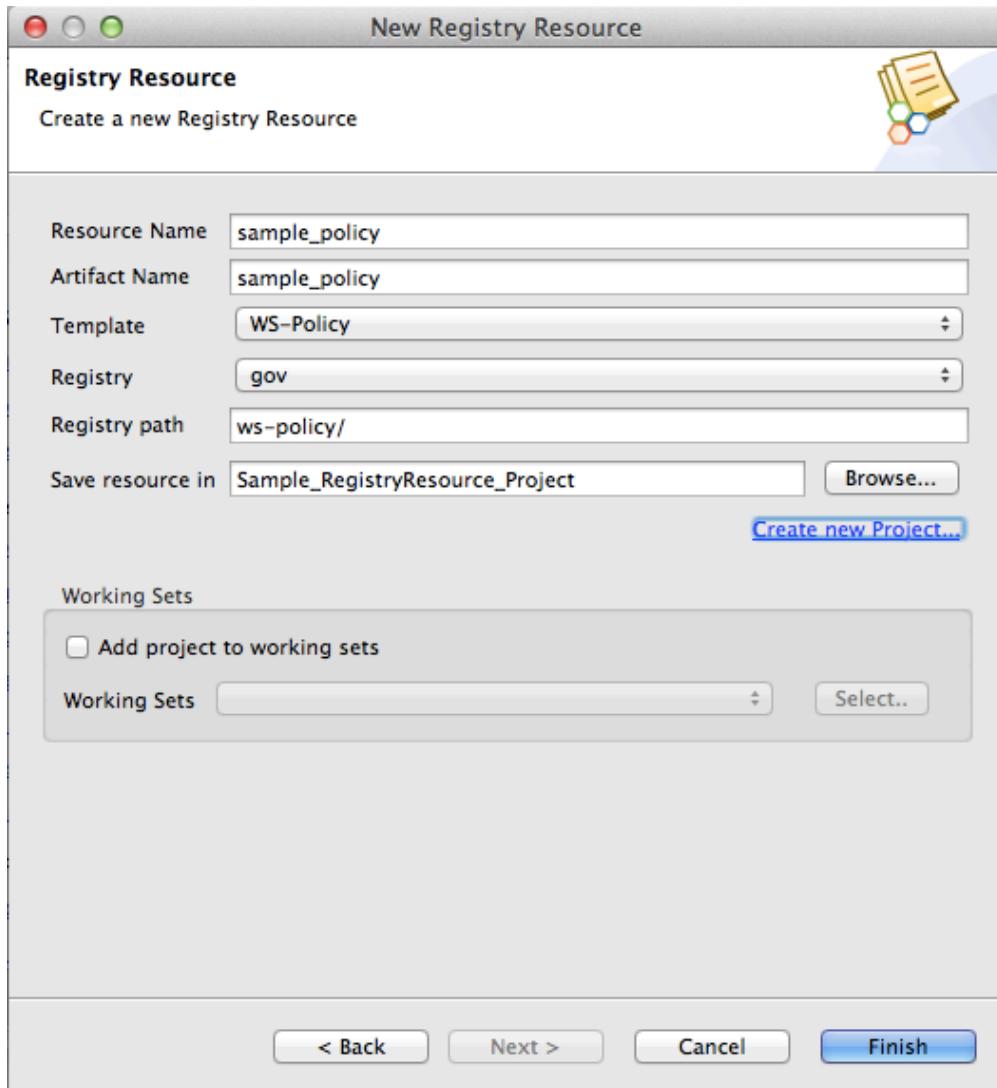
### Creating the security policy

Follow the steps below to create a security policy to define the required security configurations.

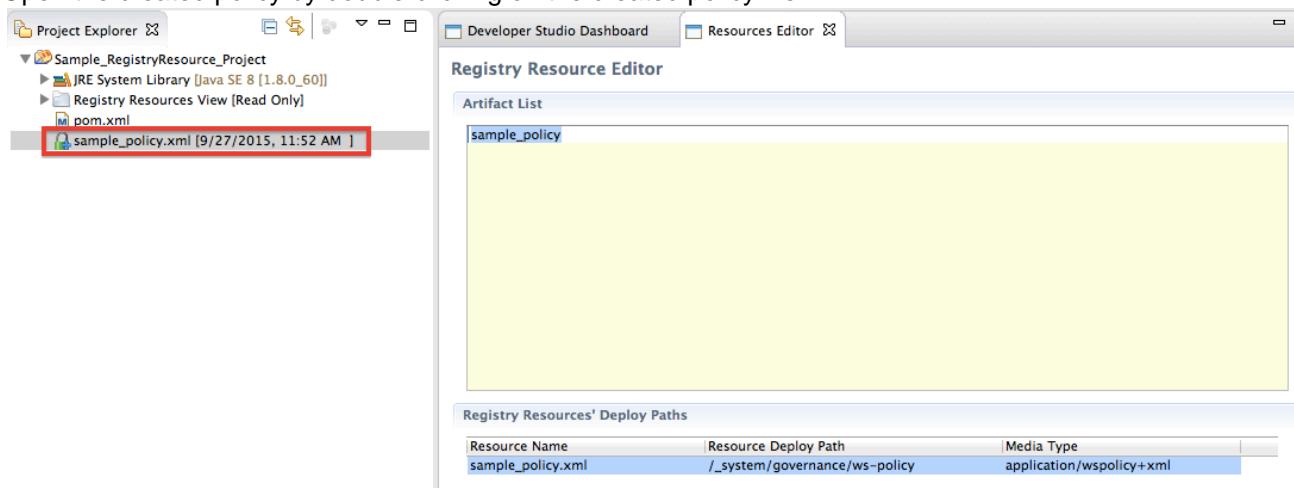
1. Open the Developer Studio Dashboard (click **Developer Studio > Open Dashboard**) and click **Registry Resource**.



2. Select the **From existing template** option and click **Next**.
3. Enter a resource name and select the **WS-Policy** template along with the preferred registry path.



4. Click **Finish**.
5. Open the created policy by double-clicking on the created policy file.



6. The policy file opens in a multi page editor with a Security Form Editor as the design view and an XML editor as the source view.

#### Design View

## WS-Policy for Service

### ▼ Security for the service

#### Basic Scenarios

-   [UsernameToken](#)
-   [Non-repudiation](#)
-   [Integrity](#)
-   [Confidentiality](#)

User Roles

#### Advanced Scenarios

-   [Sign and Encrypt – X509 Authentication](#)
-   [Sign and Encrypt – Anonymous clients](#)
-   [Encrypt only – Username Token Authentication](#)
-   [Sign and Encrypt – Username Token Authentication](#)
-   [SecureConversation – Sign only – Service as STS – Bootstrap policy – Sign and Encrypt , X509 Authentication](#)
-   [SecureConversation – Encrypt only – Service as STS – Bootstrap policy – Sign and Encrypt , X509 Authentication](#)
-   [SecureConversation – Sign and Encrypt – Service as STS – Bootstrap policy – Sign and Encrypt , X509 Authentication](#)
-   [SecureConversation – Sign Only – Service as STS – Bootstrap policy – Sign and Encrypt , Anonymous clients](#)
-   [SecureConversation – Sign and Encrypt – Service as STS – Bootstrap policy – Sign and Encrypt , Anonymous clients](#)
-   [SecureConversation – Encrypt Only – Service as STS – Bootstrap policy – Sign and Encrypt , Username Token Authentication](#)
-   [SecureConversation – Sign and Encrypt – Service as STS – Bootstrap policy – Sign and Encrypt , Username Token Authentication](#)

Design Source

## Source View

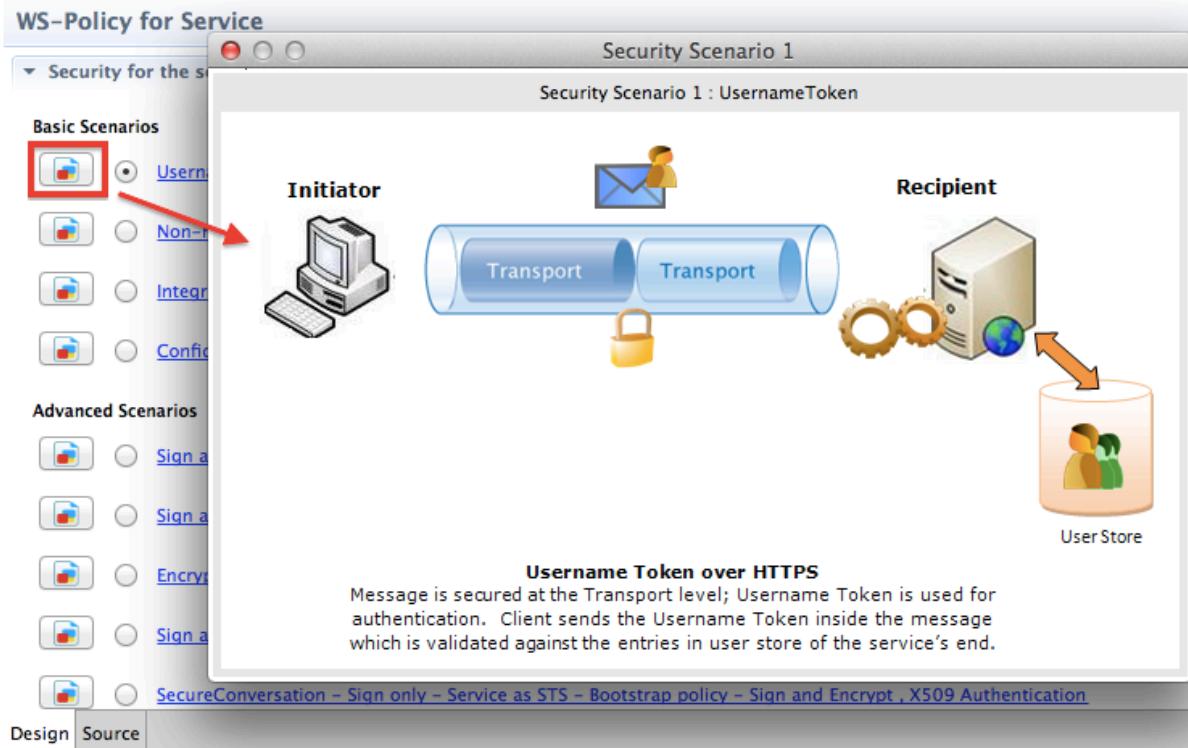
```

1<wsp:Policy wsu:Id="UTOverTransport"
2 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
3 xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
4<wsp:ExactlyOne>
5<wsp:All>
6<sp:TransportBinding
7 xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
8<wsp:Policy>
9<sp:TransportToken>
10<wsp:Policy>
11 <sp:HttpsToken RequireClientCertificate="false" />
12</wsp:Policy>
13</sp:TransportToken>
14<sp:AlgorithmSuite>
15<wsp:Policy>
16 <sp:Basic256 />
17</wsp:Policy>
18</sp:AlgorithmSuite>
19<sp:Layout>
20<wsp:Policy>
21 <sp:Lax />
22</wsp:Policy>
23</sp:Layout>
24<sp:IncludeTimestamp />
25</wsp:Policy>
26</sp:TransportBinding>
27<sp:SignedSupportingTokens
28 xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
29<wsp:Policy>
30 <sp:UsernameToken
31 sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/AlwaysToRecipient" />
32</wsp:Policy>
33</sp:SignedSupportingTokens>
34</wsp:All>
35</wsp:ExactlyOne>
36<rampart:RampartConfig xmlns:rampart="http://ws.apache.org/rampart/policy">
37 <rampart:user>wso2carbon</rampart:user>
38 <rampart:encryptionUser>useReqSigCert</rampart:encryptionUser>
39 <rampart:timestampPrecisionInMilliseconds>true
40 </rampart:timestampPrecisionInMilliseconds>
41 <rampart:timestampTTL>300</rampart:timestampTTL>
42 <rampart:timestampMaxSkew>300</rampart:timestampMaxSkew>
43 <rampart:timestampStrict>false</rampart:timestampStrict>
44 <rampart:tokenStoreClass>org.wso2.carbon.security.util.SecurityTokenStore

```

Design Source

7. Enable security by specifying the required scenario in the Security Form Editor. Click the icon next to each scenario for more information.



8. You can provide service information as private store and advanced configuration information as rampart configuration.

**Encryption Properties**

Alias:	wso2carbon
Privatestore:	* wso2carbon.jks
Tenant id:	-1234
Truststores:	* wso2carbon.jks
User:	wso2carbon

**Signature Properties**

Alias:	wso2carbon
Privatestore:	* wso2carbon.jks
Tenant id:	-1234
Truststores:	* wso2carbon.jks
User:	wso2carbon

Design   Source

- Client authenticates with SAML 2.0 protection token, Symmetric Key, X509 Certificate by the Service
- Client authenticates with SAML 1.1 protection token, Symmetric Key, X509 Certificate by the Service
- Sign and Encrypt – X509 Authentication – SAML 2.0 Token Required as Supporting
- Sign and Encrypt – X509 Authentication – SAML 1.1 Token Required as Supporting

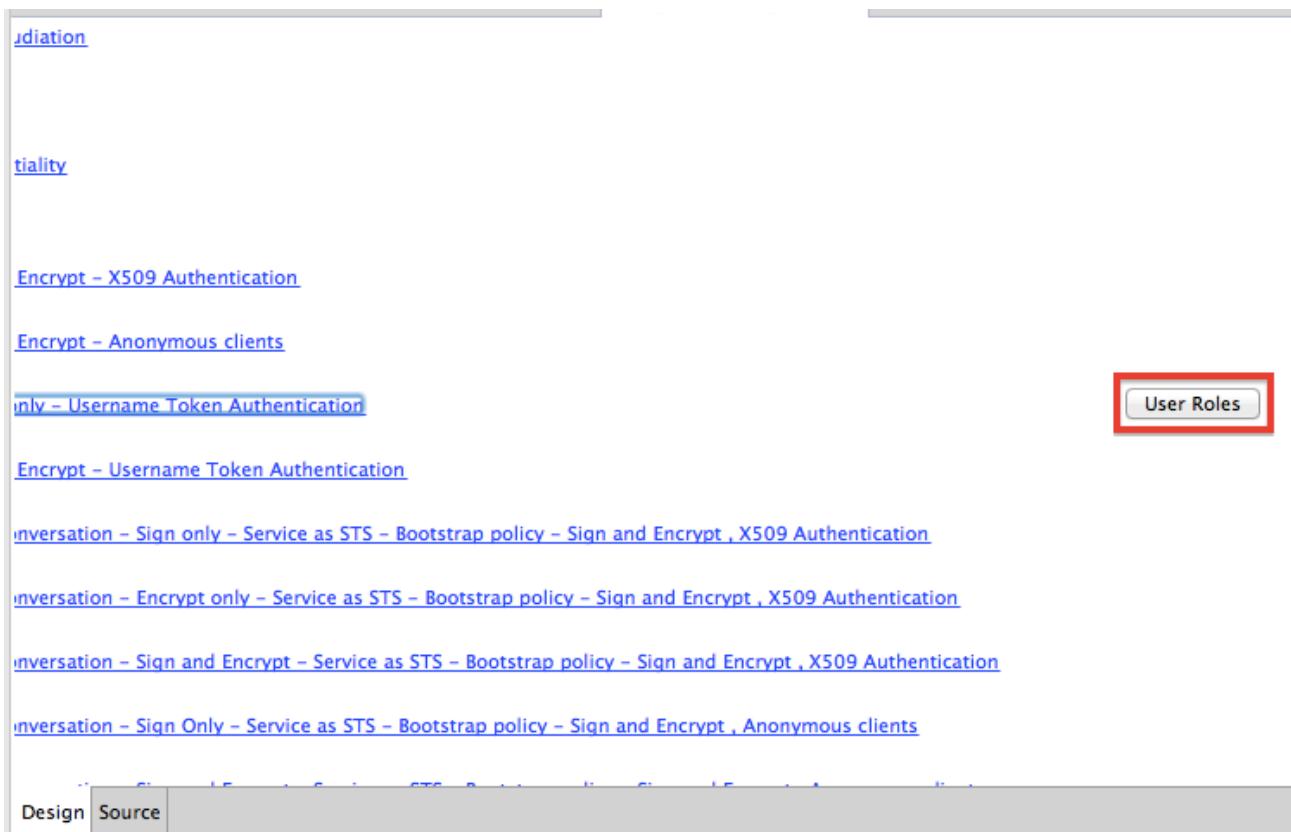
**Advance Configuration(Rampart)**

**Rampart Configuration**

User :	wso2carbon
encryptionUser :	useReqSigCert
PrecisionInMilliseconds :	true
timestampTTL :	300
timestampMaxSkew :	300
timestampStrict :	false
tokenStoreClass :	org.wso2.carbon.security.util.SecurityTokenStore
nonceLifeTime :	300

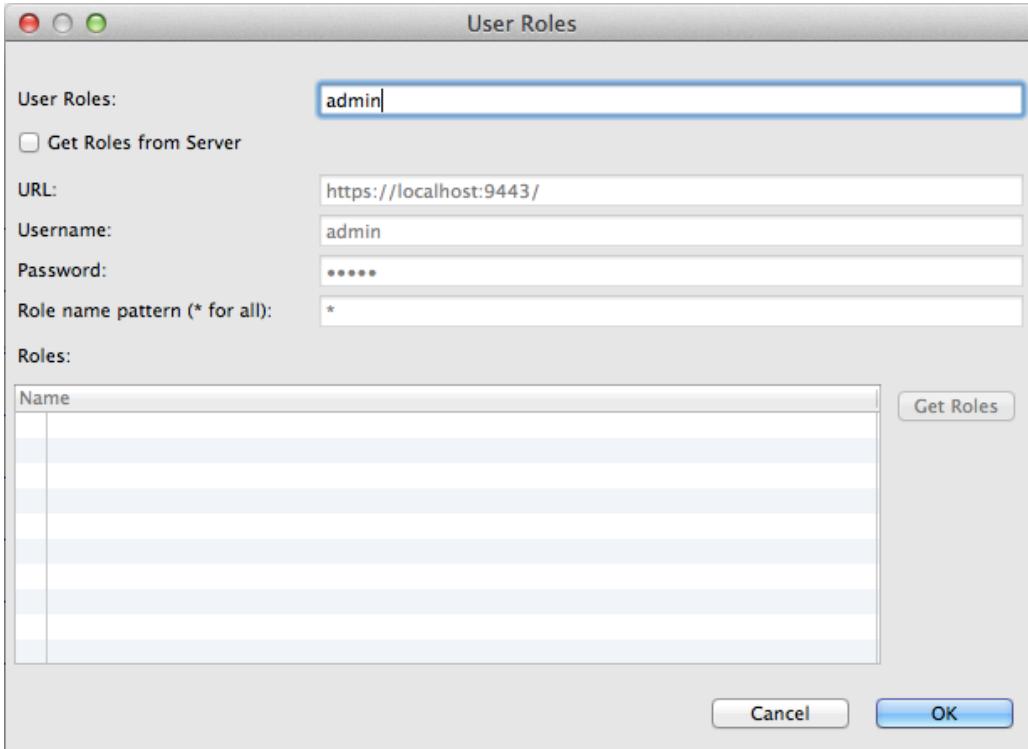
Design   Source

- For certain scenarios, you can specify user roles. After you select the scenario, scroll to the right to see the **User Roles** button. Alternatively, maximize the window.

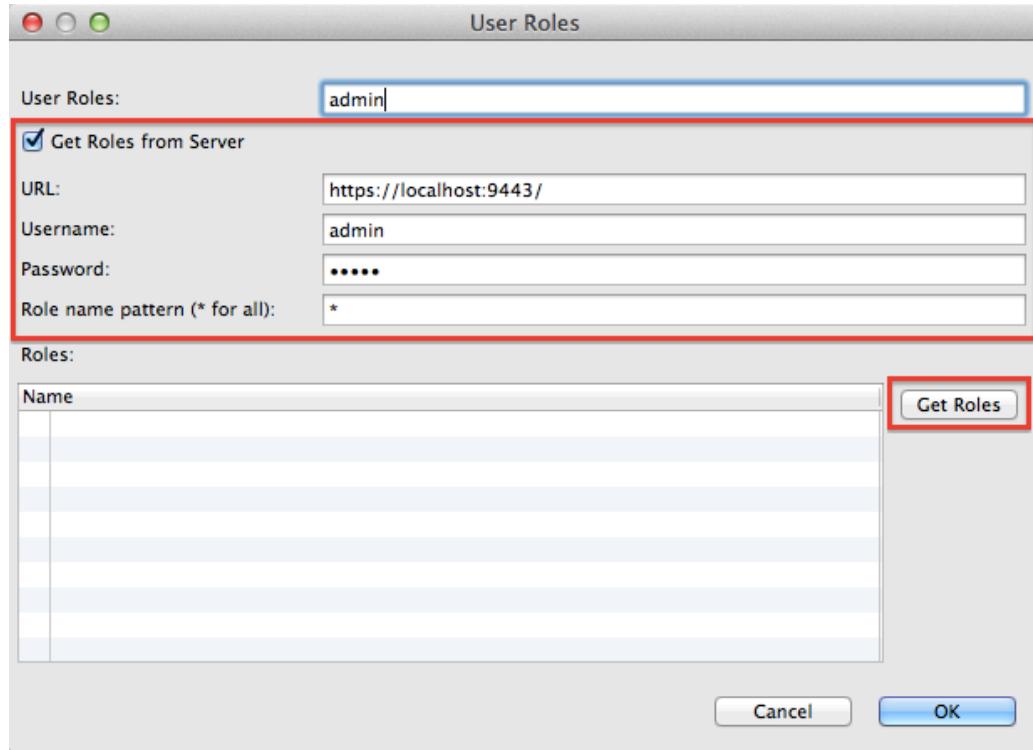


10. Either define the user roles inline or retrieve the user roles from the server.

#### Define Inline



**Get from the server**

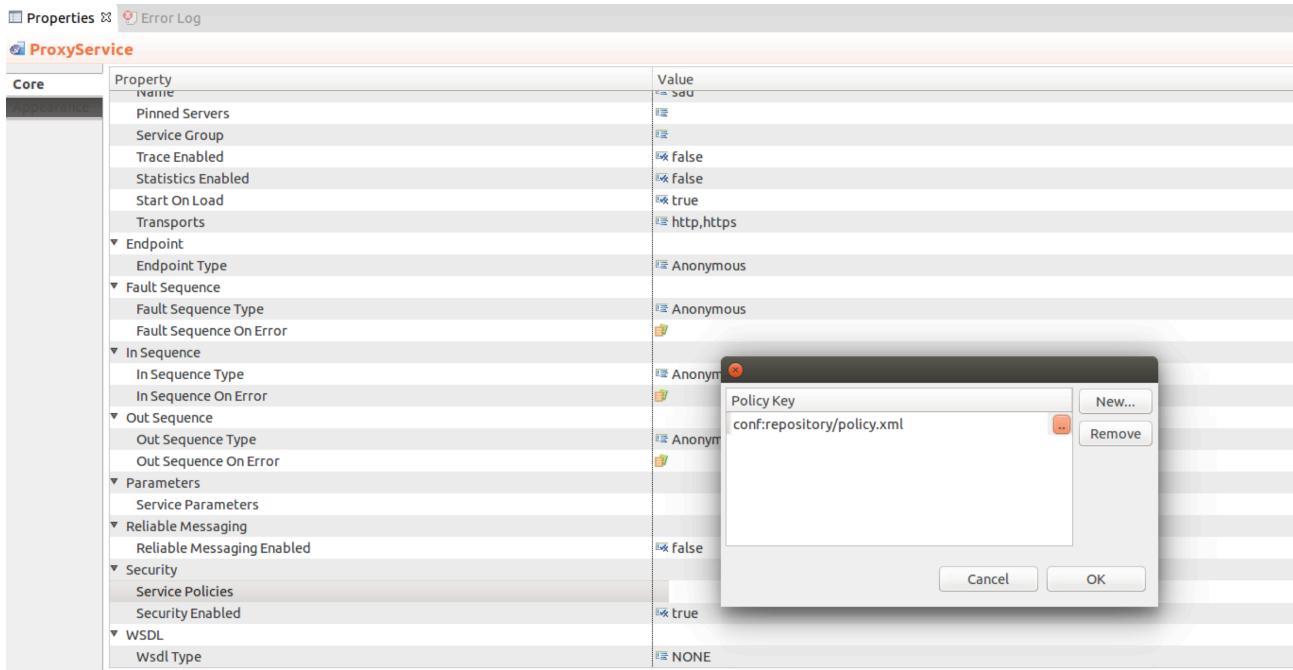


## Applying security to a proxy service

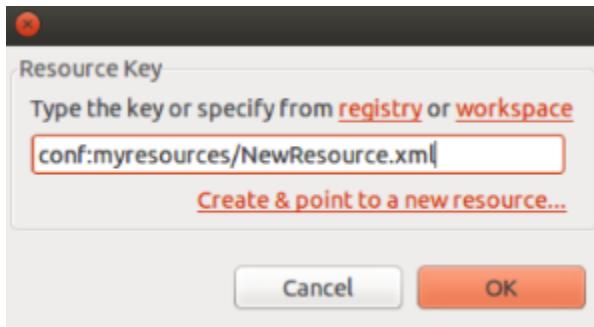
Follow the steps below to apply security to a proxy service.

- Once you have configured the policy file, you can apply security for a proxy service by setting the **Security Enabled** property to true and pointing to the policy key under **Service Policies** in the proxy properties.

Property	Value
Pinned Servers	["seu"]
Service Group	[""]
Trace Enabled	false
Statistics Enabled	false
Start On Load	true
Transports	["http,https"]
Endpoint	
Endpoint Type	Anonymous
Fault Sequence	
Fault Sequence Type	Anonymous
Fault Sequence On Error	Anonymous
In Sequence	
In Sequence Type	Anonymous
In Sequence On Error	Anonymous
Out Sequence	
Out Sequence Type	Anonymous
Out Sequence On Error	Anonymous
Parameters	
Service Parameters	
Reliable Messaging	
Reliable Messaging Enabled	false
Security	
Service Policies	
Security Enabled	true
Wsdl	
Wsdl Type	true



- Specify the policy path inline or browse from the registry or workspace. You can also create and point to a new resource.



By default, the role names are not case sensitive. If you want to make them case sensitive, add the following property under the `<AuthorizationManager>` configuration in the `user-mgt.xml` file:

```
<Property name="CaseSensitiveAuthorizationRules">true</Property>
```

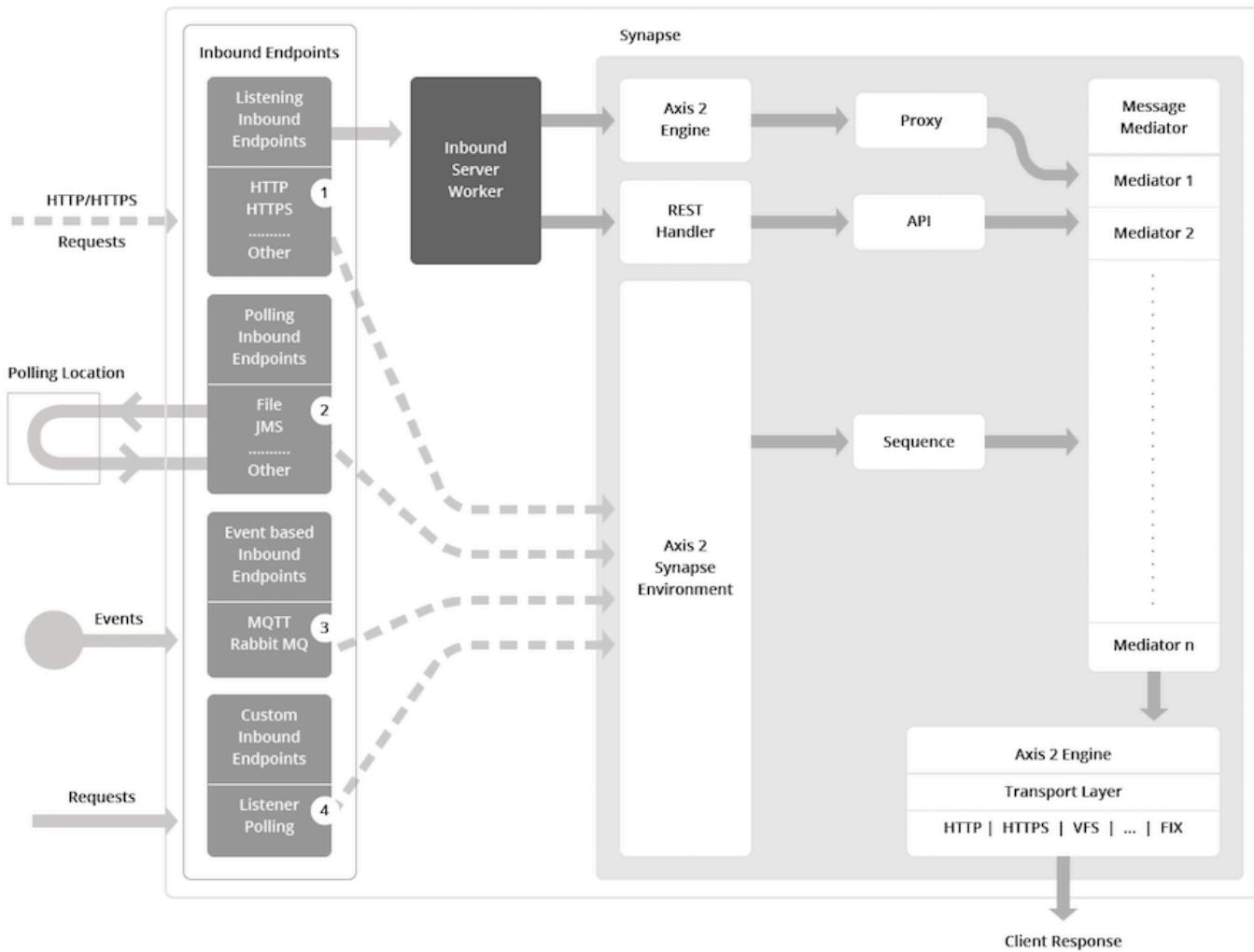
## Deploying the secured proxy service in WSO2 ESB

Create a Composite Application Project including the secured proxy service and the security policy registry resource, and then create a CAR file to deploy it in the WSO2 ESB server. For instructions on creating and deploying the Composite Application Project, see [Packaging Artifacts into Composite Applications](#).

If the security policy registry resource is deployed in WSO2 ESB, at the time of creating the Composite Application Project, ensure the server role selected for the registry resource in the **Composite Application Project POM Editor** is changed to **EnterpriseServiceBus**.

## Working with Inbound Endpoints

An inbound endpoint is a message entry point that can inject messages directly from the transport layer to the mediation layer, without going through the Axis engine. The following diagram illustrates the inbound endpoint architecture.



Out of the existing transports only the HTTP transport supports multi-tenancy, this is one limitation that is overcome with the introduction of the inbound architecture. Another limitation when it comes to conventional Axis2 based transports is that the transports do not support dynamic configurations. With WSO2 ESB inbound endpoints, it is possible to create inbound messaging channels dynamically, and there is also built-in cluster coordination as well as multi-tenancy support for all transports.

For detailed information on each type of inbound endpoint available with WSO2 ESB, see [WSO2 ESB Inbound Endpoints](#).

### Working with Inbound Endpoints

When it comes to creating and managing inbound endpoints, you can use the ESB tooling plug-in to create a new inbound endpoint and to import an existing inbound endpoint, or you can manage inbound endpoints via the ESB Management Console. For detailed information on how to work with inbound endpoints, see the following topics:

- [Working with Inbound Endpoints via WSO2 ESB Tooling](#)
- [Managing Inbound Endpoints via the Management Console](#)

## WSO2 ESB Inbound Endpoints

Based on the protocol, the behaviour of an inbound endpoint can either be listening, polling or event-based.

For detailed information on listening, polling and event-based inbound endpoints see the following topics:

- Listening Inbound Endpoints
- Polling Inbound Endpoints
- Event-Based Inbound Endpoints

For information on how to create a custom inbound endpoint based on your requirement, see [Custom Inbound Endpoint](#).

Following is a sample inbound endpoint configuration:

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
 name="HttpListenerEP"
 sequence="TestIn"
 onError="fault"
 protocol="http"
 suspend="false">
 <parameters>
 <parameter name="inbound.http.port">8085</parameter>
 </parameters>
</inboundEndpoint>
```

In an inbound endpoint configuration, the common inbound endpoint parameters are specified as attributes of the `<i nboundEndpoint>` element whereas the protocol specific parameters are specified as `<parameter>` elements.

### Common inbound endpoint parameters

The following parameters are common to all inbound endpoints:

Parameter Name	Description	Required	Possible Values	Default Value
sequential	The behavior when executing the given sequence. When set as <code>true</code> , mediation will happen within the same thread. When set as <code>false</code> , the mediation engine will use the inbound thread pool. (The default thread pool values can be found in the <code>&lt;ESB_HOME&gt;/repository/conf/synapse.properties</code> file).	Yes	true or false	true
suspend	When set to <code>true</code> , this makes the inbound endpoint inactive.	Yes	true or false	false

### Specifying inbound endpoint parameters as registry values

Other than specifying parameter values inline, you can also specify parameter values as registry entries. The advantage of specifying a parameter value as a registry entry is that the same inbound endpoint configuration can be used in different environments simply by changing the registry entry value.

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse" name="file"
sequence="request" onError="fault" protocol="file" suspend="false">
 <parameters>

 <parameter name="transport.vfs.FileURI"
key="conf:/repository/esb/esb-configurations/test"/>

 </parameters>
</inboundEndpoint>
```

If you need to provide the registry entry value via the Management Console, specify it as `$registry:conf:/repo`

sitory/esb/esb-configurations/test.

## Using secure vault aliases in your inbound endpoint configuration

To use secure vault support in your inbound configuration, you can add `{wso2:vault-lookup('xx')}` as inbound parameters, where `xx` is the alias.

You can [encrypt and store the password](#) using the alias `my.password`, and retrieve this password by adding the following parameter in your inbound endpoint configuration:

```
<parameter
name="transport.jms.Password">{wso2:vault-lookup('my.password')}</parameter>
```

### Listening Inbound Endpoints

A listening inbound endpoint listens on a given port for requests that are coming in. When a request is available it is injected to a given sequence. Listening inbound endpoints support two way operations and are synchronous.

See the following topics for detailed information on each listening inbound endpoint available with WSO2 ESB:

- [HTTP Inbound Protocol](#)
- [HTTPS Inbound Protocol](#)
- [HL7 Inbound Protocol](#)
- [CXF WS-RM Inbound Protocol](#)
- [WebSocket Inbound Protocol](#)
- [Secure WebSocket Inbound Protocol](#)

For information on how to create a custom inbound endpoint by extending the listening inbound endpoint behaviour, see [Creating a custom listening inbound endpoint](#).

### **HTTP Inbound Protocol**

The HTTP inbound protocol is used to separate endpoint listeners for each HTTP inbound endpoint so that messages are handled separately. The HTTP inbound endpoint can bypass the inbound side axis2 layer and directly inject messages to a given sequence or API. For proxy services, messages will be routed through the axis2 transport layer in a manner similar to normal transports. You can start dynamic HTTP inbound endpoints without restarting the server.

Following is a sample HTTP inbound endpoint configuration:

```
<inboundEndpoint name="HttpListenerEP" protocol="http" suspend="false"
sequence="TestIn" onError="fault" >
 <p:parameters xmlns:p="http://ws.apache.org/ns/synapse">
 <p:parameter name="inbound.http.port">8081</p:parameter>
 </p:parameters>
</inboundEndpoint>
```

HTTP inbound endpoint parameters

Parameter	Description	Required
<code>inbound.http.port</code>	The port on which the endpoint listener should be started .	Yes

### Note

- A sequence should be designed with a call/respond or send/receive sequence. It is not recommended to use a sequence that has in and out mediators.

- If a send mediator is used within the inbound endpoint sequence, specify a receiving sequence. If you do not specify a receiving sequence, the response will dispatch to the main sequence.

#### Worker pool configuration parameters

By default inbound endpoints share the PassThrough transport worker pool to handle incoming requests. If you need a separate worker pool for the inbound endpoint, you need to configure the following parameters:

Parameter	Description
inbound.worker.pool.size.core	The initial number of threads in the worker thread pool. This value is based on the number of messages to be processed. The default value is the value of the <code>inbound.worker.pool.size.max</code> parameter.
inbound.worker.pool.size.max	The maximum number of threads in the worker thread pool. This value is used to avoid performance degradation that can occur due to context switching between threads.
inbound.worker.thread.keep.alive.sec	The keep-alive time for extra threads in the worker pool. When this time is elapsed for an extra thread, it is terminated. The purpose of this parameter is to optimize the usage of resources by avoiding extra threads that are not utilized.
inbound.worker.pool.queue.length	The length of the queue that is used to hold runnable tasks. The thread pool starts queuing jobs when all existing threads have reached the maximum number of threads. The value for this parameter is the maximum length of the queue. If a bound queue is used and the queue gets filled up, new jobs submitted to submit jobs will fail causing synapse to drop some messages.
inbound.thread.group.id	Unique Identifier of the thread group.
inbound.thread.id	Unique Identifier of the thread.
dispatch.filter.pattern	The regular expression that defines the proxy services accessible via the inbound endpoint. Provide the <code>.*</code> expression to expose all proxy services. An expression similar to <code>^( /foo   /bar   /services/MyProxy )\$</code> will expose via the inbound endpoint. If you do not provide a filter, none of the inbound endpoint will be accessible.

Following is a sample configuration for an inbound endpoint with a separate worker pool:

```

<inboundEndpoint name="HttpListenerEP" protocol="http" suspend="false"
sequence="TestIn" onError="fault" >
 <p:parameters xmlns:p="http://ws.apache.org/ns/synapse">
 <p:parameter name="inbound.http.port">8081</p:parameter>
 <p:parameter name="api.dispatching.enabled">false</p:parameter>
 <p:parameter name="inbound.thread.group.id">Pass_Through Inbound worker
Pool</p:parameter>
 <p:parameter name="inbound.worker.pool.size.max">500</p:parameter>
 <p:parameter name="inbound.thread.id">PassThroughInboundWorkerPool</p:parameter>
 <p:parameter name="inbound.worker.pool.queue.length">-1</p:parameter>
 <p:parameter name="inbound.worker.pool.size.core">400</p:parameter>
 <p:parameter name="inbound.worker.thread.keep.alive.sec">60</p:parameter>
 </p:parameters>
</inboundEndpoint>

```

Inbound endpoint parameter for proxy services

If a proxy service is to be exposed only via inbound endpoints, the following service parameter has to be set in the proxy configuration.

Service Parameter	Description	Default Value
inbound.only	Whether the proxy service needs to be exposed only via inbound endpoints.  If set to true all requests that the proxy service receives via normal transport will be rejected. The proxy service will process only the requests that are received via inbound endpoints.	false

Following is a sample proxy configuration:

```

<proxy xmlns="http://ws.apache.org/ns/synapse" name="InboundProxy"
transports="https,http" statistics="disable" trace="disable" startOnLoad="true">
 <target>
 <outSequence>
 <send/>
 </outSequence>
 <endpoint>
 <address uri="http://localhost:9773/services/HelloService/" />
 </endpoint>
 </target>
 <parameter name="inbound.only">true</parameter>
</proxy>

```

Samples

For a sample that demonstrates how a HTTP inbound endpoint can act as a dynamic http listener, see [Sample 902: HTTP Inbound Endpoint Sample](#).

#### HTTPS Inbound Protocol

The HTTPS inbound protocol is used to separate endpoint listeners for each HTTPS inbound endpoint so that messages are handled separately. This transport can bypass the inbound side axis2 layer and directly inject messages to a given sequence or API. You can start dynamic HTTPS inbound endpoints without restarting the

server.

Configuration parameters for a HTTPS inbound endpoint are XML fragments that represent various properties.

Following is a sample HTTPS inbound endpoint configuration:

```

<inboundEndpoint name="HttpListenerEP" protocol="https" suspend="false"
sequence="TestIn" onError="fault" >
 <p:parameters xmlns:p="http://ws.apache.org/ns/synapse">
 <p:parameter name="inbound.http.port">8081</p:parameter>
 <p:parameter name="keystore">
 <KeyStore>
 <Location>repository/resources/security/wso2carbon.jks</Location>
 <Type>JKS</Type>
 <Password>wso2carbon</Password>
 <KeyPassword>wso2carbon</KeyPassword>
 </KeyStore>
 </p:parameter>
 <p:parameter name="truststore">
 <TrustStore>
 <Location>repository/resources/security/client-truststore.jks</Location>
 <Type>JKS</Type>
 <Password>wso2carbon</Password>
 </TrustStore>
 </p:parameter>
 <p:parameter name="SSLVerifyClient">require</p:parameter>
 <p:parameter name="HttpsProtocols">TLSv1,TLSv1.1,TLSv1.2</p:parameter>
 <p:parameter name="SSLProtocol">SSLV3</p:parameter>
 <p:parameter name="CertificateRevocationVerifier">
 <CertificateRevocationVerifier enable="true">
 <CacheSize>10</CacheSize>
 <CacheDelay>2</CacheDelay>
 </CertificateRevocationVerifier>
 </p:parameter>
 </p:parameters>
</inboundEndpoint>
```

#### HTTPS inbound endpoint parameters

Parameter	Description	Required
inbound.http.port	The port on which the endpoint listener should be started .	Yes
keystore	The KeyStore location where keys are stored.	Yes
truststore	The TrustStore location where keys are stored.	No

SSLVerifyClient	Used when enabling mutual verification.	No
HttpsProtocols	The supporting protocols.	No
SSLProtocol	The supporting SSL protocol.	No
CertificateRevocationVerifier	When the enable attribute is set to true, this validates and verifies the revocation status of the host certificates using OCSP/CRL, when making HTTPS connections. If the enable attribute of this parameter is set to true, you also need to specify the CacheSize and CacheDelay. CacheSize - The maximum size of the cache. CacheDelay - The time duration between two consecutive scheduled cache managing tasks that perform housekeeping work for the cache.	No

**Note**

- A sequence should be designed with a call/respond or send/receive sequence. It is not recommended to use a sequence that has in and out mediators.
- If a send mediator is used within the inbound endpoint sequence, specify a receiving sequence. If you do not specify a receiving sequence, the response will dispatch to the main sequence.

Worker pool configuration parameters

By default inbound endpoints share the PassThrough transport worker pool to handle incoming requests. If you need a separate worker pool for the inbound endpoint, you need to configure the following parameters:

Parameter	Description
inbound.worker.pool.size.core	The initial number of threads in the worker thread pool. This value is based on the number of messages to be processed. The formula here is the value of the inbound.worker.pool.size * (1 + (inbound.worker.pool.size * 0.2))

inbound.worker.pool.size.max	The maximum number of threads in the worker thread pool to avoid performance degradation that can occur due to co
inbound.worker.thread.keep.alive.sec	The keep-alive time for extra threads in the worker pool. timeout. When this time is elapsed for an extra thread, its parameter is to optimize the usage of resources by avoiding extra threads that are not utilized.
inbound.worker.pool.queue.length	The length of the queue that is used to hold runnable tasks. The thread pool starts queuing jobs when all existing threads reach the maximum number of threads. The value for this parameter is a bound queue. If a bound queue is used and the queue gets filled to submit jobs will fail causing synapse to drop some messages.
inbound.thread.group.id	Unique Identifier of the thread group.
inbound.thread.id	Unique Identifier of the thread.
dispatch.filter.pattern	The regular expression that defines the proxy services accessible via endpoint. Provide the . * expression to expose all proxy services. expression similar to ^(/foo /bar /services/MyProxy) to expose via the inbound endpoint. If you do not provide a pattern, none of the inbound endpoint will be accessible.

#### Samples

For a sample that demonstrates how an HTTPS inbound endpoint can act as a dynamic https listener, see [Sample 903: HTTPS Inbound Endpoint Sample](#).

#### **HL7 Inbound Protocol**

The WSO2 ESB HL7 inbound protocol is a multi-tenant capable alternative to the WSO2 ESB HL7 transport. The HL7 inbound endpoint implementation is fully asynchronous and is based on the **Minimal Lower Layer Protocol(MLLP)** implemented on top of event driven I/O.

Following is a sample HL7 inbound endpoint configuration:

#### Sample HL7 Inbound Endpoint

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
 name="InboundHL7"
 sequence="main"
 onError="fault"
 protocol="hl7"
 suspend="false">
<parameters>
 <parameter name="inbound.hl7.Port">20000</parameter>
 <parameter name="inbound.hl7.AutoAck">true</parameter>
 <parameter name="inbound.hl7.ValidateMessage">true</parameter>
 <parameter name="inbound.hl7.TimeOut">10000</parameter>
 <parameter name="inbound.hl7.CharSet">UTF-8</parameter>
 <parameter name="inbound.hl7.BuildInvalidMessages">false</parameter>
 <parameter name="inbound.hl7.PassThroughInvalidMessages">false</parameter>
</parameters>
</inboundEndpoint>
```

In the above inbound endpoint configuration, you will see that the endpoint is defined using the HL7 protocol and that all requests coming into this endpoint are directed to the `main` sequence.

#### HL7 inbound endpoint parameters

The following table provides information on the HL7 inbound endpoint parameters you can set:

Parameter	Description
<code>inbound.hl7.Port</code>	The port on which you need to run the MLLP listener.
<code>inbound.hl7.AutoAck</code>	Whether or not an auto acknowledgement should be sent on message reception. If set to <code>false</code> , you can define the type of HL7 acknowledgement to be sent. For more information, see <a href="#">HL7 inbound endpoint mediation level properties</a> .
<code>inbound.hl7.ValidateMessage</code>	This enables HL7 message validation.
<code>inbound.hl7.TimeOut</code>	The timeout interval in milliseconds to trigger a NACK message.
<code>inbound.hl7.CharSet</code>	The character set used for encoding and decoding messages. Some message encodings (e.g. UTF-16, UTF-32) may result in byte values equal to the characters or byte values lower than 0x1F, resulting in errors.
<code>inbound.hl7.BuildInvalidMessages</code>	If the <code>inbound.hl7.ValidateMessage</code> parameter is set to <code>false</code> and a message is invalid, this parameter specifies whether the raw message received via the MLLP transport should be passed onto the mediation layer.
<code>inbound.hl7.PassthroughInvalidMessages</code>	If the <code>inbound.hl7.BuildInvalidMessages</code> parameter is set to <code>true</code> , it notifies the Axis2 HL7 transport sender whether to use the raw message or the parsed message.
<code>inbound.hl7.MessagePreProcessor</code>	An implementation of the <code>org.wso2.carbon.inbound.endpoint.processor.HL7MessagePreprocessor</code> interface can be defined here. It provides an opportunity to intercept incoming messages before any type of message parsing occurs.

#### HL7 inbound endpoint mediation level properties

Following are the mediation level properties that you can set in the HL7 inbound endpoint:

#### Note

The scope of these properties is the `default` scope.

Property	Description
<code>&lt;property name="HL7_RESULT_MODE" value="ACK NACK" scope="default"/&gt;</code>	This is used to define the type of HL7 acknowledgement to be sent. If the <code>inbound.hl7.AutoAck</code> parameter is set to <code>true</code> this property has no effect.
<code>&lt;property name="HL7_NACK_MESSAGE" value="&lt;ERROR MESSAGE&gt;" scope="default" /&gt;</code>	This is used to define a custom error message to be sent if you have set the property <code>HL7_RESULT_MODE</code> as <code>NACK</code> .

```
<property
 name="HL7_APPLICATION_ACK"
 value="true"
 scope="default"/>
```

If the `inbound.hl7.AutoAck` parameter is set to `false` and no immediate auto generated ACK is sent back to the client, this property defines whether we should automatically generate the ACK for the request once the mediation flow is complete. If both the `inbound.hl7.AutoAck` parameter and this property are set to `false`, you need to generate an ACK message in the correct format as a response.

#### Tuning the HL7 inbound endpoint

The HL7 inbound endpoint can be configured using the `<ESB_HOME>/repository/conf/hl7.properties` file.

The supported list of tuning parameters for this file are as follows:

Property	Description	Value
hl7_id_generator	By default the HAPI HL7 parsing library uses a file based ID generator to generate unique control IDs. To use a UUID based ID generator you can change this to 'uuid'.	file   uuid (default = file)
worker_threads_core	Defines the HL7 inbound worker thread pool size.	[0..9]* (default = 100)
io_thread_count	Defines the number of IO threads the IO Reactor uses. It is recommended to set this to the number of cores on the machine.	[0..9]* (default = 2)
so_timeout	Defines TCP socket timeout.	[0..9]* (default = 0)
connect_timeout	Defines the TCP connect timeout.	[0..9]* (default = 0)
so_keep_alive	Defines TCP socket keep alive.	true   false (default = true)
so_rcvbuf	Defines the TCP socket receive buffer size.	[0..9]* (default = 0 uses OS default. Maximum value depends on OS settings).
so_sndbuf	Defines the TCP socket send buffer size.	[0..9]* (default = 0 uses OS default. Maximum value depends on OS settings).

#### Samples

For a sample that demonstrates how the HL7 inbound protocol can be used to receive a simple HL7 message, see [Sample 905: Inbound HL7 with Automatic Acknowledgement](#).

#### CXF WS-RM Inbound Protocol

WS-ReliableMessaging allows SOAP messages to be reliably delivered between distributed applications, regardless of software or hardware failures. The CXF WS--RM inbound endpoint allows a client (RM Source) to communicate with the WSO2 ESB (RM Destination) with a guarantee that a message sent will be delivered.

#### Note

To configure the CXF WS-RM Inbound endpoint, you need to install the **CXF WS Reliable Messaging** feature. For information on how to install this feature, see [Installing Features](#). After you install this feature, you can configure the CXF WS-RM inbound endpoint as a custom inbound endpoint.

Following is a sample sequence that sends messages that are from a RM source to a non RM backend:

```
<sequence xmlns="http://ws.apache.org/ns/synapse" name="RMIn" onerror="fault">
 <in>
 <property name="PRESERVE_WS_ADDRESSING" value="true" />
 <header xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm"
name="wsrm:Sequence" action="remove" />
 <header xmlns:wsa="http://www.w3.org/2005/08/addressing" name="wsa:To"
action="remove" />
 <header xmlns:wsa="http://www.w3.org/2005/08/addressing" name="wsa:FaultTo"
action="remove" />
 <send>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </in>
 <out>
 <log level="custom">
 <property name="RM OutSequence" value="Sending Response" />
 </log>
 <send/>
 </out>
</sequence>
```

Since the WS-ReliableMessaging protocol uses specific SOAP headers, those SOAP headers should be removed from the sequence since in most cases the backend service would fail to understand them.

The CXF bus should be configured as required in order to control the manner in which WS-RM communication takes place. This can be done through a CXF spring configuration. Create a directory named `cxf` within `<ESB_HOME>/repository/conf` and save the CXF spring configuration file in the `<ESB_HOME>/repository/conf/cxf` directory.

```

<beans
 xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:wsa="http://cxf.apache.org/ws/addressing"
 xmlns:http="http://cxf.apache.org/transports/http/configuration"
 xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
 xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
 xsi:schemaLocation="http://cxf.apache.org/core
 http://cxf.apache.org/schemas/core.xsd
 http://cxf.apache.org/transports/http/configuration
 http://cxf.apache.org/schemas/configuration/http-conf.xsd
 http://schemas.xmlsoap.org/ws/2005/02/rm/policy
 http://schemas.xmlsoap.org/ws/2005/02/rm/wsrmpolicy.xsd
 http://cxf.apache.org/ws/rm/manager
 http://cxf.apache.org/schemas/configuration/wsrmpolicy.xsd
 http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd"
 xmlns:cxf="http://cxf.apache.org/core">
 <cxf:bus>
 <cxf:features>
 <wsa:addressing/>
 <wsrm-mgr:reliableMessaging>
 <wsrm-policy:RMAssertion>
 <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
 <wsrm-policy:AcknowledgementInterval Milliseconds="2000"/>
 </wsrm-policy:RMAssertion>
 <wsrm-mgr:destinationPolicy>
 <wsrm-mgr:acksPolicy intraMessageThreshold="0"/>
 </wsrm-mgr:destinationPolicy>
 </wsrm-mgr:reliableMessaging>
 </cxf:features>
 </cxf:bus>
</beans>

```

If you need to secure the endpoint, you should change the CXF spring configuration as follows:

```

<beans
 xmlns="http://www.springframework.org/schema/beans"
 xmlns:sec="http://cxf.apache.org/configuration/security"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:cxf="http://cxf.apache.org/core"
 xmlns:wsa="http://cxf.apache.org/ws/addressing"
 xmlns:httpj="http://cxf.apache.org/transports/http-jetty/configuration"
 xmlns:http="http://cxf.apache.org/transports/http/configuration"
 xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
 xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
 xsi:schemaLocation="http://cxf.apache.org/core
 http://cxf.apache.org/transports/http/configuration
 http://cxf.apache.org/schemas/configuration/http-conf.xsd
 http://schemas.xmlsoap.org/ws/2005/02/rm/policy
 http://schemas.xmlsoap.org/ws/2005/02/rm/wsrmpolicy.xsd
 http://cxf.apache.org/configuration/security
 http://cxf.apache.org/schemas/configuration/security.xsd
 http://cxf.apache.org/ws/rm/manager"

```

```

http://cxf.apache.org/schemas/configuration/wsrm-manager.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/transports/http-jetty/configuration
http://cxf.apache.org/schemas/configuration/http-jetty.xsd">
<cxft:bus>
 <cxft:features>
 <wsa:addressing/>
 <wsrm-mgr:reliableMessaging>
 <wsrm-policy:RMAssertion>
 <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
 <wsrm-policy:AcknowledgementInterval Milliseconds="2000"/>
 </wsrm-policy:RMAssertion>
 <wsrm-mgr:destinationPolicy>
 <wsrm-mgr:acksPolicy intraMessageThreshold="0"/>
 </wsrm-mgr:destinationPolicy>
 </wsrm-mgr:reliableMessaging>
 </cxft:features>
</cxft:bus>
<http:destination
name="{http://apache.org/hello_world_soap_http}GreeterPort.http-destination"></http:de
stination>
 <httpj:engine-factory>
 <httpj:engine port="8081">
 <httpj:tlsServerParameters>
 <sec:keyManagers keyPassword="wso2carbon">
 <sec:keyStore file="/path/to/file/wso2carbon.jks"
password="wso2carbon" type="JKS"/>
 </sec:keyManagers>
 <sec:trustManagers>
 <sec:keyStore file="/path/to/file/client-truststore.jks"
password="wso2carbon" type="JKS"/>
 </sec:trustManagers>
 <sec:cipherSuitesFilter>
 <!-- these filters ensure that a ciphersuite with
 export-suitable or null encryption is used,
 but exclude anonymous Diffie-Hellman key change as
 this is vulnerable to man-in-the-middle attacks -->
 <sec:include>.*_EXPORT_.*</sec:include>
 <sec:include>.*_EXPORT1024_.*</sec:include>
 <sec:include>.*_WITH_DES_.*</sec:include>
 <sec:include>.*_WITH_AES_.*</sec:include>
 <sec:include>.*_WITH_NULL_.*</sec:include>
 <sec:exclude>.*_DH_anon_.*</sec:exclude>
 </sec:cipherSuitesFilter>
 <sec:clientAuthentication want="true" required="true"/>
 </httpj:tlsServerParameters>

```

```

</httpj:engine>
</httpj:engine-factory>
</beans>

```

### Sample CXF WS-RM Inbound

```

<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
 name="RM_INBOUND"
 sequence="RMIn"
 onError="fault"

 class="org.wso2.carbon.inbound.endpoint.ext.wsrm.InboundRMHttpListener"
 suspend="false">
 <parameters>
 <parameter name="inbound.cxf.rm.port">20940</parameter>
 <parameter
name="inbound.cxf.rm.config-file">repository/conf/cxf/server.xml</parameter>
 <parameter name="coordination">true</parameter>
 <parameter name="inbound.cxf.rm.host">127.0.0.1</parameter>
 <parameter name="inbound.behavior">listening</parameter>
 <parameter name="sequential">true</parameter>
 </parameters>
</inboundEndpoint>

```

The CXF WS-RM Inbound endpoint can be configured by specifying the following parameters:

- Sequence - The sequence that the message will be injected to.
- Error sequence - The sequence to be called if a fault occurs.
- Suspend - If the inbound listener should pause when accepting incoming requests, set this to true. If the inbound listener should not pause when accepting incoming requests, set this to false.
- inbound.cxf.rm.host - The host name.
- inbound.cxf.rm.port - The port to listen to.

### Note

When configuring a SSL enabled cxf\_ws\_rm inbound endpoint, the inbound.cxf.rm.port parameter should be set to the same value as the engine port number specified in the CXF spring configuration file saved in the <ESB\_HOME>/repository/conf/cxf directory.

For example, in the above CXF spring configuration, the engine port is 8081. This same port number should be specified as the inbound.cxf.rm.port in the cxf\_ws\_rm inbound endpoint configuration.

- inbound.cxf.rm.config-file - The path to the CXF Spring configuration file.
- enableSSL - Set to true if SSL is enabled in the CXF Spring configuration file.

### WebSocket Inbound Protocol

The WSO2 ESB WebSocket protocol implementation is based on the [WebSocket protocol](#), and allows full-duplex message mediation.

Following is a sample WebSocket inbound endpoint configuration:

```

<inboundEndpoint name="WebSocketListenerEP" onError="fault" protocol="ws"
sequence="TestIn" suspend="false">
<parameters>
<parameter name="inbound.ws.port">9091</parameter>
<parameter name="ws.outflow.dispatch.sequence">TestOut</parameter>
<parameter name="ws.client.side.broadcast.level">0</parameter>
<parameter name="ws.outflow.dispatch.fault.sequence">fault</parameter>
</parameters>
</inboundEndpoint>

```

#### WebSocket inbound endpoint parameters

Parameter	Description	Required
inbound.ws.port	The netty listener port on which the WebSocket inbound listens.	Yes
ws.client.side.broadcast.level	The client broadcast level that defines how WebSocket frames are broadcasted from the WebSocket inbound endpoint to the client. Broadcast happens based on the subscriber path client connected to the WebSocket inbound endpoint. The three possible levels are as follows: 0 - Only a unique client can receive the frame from a WebSocket inbound endpoint. 1 - All the clients connected with the same subscriber path receives the WebSocket frame. 2 - All the clients connected with the same subscriber path, except the one who publishes the frame to the inbound, receives the WebSocket frame.	Yes
ws.outflow.dispatch.sequence	The sequence for the back-end to client mediation.	Yes
ws.outflow.dispatch.fault.sequence	The fault sequence for the back-end to client mediation path.	Yes
ws.boss.thread.pool.size	The size of the netty boss pool.	No
ws.worker.thread.pool.size	The size of the worker thread pool.	No
ws.subprotocol.handler.class	The custom subprotocol handler classes separated by a semicolon.	No

#### Secure WebSocket Inbound Protocol

The WSO2 ESB secure WebSocket inbound protocol implementation is based on the [WebSocket protocol](#), and allows full-duplex, secure message mediation.

Following is a sample secure WebSocket inbound endpoint configuration:

```

<inboundEndpoint name="SecureWebSocketEP" onError="fault" protocol="wss"
sequence="TestIn" suspend="false">
<parameters>
<parameter name="inbound.ws.port">9091</parameter>
<parameter name="ws.client.side.broadcast.level">0</parameter>
<parameter name="ws.outflow.dispatch.sequence">TestOut</parameter>
<parameter name="ws.outflow.dispatch.fault.sequence">fault</parameter>
<parameter
name="wss.ssl.key.store.file">repository/resources/security/wso2carbon.jks</parameter>
<parameter name="wss.ssl.key.store.pass">wso2carbon</parameter>
<parameter
name="wss.ssl.trust.store.file">repository/resources/security/client-truststore.jks</p
arameter>
<parameter name="wss.ssl.trust.store.pass">wso2carbon</parameter>
<parameter name="wss.ssl.cert.pass">wso2</parameter>
</parameters>
</inboundEndpoint>

```

#### WebSocket inbound endpoint parameters

Parameter	Description	Required
inbound.ws.port	The netty listener port on which the WebSocket inbound listens.	Yes
ws.client.side.broadcast.level	The client broadcast level that defines how WebSocket frames are broadcasted from the WebSocket inbound endpoint to the client. Broadcast happens based on the subscriber path client connected to the WebSocket inbound endpoint. The three possible levels are as follows: 0 - Only a unique client can receive the frame from a WebSocket inbound endpoint. 1 - All the clients connected with the same subscriber path receives the WebSocket frame. 2 - All the clients connected with the same subscriber path, except the one who publishes the frame to the inbound, receives the WebSocket frame.	Yes
ws.outflow.dispatch.sequence	The sequence for the back-end to client mediation.	Yes
ws.outflow.dispatch.fault.sequence	The fault sequence for the back-end to client mediation path.	Yes
wss.ssl.key.store.file	The keystore location where keys are stored.	Yes
wss.ssl.key.store.pass	The password to access the keystore file.	Yes
wss.ssl.trust.store.file	The truststore location where keys are stored.	Yes
wss.ssl.trust.store.pass	The password to access the truststore file.	Yes
wss.ssl.cert.pass	The SSL certificate password.	Yes
ws.boss.thread.pool.size	The size of the netty boss pool.	No

ws.worker.thread.pool.size	The size of the worker thread pool.	No
ws.subprotocol.handler.class	The custom subprotocol handler classes separated by a semicolon.	No

## Polling Inbound Endpoints

A polling inbound endpoint polls periodically for data and when data is available the data is injected to a given sequence. For example, the JMS inbound endpoint checks the JMS queue periodically for messages and when a message is available that message is injected to a specified sequence. Polling inbound endpoints support one way operations and are asynchronous.

### **Common polling inbound endpoint parameters**

The following parameters are common to all polling inbound endpoints:

Parameter Name	Description	Required	Possible Values	Default Value
interval	The polling interval for the inbound endpoint to execute each cycle. This value is set in milliseconds.	No	A positive integer	-
coordination	This parameter only applicable in a cluster environment. In a cluster environment an inbound endpoint will only be executed in worker nodes. If set to <code>true</code> in a cluster setup, this will run the inbound only in a single worker node. Once the running worker is down the inbound starts on another available worker in the cluster.	No	true or false	false

See the following topics for detailed information on each polling inbound endpoint available with WSO2 ESB:

- [File Inbound Protocol](#)
- [JMS Inbound Protocol](#)
- [Kafka Inbound Protocol](#)

For information on how to create a custom inbound endpoint by extending the polling inbound endpoint behaviour, see [Creating a custom polling inbound endpoint](#).

### **File Inbound Protocol**

The WSO2 ESB file inbound protocol is a multi-tenant capable alternative to WSO2 ESB VFS transport. The ESB file inbound protocol uses the [VFS transport](#) to process files in a specified source directory. After processing the files, it moves them to a specified location or deletes them. Note that files cannot remain in the source directory after processing or they will be processed again, so if you need to maintain these files or keep track of the files that are processed, specify the option to move them instead of deleting them after processing.

Security

The file inbound protocol supports the **FTPS protocol** with **Secure Sockets Layer (SSL)**. The configuration is identical to other protocols. The only difference is the URL prefixes and parameters. For more information, see [VFS URL parameters](#).

Failure tracking

To track failures in file processing that can occur when a resource becomes unavailable, the VFS transport creates and maintains a failed records file. This text file contains a list of files that failed to process. When a failure occurs, an entry with the failed file name and timestamp is logged in the text file. When the next polling iteration occurs, the VFS transport checks each file against the failed records file, and if a file is listed as a failed record, it will skip processing and schedule a move task to move that file.

```

<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
 name="file" sequence="request"
 onError="fault"
 protocol="file"
 suspend="false">
 <parameters>
 <parameter name="interval">1000</parameter>
 <parameter name="sequential">true</parameter>
 <parameter name="coordination">true</parameter>
 <parameter name="transport vfs.ActionAfterProcess">MOVE</parameter>
 <parameter name="transport.PollInterval">10</parameter>
 <parameter
 name="transport vfs.MoveAfterProcess">file:///home/user/test/out</parameter>
 <parameter name="transport vfs.FileURI">file:///home/user/test/in</parameter>
 <parameter
 name="transport vfs.MoveAfterFailure">file:///home/user/test/failed</parameter>
 <parameter name="transport vfs.FileNamePattern">.*.txt</parameter>
 <parameter name="transport vfs.ContentType">text/plain</parameter>
 <parameter name="transport vfs.ActionAfterFailure">MOVE</parameter>
 </parameters>
 </inboundEndpoint>

```

#### VFS service-level parameters

The VFS transport does not have any global parameters to be configured. Rather, it has a set of service-level parameters that must be specified for each proxy service that uses the VFS transport. For information on how to configure the file inbound protocol for FTP, SFTP and FILE connections, see [Configuring File Inbound Protocol for FTP, SFTP and FILE Connections](#).

Parameter	Description
transport.vfs.FileURI	The URI where the files you want to process are located. You URL.
transport.vfs.ContentType	Content type of the files processed by the transport. To specify content type with a semi-colon and the character set. For example, <parameter name="transport.vfs.ContentType">text/plain; charset=UTF-8</parameter>. When writing a file, you can set a different encoding with the Content-Type header.
transport.vfs.FileNamePattern	If the VFS listener should process only a subset of the files available in the directory. You can use the <parameter name="transport.vfs.FileNamePattern">.*.txt</parameter> parameter to select those files by name using a regular expression.
transport.vfs.ActionAfterProcess	Whether to move or delete the files after the transport has processed them.
transport.vfs.ActionAfterFailure	Whether to move or delete the files if a failure occurs.
transport.vfs.MoveAfterProcess	Where to move the files after processing if the value specified is MOVE .
transport.vfs.MoveAfterFailure	Where to move the files after processing if the value specified is MOVE .

<code>transport.vfs.ReplyFileURI</code>	The location where reply files should be written by the transpo
<code>transport.vfs.ReplyFileName</code>	The name for reply files written by the transport.
<code>transport.vfs.MoveTimestampFormat</code>	The pattern/format of the timestamp added to file names as pr
<code>transport.vfs.Streaming</code>	Whether files should be transferred in streaming mode, which
<code>transport.vfs.ReconnectTimeout</code>	Reconnect timeout value in seconds to be used in case of an
<code>transport.vfs.MaxRetryCount</code>	Maximum number of retry attempts in case of errors.
<code>transport.vfs.MoveAfterFailedMove</code>	The location to move a failed file.
<code>transport.vfs.FailedRecordsFileName</code>	The name of the file that maintains the list of failed files.
<code>transport.vfs.FailedRecordsFile Destination</code>	The location to store the failed records file.
<code>transport.vfs.MoveFailedRecord TimestampFormat</code>	The time stamp format for entries in the failed records file. The that failed and the timestamp of its failure.
<code>transport.vfs.FailedRecordNext RetryDuration</code>	The time in milliseconds to wait before retrying the move task.
<code>transport.vfs.Locking</code>	By default, file locking is enabled in the VFS transport. This pa per service basis. You can also disable locking globally by s selectively enabling locking only for a set of services.
<code>transport.vfs.FileProcessCount</code>	This parameter allows you to throttle the VFS listener by proce you want to process in each batch. If you specify a value for th
<code>transport.vfs.FileProcessInterval</code>	The interval in milliseconds between file processing batches.
<code>transport.vfs.DistributedLock</code>	This applies only in cluster deployments. Set to true if you ne same file simultaneously.
<code>transport.vfs.DistributedTimeout</code>	The timeout period in seconds for the distributed lock.
<code>transport.vfs.AutoLockRelease</code>	Set to true if you need to release locking in order to avoid files works together with the <code>transport.vfs.AutoLockReleas eSameNode</code> parameters.
<code>transport.vfs.AutoLockReleaseInterval</code>	Lock release interval in milliseconds.
<code>transport.vfs.LockReleaseSameNode</code>	Set to true if you need to release the locks only accrued by t If this is set to false, locks accrued by other nodes will be rel <code>transport.vfs.AutoLockReleaseInterval</code> .
<code>transport.vfs.FileSortAttribute</code>	The attribute by which the files should be sorted and processe
<code>transport.vfs.FileSortAscending</code>	The sort order to sort and process the files. If set to true files ribute you specify in <code>transport.vfs.FileSortAttribut</code>

transport.vfs.CreateFolder	Set to true to create a folder if a folder does not exist when no file is found.
transport.vfs.SubFolderTimestampFormat	The pattern/format of the timestamps added to the folder structure. Set transport.vfs.CreateFolder to true to in order to specify a value.
transport.vfs.Build	Set to true if you need to build the content inside the file before there is a build error, the file will not be injected to the mediation engine.

## Note

If you specify the `transport.vfs.FileProcessCount` parameter you do not need to specify the `transport.vfs.FileProcessInterval` parameter in a configuration, and vice versa. This is because the `transport.vfs.FileProcessCount` parameter and the `transport.vfs.FileProcessInterval` parameter cannot be used at the same time.

### Samples

For a sample that demonstrates how to use the file system as an input medium via the inbound file listener, see [Sample 900: Inbound Endpoint File Protocol Sample \(VFS\)](#).

### Configuring File Inbound Protocol for FTP, SFTP and FILE Connections

The following section describes how to configure the file inbound protocol for FTP, SFTP and FILE connections.

- To configure the file inbound protocol for FTP connections, you should specify the URL as `ftp://{{username}}:{{password}}@{{hostname/ip_address}}/{{source_filepath}}`

```
ftp://admin:pass@localhost/orders
```

- To configure the file inbound protocol for SFTP connections, you should specify the URL as `sftp://{{username}}:{{password}}@{{hostname/ip_address}}/{{source_filepath}}`

```
sftp://admin:pass@localhost/orders
```

If the password contains special characters, these characters will need to be replaced with their hexadecimal representation.

- To configure the file inbound protocol for FILE connections, you should specify the URL as `file://{{local_file_system_path}}`

```
file:///home/user/test/in
```

### Configuring a Proxy Server over FTP and SFTP Connections

Proxy server specific parameters can be set as URL query parameters. For example, to use Proxy over FTP, you could specify the URL as follows:

```
ftp://username:password@127.0.0.1/home/wso2/res?proxyServer=127.0.0.1&proxyPort=3128&proxyUsername=proxyuser&proxyPassword=proxyPass&timeout=2500&retryCount=3
```

Following are the URL parameters you can set:

Parameter	Description	Possible Values	Default Value
proxyServer	IP Address of the proxy server	127.0.0.1	false
proxyPort	Running port of the proxy server	1328	false
proxyUsername	User name of the proxy server		false
proxyPassword	Password of the proxy server		false
timeout	Connection timeout in milliseconds	1000	5000
retryCount	No of retry attempts if the connection timeout	3	5

### JMS Inbound Protocol

The WSO2 ESB JMS inbound protocol is a multi-tenant capable alternative to the WSO2 ESB JMS transport. The JMS inbound protocol implementation requires an active JMS server instance to be able to receive messages, and you need to place the client JARs for your JMS server in the ESB classpath.

The recommended JMS servers are [WSO2 Message Broker](#) or Apache ActiveMQ, but other implementations such as Apache Qpid and Tibco are also supported.

- To configure the JMS inbound protocol with WSO2 Message Broker , see [Configuring the JMS Inbound Protocol with WSO2 Message Broker](#).
- To configure the JMS inbound protocol with ActiveMQ, see [Configuring the JMS Inbound Protocol with ActiveMQ](#).

Configuration parameters for a JMS inbound endpoint are XML fragments that represent JMS connection factories.

Following is a sample JMS inbound endpoint configuration:

```

<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
 name="jms" sequence="request"
 onError="fault"
 protocol="jms"
 suspend="false">
 <parameters>
 <parameter name="interval">1000</parameter>
 <parameter name="coordination">true</parameter>
 <parameter name="transport.jms.Destination">ordersQueue</parameter>
 <parameter name="transport.jms.CacheLevel">1</parameter>
 <parameter
name="transport.jms.ConnectionFactoryJNDIName">QueueConnectionFactory</parameter>
 <parameter name="sequential">true</parameter>
 <parameter
name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFact
ory</parameter>
 <parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
 <parameter
name="transport.jms.SessionAcknowledgement">AUTO_ACKNOWLEDGE</parameter>
 <parameter name="transport.jms.SessionTransacted">false</parameter>
 <parameter name="transport.jms.ConnectionFactoryType">queue</parameter>
 </parameters>
</inboundEndpoint>

```

The parameters `interval` and `coordination` are common to all polling inbound endpoints. For descriptions of these parameters, see [Common polling inbound endpoint parameters](#).

JMS inbound endpoint parameters

Parameter Name	Description	Required
<code>java.naming.factory.initial</code>	The JNDI initial context factory class. The class must implement the <code>java.naming.spi.InitialContextFactory</code> interface.	Yes
<code>java.naming.provider.url</code>	The URL of the JNDI provider.	Yes
<code>transport.jms.ConnectionFactoryJNDIName</code>	The JNDI name of the connection factory.	Yes
<code>sequential</code>	Whether the messages need to be polled and injected sequentially or not.	Yes
<code>transport.jms.ConnectionFactoryType</code>	The type of the connection factory.	No
<code>transport.jms.Destination</code>	The JNDI name of the destination.	No
<code>transport.jms.SessionTransacted</code>	Whether the JMS session should be transacted or not.	No
<code>transport.jms.SessionAcknowledgement</code>	The JMS session acknowledgment mode.	No
<code>transport.jms.CacheLevel</code>	The JMS resource cache level.	No
<code>transport.jms.UserName</code>	The JMS connection username.	No

<code>transport.jms.Password</code>	The JMS connection password.	No
<code>transport.jms.JMSSpecVersion</code>	The JMS API version.	No
<code>transport.jms.SubscriptionDurable</code>	Whether the connection factory is subscription durable or not.	No
<code>transport.jms.DurableSubscriberClientID</code>	The <code>ClientId</code> parameter when using durable subscriptions.	Requires <code>transport.jms</code>
<code>transport.jms.DurableSubscriberName</code>	The name of the durable subscriber.	Requires <code>transport.jms</code>
<code>transport.jms.MessageSelector</code>	Message selector implementation.	No
<code>transport.jms.ReceiveTimeout</code>	The time to wait for a JMS message during polling. Set this parameter value to a negative integer to wait indefinitely. Set it to zero to prevent waiting.	No
<code>transport.jms.ContentType</code>	How the inbound listener should determine the content type of received messages. Priority is always given to the JMS message type.	No
<code>transport.jms.ContentTypeProperty</code>	Get the content type from message property.	No
<code>transport.jms.ReplyDestination</code>	The destination that the response generated by the back-end service is stored.	No
<code>transport.jms.PubSubNoLocal</code>	Whether messages should be published via the same connection that they were received.	No
<code>transport.jms.SharedSubscription</code>	If set to true, messages will be forwarded to only one of the consumers and consumers will share the messages that are published to the topic.	No
<code>pinnedServers</code>	List of synapse server names separated by commas or spaces where this inbound endpoint should be deployed. If there is no pinned server list, the inbound endpoint configuration will be deployed in all server instances.	No
<code>transport.jms.ConcurrentConsumers</code>	Number of concurrent threads to be started to consume messages when polling.	No
<code>transport.jms.retry.duration</code>	The retry interval to reconnect to the JMS server.	No

**Note**

To subscribe to topics, set the value of `transport.jms.CacheLevel` to 3.

**Samples**

For a sample that demonstrates how one way message bridging from JMS to HTTP can be done using the JMS inbound endpoint, see [Sample 901: Inbound Endpoint JMS Protocol Sample](#).

## Configuring the JMS Inbound Protocol with WSO2 Message Broker

This section describes how to configure the ESB's JMS inbound protocol with WSO2 Message Broker (WSO2 MB).

### Note

When you configure WSO2 ESB's JMS inbound protocol with WSO2 MB, the recommended version of WSO2 MB to be used is WSO2 MB 3.0.0. We do not recommend the use of WSO2 MB 2.2.0 or lower.

Follow the steps below to configure the JMS inbound protocol with WSO2 MB 3.0.0

Setting up WSO2 MB

1. Download and install WSO2 MB. For instructions on how to download and install WSO2 MB, see [Getting Started with WSO2 MB](#).

The unzipped WSO2 MB distribution folder will be referred to as <MB\_HOME> throughout the documentation.

### Note

It is not possible to start multiple WSO2 products with their default configurations simultaneously in the same environment. Since all WSO2 products use the same port in their default configuration, there will be port conflicts. Therefore, to avoid port conflicts, apply a port offset in the <MB\_HOME>/repository/conf/carbon.xml file by changing the offset value to 1. For example,

```
<Ports>
 <!-- Ports offset. This entry will set the value of the ports defined
 below to
 the define value + Offset.
 e.g. Offset=2 and HTTPS port=9443 will set the effective HTTPS port to
 9445
 -->
 <Offset>1</Offset>
```

2. Open a command prompt (or a shell in Linux) and go to the <MB\_HOME>\bin directory.
3. Start the Message Broker by executing sh wso2server.sh (on Linux/OS X) or wso2server.bat (on Windows).

Setting up WSO2 ESB

1. If you have not already done so, see [Getting Started with WSO2 ESB](#) for details on installing and running WSO2 ESB.
2. Configure the JMS inbound listener. Following is a sample JMS inbound listener configuration:

```

<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse" name="jms"
sequence="request" onError="fault" protocol="jms" suspend="false">
 <parameters>
 <parameter name="interval">1000</parameter>
 <parameter name="transport.jms.CacheLevel">1</parameter>
 <parameter
name="transport.jms.ConnectionFactoryJNDIName">TopicConnectionFactory</parameter>
 <parameter name="sequential">true</parameter>
 <parameter
name="java.naming.factory.initial">org.wso2.andes.jndi.PropertiesFileInitialConte
xtFactory</parameter>
 <parameter
name="java.naming.provider.url">repository/conf/jndi.properties</parameter>
 <parameter
name="transport.jms.SessionAcknowledgement">AUTO_ACKNOWLEDGE</parameter>
 <parameter name="transport.jms.SessionTransacted">false</parameter>
 <parameter name="transport.jms.ConnectionFactoryType">topic</parameter>
 </parameters>
</inboundEndpoint>

```

For more information on the JMS configuration parameters used in the code segments above, see [JMS Connection Factory Parameters](#).

3. Copy the following JAR files from the <MB\_HOME>/client-lib folder to <ESB\_HOME>/repository/components/lib folder.
  - andes-client-3.0.1.jar
  - geronimo-jms\_1.1\_spec-1.1.0.wso2v1.jar
  - org.wso2.securevault-1.0.0-wso2v2.jar
4. Open <ESB\_HOME>/repository/conf/jndi.properties file and make a reference to the running Message Broker as specified below:
  - Use **carbon** as the virtual host.
  - Define a queue named JMSMS.
  - Comment out the topic, since it is not required in this scenario. However, in order to avoid getting the javax.naming.NameNotFoundException:TopicConnectionFactory exception during server startup, make a reference to the Message Broker from the TopicConnectionFactory as well.

For example:

```

register some connection factories
connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.QueueConnectionFactory =
amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5673'
connectionfactory.TopicConnectionFactory =
amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5673'
register some queues in JNDI using the form
queue.[jndiName] = [physicalName]
queue.JMSMS=JMSMS
queue.StockQuotesQueue = StockQuotesQueue

```

5. Ensure that WSO2 Message Broker is running, and then open a command prompt (or a shell in Linux) and go to the <ESB\_HOME>\bin directory.
6. Start the ESB server by executing  
`sh wso2server.sh -Dqpid.dest_syntax=BURL (on Linux/OS X) or`

wso2server.bat -Dqpid.dest\_syntax=BURL (on Windows).

Now you have an instance of WSO2 Message Broker and a WSO2 ESB inbound endpoint configured, up and running.

## Configuring the JMS Inbound Protocol with ActiveMQ

This section describes how to configure WSO2 ESB JMS inbound protocol with ActiveMQ.

Follow the instructions below to set up and configure Apache ActiveMQ as the JMS server:

1. Download and set up Apache ActiveMQ. For more information, see [Installation Prerequisites](#)
2. Set up WSO2 ESB. For information on getting the ESB set up, see [Installation Guide](#).

### Note

ActiveMQ should be up and running before starting the ESB.

3. Copy the following client libraries from the <AMQ\_HOME>/lib directory to the <ESB\_HOME>/repository/components/lib directory.

#### For ActiveMQ 5.8.0 and above

- activemq-broker-5.8.0.jar
- activemq-client-5.8.0.jar
- geronimo-jms\_1.1\_spec-1.1.1.jar
- geronimo-j2ee-management\_1.1\_spec-1.0.1.jar

### Note

If you are using ActiveMQ version 5.8.0 or later, copy [hawtbuf-1.2.jar](#) to the <ESB\_HOME>/repository/components/lib directory.

#### For earlier versions of ActiveMQ

- activemq-core-5.5.1.jar
- geronimo-j2ee-management\_1.0\_spec-1.0.jar
- geronimo-jms\_1.1\_spec-1.1.1.jar

4. Next, configure the inbound listener in the ESB.

Configuring the JMS inbound listener

Following is a sample JMS inbound listener configuration:

```

<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse" name="jms" sequence="request"
onError="fault" protocol="jms" suspend="false">
 <parameters>
 <parameter name="interval">1000</parameter>
 <parameter name="transport.jms.Destination">ordersQueue</parameter>
 <parameter name="transport.jms.CacheLevel">1</parameter>
 <parameter
name="transport.jms.ConnectionFactoryJNDIName">QueueConnectionFactory</parameter>
 <parameter name="sequential">true</parameter>
 <parameter
name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFact
ory</parameter>
 <parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
 <parameter
name="transport.jms.SessionAcknowledgement">AUTO_ACKNOWLEDGE</parameter>
 <parameter name="transport.jms.SessionTransacted">false</parameter>
 <parameter name="transport.jms.ConnectionFactoryType">queue</parameter>
 </parameters>
</inboundEndpoint>

```

## Note

For details on the JMS configuration parameters used in the sample configuration above, see [JMS Connection Factory Parameters](#).

The sample configuration above does not address the problem of transient failures of the ActiveMQ message broker. For example, if we consider a scenario where the ActiveMQ broker goes down for some reason and comes back up after a while. The ESB will not reconnect to ActiveMQ but instead it will throw errors when requests are sent to the ESB until it is restarted.

In order to tackle this issue you need to specify the following as the `java.naming.provider.url` parameter value.

`failover:tcp://localhost:61616`

Setting this as the value for the `java.naming.provider.url` parameter will make sure that re-connection takes place when ActiveMQ is up and running. The failover prefix is associated with the failover transport of ActiveMQ. For more information, see [Failover Transport](#).

Now you have an instances of ActiveMQ and WSO2 ESB inbound endpoint configured, up and running.

### Kafka Inbound Protocol

Kafka is a distributed, partitioned, replicated commit log service. It provides the functionality of a messaging system.Kafka maintains feeds of messages in topics. Producers write data to topics and consumers read from topics. For more information on Apache Kafka, go to [Apache Kafka documentation](#).

WSO2 ESB kafka inbound endpoint acts as a message consumer. It creates a connection to zookeeper and requests messages for a topic, topics or topic filters.

In order to use the kafka inbound endpoint, you need to download and install [Apache Kafka](#). The recommend version is `kafka_2.9.2-0.8.1.1` or later.

To configure the kafka inbound endpoint, copy the following client libraries from the <KAFKA\_HOME>/lib directory to the <ESB\_HOME>/repository/components/lib directory.

- kafka\_2.9.2-0.8.1.1.jar
- scala-library-2.9.2.jar
- zkclient-0.3.jar
- zookeeper-3.3.4.jar
- metrics-core-2.2.0.jar

### Note

If you use kafka\_2.x.x-0.8.2.0 or later, you need to add kafka-clients-0.8.x.x.jar also to the <ESB\_HOME>/repository/components/lib directory.

Configuration parameters for a kafka inbound endpoint are XML fragments that represent various properties.

Following is a sample high level kafka configuration that can be used to consume messages using the specified topic or topics:

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
 name="KafkaListenerEP"
 sequence="requestHandlerSeq"
 onError="inFault"
 protocol="kafka"
 suspend="false">
<parameters>
 <parameter name="interval">100</parameter>
 <parameter name="coordination">true</parameter>
 <parameter name="sequential">true</parameter>
 <parameter name="zookeeper.connect">localhost:2181</parameter>
 <parameter name="consumer.type">highlevel</parameter>
 <parameter name="content.type">application/xml</parameter>
 <parameter name="topics">test,sampletest</parameter>
 <parameter name="group.id">test-group</parameter>
</parameters>
</inboundEndpoint>
```

Kafka inbound endpoint parameters for a high level configuration

Parameter	Description	Required	Possible Values	Default Value
zookeeper.connect	The host and port of a ZooKeeper server (hostname :port).	Yes	localhost:2181	
consumer.type	The consumer configuration type. This can either be simple or highlevel.	Yes	highlevel, simple	
interval	The polling interval for the inbound endpoint to poll the messages.	Yes		

coordination	If set to true in a cluster setup, this will run the inbound only in a single worker node.	Yes	true, false	true
sequential	The behaviour when executing the given sequence.	Yes	true, false	true
topics	The category to feed the messages. A high level kafka configuration can have more than one topic. You can specify multiple topic names as comma separated values.	Yes		
content.type	The content of the message.	Yes	application/xml, application/json	
group.id	If all the consumer instances have the same consumer group, this works as a traditional queue balancing the load over the consumers.  If all the consumer instances have different consumer groups, this works as publish-subscribe and all messages are broadcast to all consumers.	Yes		
thread.count	The number of threads.	No		1
consumer.id	The id of the consumer.	No		null
socket.timeout.ms	The socket timeout for network requests.	No		30 *
socket.receive.buffer.bytes	The socket receive buffer for network requests.	No		64 *
fetch.message.max.bytes	The number of bytes of messages to attempt to fetch for each topic-partition in each fetch request.	No		1024 1024
num.consumer.fetchers	The number fetcher threads used to fetch data.	No		1
auto.commit.enable	The committed offset to be used as the position from which the new consumer will begin when the process fails.	No	true, false	true
auto.commit.interval.ms	The frequency in ms that the consumer offsets are committed to zookeeper.	No		60 *

queued.max.message.chunks	The maximum number of message chunks buffered for consumption. Each chunk can go up to the value specified in fetch.message.max.bytes.	No		2
rebalance.max.retries	The maximum number of retry attempts.	No		4
fetch.min.bytes	The minimum amount of data the server should return for a fetch request.	No		1
fetch.wait.max.ms	The maximum amount of time the server will block before responding to the fetch request when sufficient data is not available to immediately serve fetch.min.bytes.	No		100
rebalance.backoff.ms	The backoff time between retries during rebalance.	No		2000
refresh.leader.backoff.ms	The backoff time to wait before trying to determine the leader of a partition that has just lost its leader.	No		200
auto.offset.reset	<p>Set this to one of the following values based on what you need to do when there is no initial offset in ZooKeeper or if an offset is out of range.</p> <p>smallest - Automatically reset the offset to the smallest offset.</p> <p>largest - Automatically reset the offset to the largest offset.</p> <p>anything else - Throw an exception to the consumer.</p>	No	smallest, largest, anything else	large
consumer.timeout.ms	The timeout interval after which a timeout exception is to be thrown to the consumer if no message is available for consumption. It is a good practice to set this value lower than the interval of the Kafka inbound endpoint.	No		3000
exclude.internal.topics	Set to true if messages from internal topics such as offsets should be exposed to the consumer.	No	true, false	true

partition.assignment.strategy	The partitions assignment strategy to be used when assigning partitions to consumer streams.	No	range, roundrobin	range
client.id	The user specified string sent in each request to help trace calls.	No		value, group
zookeeper.session.timeout.ms	The ZooKeeper session timeout value in milliseconds.	No		6000
zookeeper.connection.timeout.ms	The maximum time in milliseconds that the client should wait while establishing a connection to ZooKeeper.	No		6000
zookeeper.sync.time.ms	The time difference in milliseconds that a ZooKeeper follower can be behind a ZooKeeper leader.	No		2000
offsets.storage	The offsets storage location.	No	zookeeper, kafka	zookeeper
offsets.channel.backoff.ms	The backoff period in milliseconds when reconnecting the offsets channel or retrying failed offset fetch/commit requests.	No		1000
offsets.channel.socket.timeout.ms	The socket timeout in milliseconds when reading responses for offset fetch/commit requests.	No		1000
offsets.commit.max.retries	The maximum retry attempts allowed. If a consumer metadata request fails for any reason, retry takes place but does not have an impact on this limit.	No		5
dual.commit.enabled	If offsets.storage is set to kafka, the commit offsets can be dual to ZooKeeper. Set this to true if you need to perform migration from zookeeper-based offset storage to kafka-based offset storage.	No	true, false	true

Following is a sample high level kafka configuration that can be used to consume messages using the topics filter, which can either be a white list or a black list :

```

<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
 name="KafkaListenerEP"
 sequence="requestHandlerSeq"
 onError="inFault"
 protocol="kafka"
 suspend="false">
 <parameters>
 <parameter name="interval">100</parameter>
 <parameter name="coordination">true</parameter>
 <parameter name="sequential">true</parameter>
 <parameter name="zookeeper.connect">localhost:2181</parameter>
 <parameter name="consumer.type">highlevel</parameter>
 <parameter name="content.type">application/xml</parameter>
 <parameter name="topic.filter">test</parameter>
 <parameter name="filter.from.whitelist">true</parameter>
 <parameter name="group.id">test-group</parameter>
 </parameters>
</inboundEndpoint>

```

In the above configuration, you will see that instead of using the `topics` parameter, the following parameters are set for topic filter consuming:

```

<parameter name="topic.filter">test</parameter>
<parameter name="filter.from.whitelist">true</parameter>

```

Following are the description for the parameters set in a topic filter consuming kafka configuration:

Parameter	Description	Required
<code>topic.filter</code>	The topic filter name.	Yes
<code>filter.from.whitelist</code>	If set to <code>true</code> , messages will be consumed from the whitelist(include) whereas if set to <code>false</code> messages will be consumed from the blacklist(exclude).	Yes

Following is a sample low level kafka configuration that can be used to consume the messages from a specific server and specific partition so that the messages are limited.:

```

<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
 name="KafkaListenerEP"
 sequence="requestHandlerSeq"
 onError="inFault"
 protocol="kafka"
 interval="1000"
 suspend="false">
 <parameters>
 <parameter name="zookeeper.connect">localhost:2181</parameter>
 <parameter name="group.id">test-group</parameter>
 <parameter name="content.type">application/xml</parameter>
 <parameter name="consumer.type">simple</parameter>
 <parameter name="simple.max.messages.to.read">5</parameter>
 <parameter name="simple.topic">test</parameter>
 <parameter name="simple.brokers">localhost</parameter>
 <parameter name="simple.port">9092</parameter>
 <parameter name="simple.partition">1</parameter>
 <parameter name="interval">1000</parameter>
 </parameters>
</inboundEndpoint>

```

Kafka inbound endpoint parameters for a low level configuration

Parameter	Description	Required
simple.topic	The category to feed the messages.	Yes
simple.brokers	The specific Kafka broker name.	Yes
simple.port	The specific Kafka server port number.	Yes
simple.partition	The partition of the topic.	Yes
simple.max.messages.to.read	The maximum number of messages to receive.	Yes

#### Samples

For a sample that demonstrates how one way message bridging from Kafka to HTTP can be done using the kafka inbound endpoint, see [Sample 904: Kafka Inbound Endpoint Sample](#).

#### Event-Based Inbound Endpoints

An event-based inbound endpoint polls only once to establish a connection with the remote server and then consumes events.

#### **Common event-based endpoint parameters**

The following parameter is common to all event-based inbound endpoints:

Parameter Name	Description	Required	Possible Values	Default Value
coordination	This parameter is only applicable in a cluster environment. In a cluster environment an inbound endpoint will only be executed in worker nodes. If this parameter is set to true in a cluster environment, the inbound will only be executed in a single worker node. Once the running worker node is down the inbound will start on another available worker node in the cluster.	No	true or false	false

See the following topics for detailed information on each event-based inbound endpoint available with WSO2 ESB:

- [MQTT Inbound Protocol](#)
- [RabbitMQ Inbound Protocol](#)

For information on how to create a custom inbound endpoint by extending the event-based inbound endpoint behaviour, see [Creating a custom event-based inbound endpoint](#).

#### ***MQTT Inbound Protocol***

MQ Telemetry Transport (MQTT) is a lightweight broker-based publish/subscribe messaging protocol, designed to be open, simple, lightweight and easy to implement. These characteristics make it ideal for use in constrained environments.

For example,

- When the network is expensive, has low bandwidth or is unreliable.
- When running on an embedded device with limited processor or memory resources.

To configure the MQTT inbound endpoint, you need to specify XML fragments that represents various parameters.

Following is a sample MQTT inbound endpoint configuration that can subscribe to messages using the specified topic:

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse" name="Test" sequence="TestIn"
onError="fault" protocol="mqtt" suspend="false">
 <parameters>
 <parameter name="sequential">true</parameter>
 <parameter name="mqtt.connection.factory">mqttFactory</parameter>
 <parameter name="mqtt.server.host.name">localhost</parameter>
 <parameter name="mqtt.server.port">1883</parameter>
 <parameter name="mqtt.topic.name">esb.test2</parameter>
 <parameter name="mqtt.subscription.qos">2</parameter>
 <parameter name="content.type">application/xml</parameter>
 <parameter name="mqtt.session.clean">false</parameter>
 <parameter name="mqtt.ssl.enable">false</parameter>
 <parameter name="mqtt.subscription.username">client</parameter>
 <parameter name="mqtt.subscription.password">e13</parameter>
 <parameter name="mqtt.temporary.store.directory">mly</parameter>
 <parameter name="mqtt.reconnection.interval">5</parameter>
 </parameters>
</inboundEndpoint>
```

MQTT inbound endpoint parameters

Parameter	Description	Required

sequential	The behaviour when executing the given sequence.	Yes
coordination	In a cluster setup, if set to true, the inbound endpoint will run only in a single worker node. If set to false, it will run on all worker nodes.	Yes
suspend	If set to true, the inbound listener will not accept incoming request messages from clients and will not inject messages to the synapse message mediation. If set to false, incoming messages will be accepted.	Yes
mqtt.connection.factory	Name of the connection factory.	Yes
mqtt.server.host.name	Address of the message broker (eg., localhost).	Yes
mqtt.server.port	Port of the message broker (e.g., 1883).	Yes
mqtt.topic.name	MQTT topic to which the message should be published.	Yes
mqtt.subscription.qos	<p>The quality of service level that need to be maintained with the subscription. The quality of service level can be either 0,1 or 2.</p> <p>0 - Specifying this level ensures that the message delivery is efficient. However, specifying this level does not guarantee that the message will be delivered to its recipient.</p> <p>1 - Specifying this level ensures that the message is delivered at least once, but this can lead to messages being duplicated.</p> <p>2 - This is the highest level of quality of service. Specifying this guarantees that the message is delivered and that it is delivered only once.</p>	No
mqtt.client.id	The id of the client.	No
content.type	The content type of the message. (i.e., XML or JSON)	Yes
mqtt.session.clean	Whether the client and server should remember the state across restarts and reconnects.	No
mqtt.ssl.enable	Whether to use tcp connection or ssl connection.	No
mqtt.subscription.username	The username for the subscription.	No
mqtt.subscription.password	The password for the subscription.	No
mqtt.temporary.store.directory	Path of the directory to be used as the persistent data store for quality of service purposes.	No
mqtt.reconnection.interval	The retry interval to reconnect to the MQTT server.	No

Samples

For a sample that demonstrates how the MQTT connector publishes a message on a particular topic and how a MQTT client that is subscribed to that topic receives it, see [Sample 906: MQTT Inbound Endpoint Sample](#).

#### RabbitMQ Inbound Protocol

**AMQP** is a wire-level messaging protocol that describes the format of the data that is sent across the network. If a system or application can read and write AMQP, it can exchange messages with any other system or application that understands AMQP regardless of the implementation language.

Configuration parameters for a RabbitMQ inbound endpoint are XML fragments that represent various properties.

Following is a sample RabbitMQ inbound endpoint configuration that consumes a messages from a queue:

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse" name="RabbitMQConsumer" sequence="amqpSeq" onError="amqpErrorSeq" protocol="rabbitmq" suspend="false">
 <parameters>
 <parameter name="sequential">true</parameter>
 <parameter name="coordination">true</parameter>
 <parameter name="rabbitmq.connection.factory">AMQPConnectionFactory</parameter>
 <parameter name="rabbitmq.server.host.name">localhost</parameter>
 <parameter name="rabbitmq.server.port">5672</parameter>
 <parameter name="rabbitmq.server.user.name">guest</parameter>
 <parameter name="rabbitmq.server.password">guest</parameter>
 <parameter name="rabbitmq.queue.name">queue</parameter>
 <parameter name="rabbitmq.exchange.name">exchange</parameter>
 <parameter name="rabbitmq.connection.ssl.enabled">false</parameter>
 </parameters>
 </inboundEndpoint>
```

RabbitMQ inbound endpoint parameters

Parameter Name	Description	Required
sequential	The behaviour when executing the given sequence.	Yes
rabbitmq.connection.factory	Name of the connection factory.	Yes
rabbitmq.server.host.name	Host name of the server.	Yes
rabbitmq.server.port	The port number of the server.	Yes
rabbitmq.server.user.name	The user name to connect to the server.	Yes
rabbitmq.server.password	The password to connect to the server.	Yes
rabbitmq.queue.name	The queue name to send or consume messages.	Yes
rabbitmq.exchange.name	The name of the RabbitMQ exchange to which the queue is bound.	Yes

rabbitmq.server.virtual.host	The virtual host name of the server, if available.	No
rabbitmq.connection.ssl.enabled	Whether to use tcp connection or ssl connection.	No
rabbitmq.connection.ssl.keystore.location	The keystore location.	No
rabbitmq.connection.ssl.keystore.type	The keystore type.	No
rabbitmq.connection.ssl.keystore.password	The keystore password.	No
rabbitmq.connection.ssl.truststore.location	The truststore location.	No
rabbitmq.connection.ssl.truststore.type	The truststore type.	No
rabbitmq.connection.ssl.truststore.password	The truststore password.	No
rabbitmq.connection.ssl.version	The ssl version.	No
rabbitmq.queue.durable	Whether the queue should remain declared even if the broker restarts.	No
rabbitmq.queue.exclusive	Whether the queue should be exclusive or should be consumable by other connections.	No
rabbitmq.queue.auto.delete	Whether to keep the queue even if it is not being consumed anymore.	No
rabbitmq.queue.auto.ack	Whether to send back an acknowledgement.	No
rabbitmq.queue.routing.key	The routing key of the queue.	No
rabbitmq.queue.delivery.mode	The delivery mode of the queue (ie., Whether it is persistent or not).	No
rabbitmq.exchange.type	The type of the exchange.	No
rabbitmq.exchange.durable	Whether the exchange should remain declared even if the broker restarts.	No
rabbitmq.exchange.auto.delete	Whether to keep the queue even if it is not used anymore.	No
rabbitmq.factory.heartbeat	The period of time after which the connection should be considered dead.	No
rabbitmq.message.content.type	The content type of the message.	No

rabbitmq.connection.retry.count	Number of retries to reconnect the rabbitmq server.	No
rabbitmq.connection.retry.interval	Retry interval to reconnect the rabbitmq server.	No

#### Samples

For a sample that demonstrates how one way message bridging from RabbitMQ to HTTP can be done using the RabbitMQ inbound endpoint, see [Sample 907: Inbound Endpoint RabbitMQ Protocol Sample](#).

#### Custom Inbound Endpoint

WSO2 ESB supports several inbound endpoints, but there can be scenarios that require functionality not provided by the existing inbound endpoints. For example, you might need an inbound endpoint to connect to a certain back-end server or vendor specific protocol.

To support such scenarios, you can write your own custom inbound endpoint by further extending the inbound endpoint behaviour.

#### ***Custom listening inbound endpoint***

Following is a sample custom listening inbound endpoint configuration:

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse" name="custom_listener"
sequence="request" onError="fault"
 class="org.wso2.carbon.inbound.custom.listening.SampleListeningEP"
suspend="false">
 <parameters>
 <parameter name="sequential">true</parameter>
 <parameter name="inbound.behavior">listening</parameter>
 <parameter name="coordination">true</parameter>
 </parameters>
</inboundEndpoint>
```

In addition to the parameters provided in the above sample configuration, you can provide other required parameters based on the listening behaviour you need to implement.

#### ***Custom listening inbound endpoint parameters***

Parameter Name	Description	Required	Possible Values	Default Value
class	Name of the custom class implementation	Yes	A valid class name	n/a
sequence	Name of the sequence message that should be injected	Yes	A valid sequence name	n/a
onError	Name of the fault sequence that should be invoked in case of failure	Yes	A valid fault sequence name	n/a
inbound.behavior	The behaviour of the inbound endpoint	Yes	listening	n/a

You can download the maven artifact used in the sample custom listening inbound endpoint configuration above from [Custom listening inbound endpoint sample](#) .

You need to copy the built jar file to the <ESB\_HOME>/repository/components/lib directory and restart the ESB to load the class.

### **Custom polling inbound endpoint**

Following is a sample custom polling inbound endpoint configuration:

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse" name="class"
sequence="request" onError="fault"

class="org.wso2.carbon.inbound.custom.poll.SamplePollingClient" suspend="false">
<parameters>
 <parameter name="sequential">true</parameter>
 <parameter name="interval">2000</parameter>
 <parameter name="coordination">true</parameter>
</parameters>
</inboundEndpoint>
```

### **Custom polling inbound endpoint parameters**

Parameter Name	Description	Required	Possible Values	Default Value
class	Name of the custom class implementation	Yes	A valid class name	n/a

You can download the maven artifact used in the sample custom polling inbound endpoint configuration above from [Custom polling inbound endpoint sample](#).

You need to copy the built jar file to the <ESB\_HOME>/repository/components/lib directory and restart the ESB to load the class.

### **Custom event-based inbound endpoint**

Following is a sample custom event-based inbound endpoint configuration:

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse" name="custom_waiting"
sequence="request" onError="fault"
 class="org.wso2.carbon.inbound.custom.wait.SampleWaitingClient"
suspend="false">
<parameters>
 <parameter name="sequential">true</parameter>
 <parameter name="inbound.behavior">eventBased</parameter>
 <parameter name="coordination">true</parameter>
</parameters>
</inboundEndpoint>
```

In addition to the parameters provided in the above sample configuration, you can provide other required parameters based on the event-based behaviour you need to implement.

### **Custom event-based inbound endpoint parameters**

Parameter Name	Description	Required	Possible Values	Default Value
class	Name of the custom class implementation	Yes	A valid class name	n/a

sequence	Name of the sequence message that should be injected	Yes	A valid sequence name	n/a
onError	Name of the fault sequence that should be invoked in case of failure	Yes	A valid fault sequence name	n/a
inbound.behavior	The behaviour of the inbound endpoint	Yes	event-based	n/a

You can download the maven artifact used in the sample custom event-based inbound endpoint configuration above from [Custom event-based inbound endpoint sample](#).

You need to copy the built jar file to the `<ESB_HOME>/repository/components/lib` directory and restart the ESB to load the class.

## Working with Inbound Endpoints via WSO2 ESB Tooling

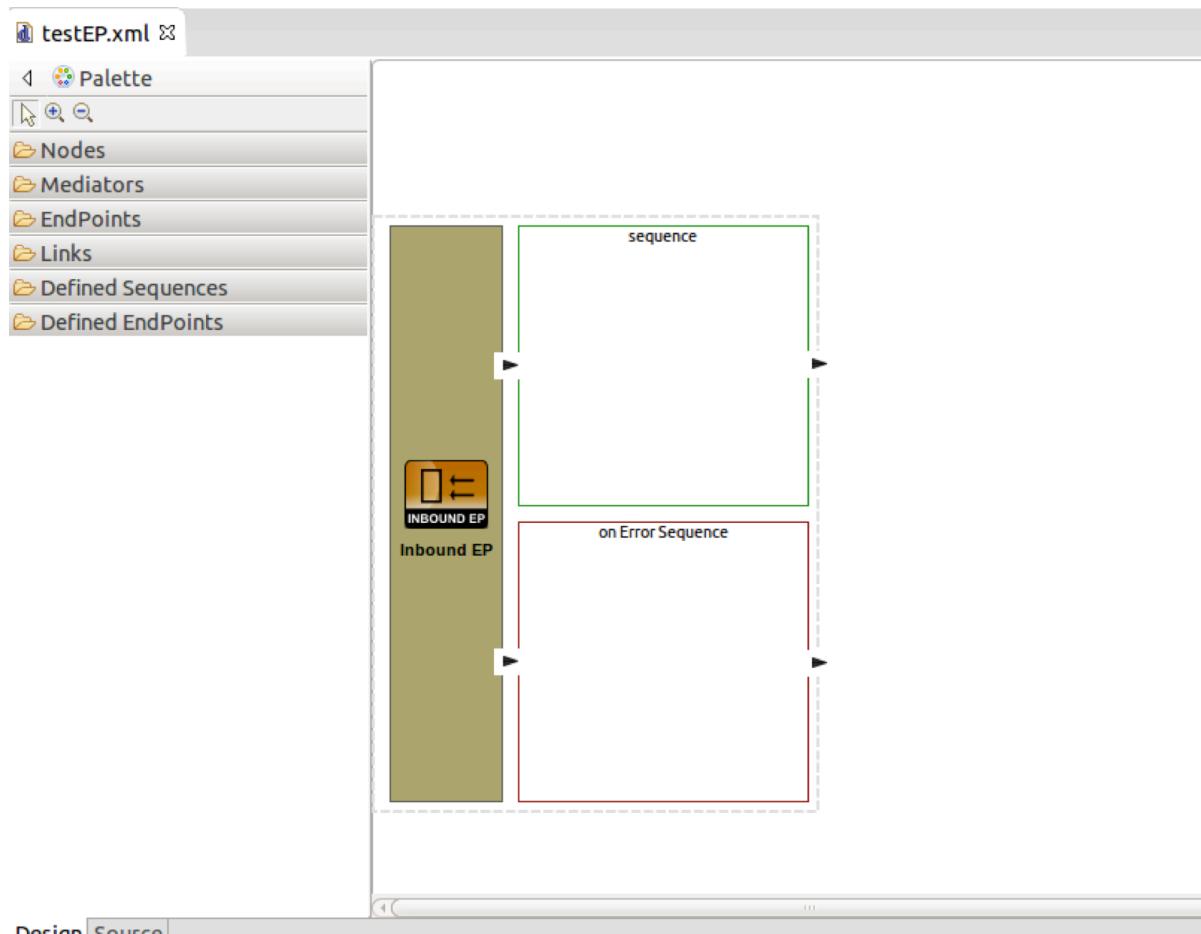
You can create a new inbound endpoint or import an existing inbound endpoint from an XML file, such as a Synapse configuration file using WSO2 ESB tooling.

You need to have WSO2 ESB tooling installed to create a new inbound endpoint or to import an existing inbound endpoint via ESB tooling. For instructions on installing WSO2 ESB tooling, see [Installing WSO2 ESB Tooling](#).

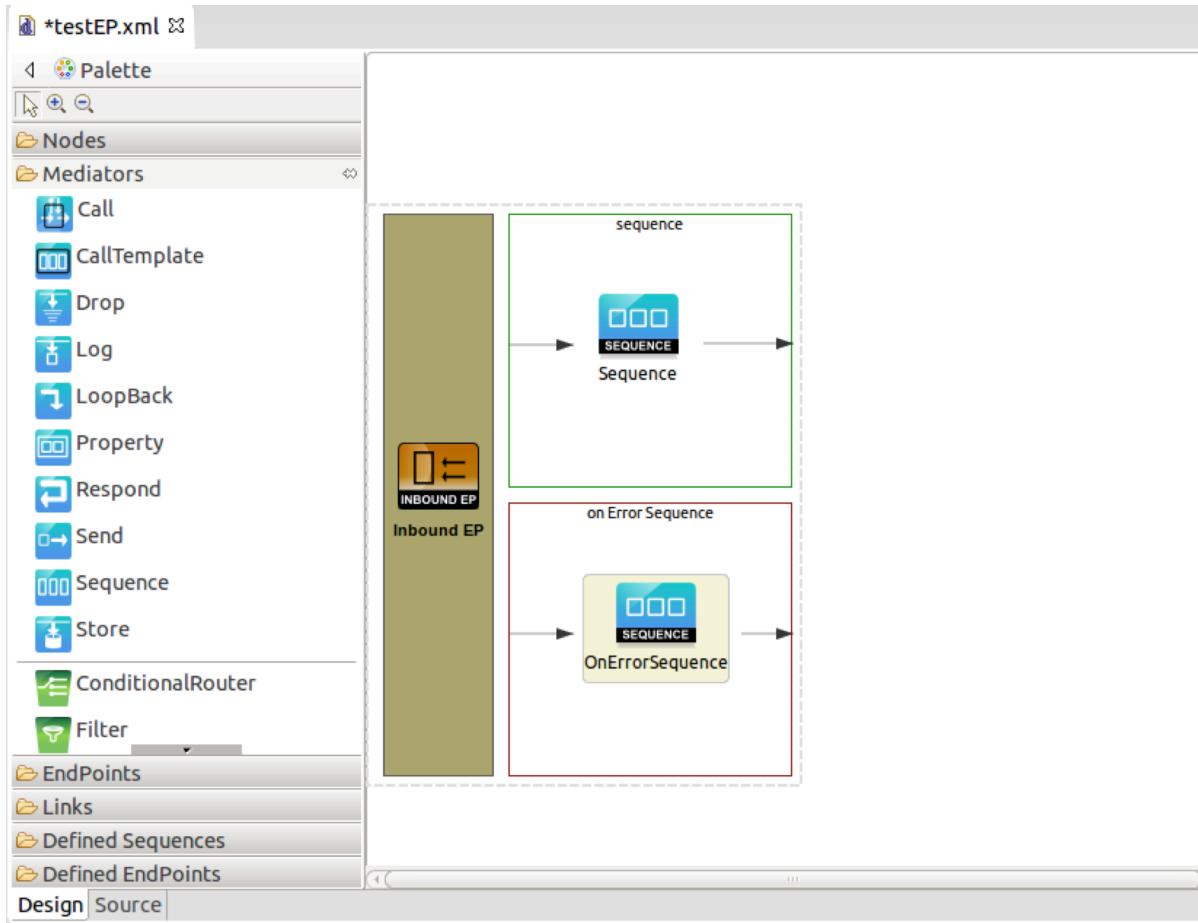
### ***Creating a new inbound endpoint***

Follow the steps below to create a new inbound endpoint.

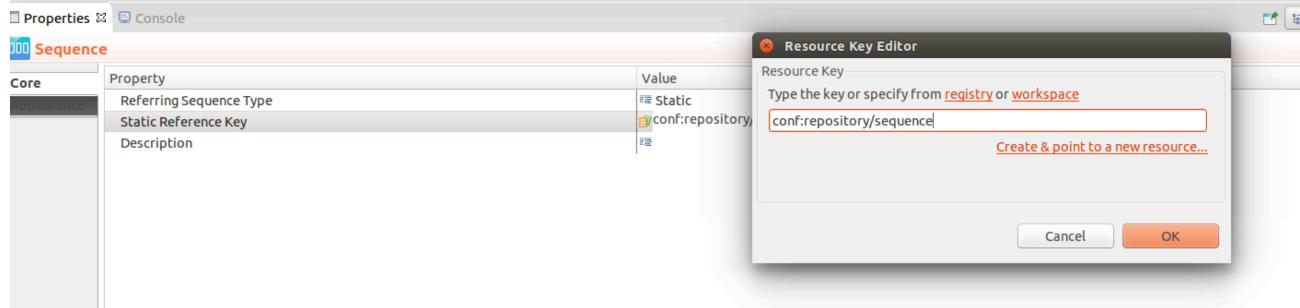
1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Inbound Endpoint** on the **Developer Studio Dashboard**. Alternatively, you can right-click on the ESB Config project and then click **New > Inbound Endpoint**.
3. Select **Create a New Inbound Endpoint** and click **Next**.
4. Give a name for the inbound endpoint, select the protocol type and click **Finish**.
5. The created inbound endpoint opens in the ESB Graphical Editor as shown below:



6. Drag and drop a sequence and an `onError` sequence from the tool palette as shown below:



7. If you want to browse a sequence from the registry, right-click on the sequence and click **Show Properties View**.
8. Point to an existing sequence in the registry in the **Static Reference Key** property.



9. Right-click on the inbound endpoint and click **Show Properties View** to open the **Property** window.
10. Update the parameters as preferred and the changes will be saved in the inbound endpoint configuration.

### **Importing an existing inbound endpoint**

Follow the steps below to import an existing inbound endpoint from an XML file such as a Synapse configuration file, into an ESB Config project.

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Inbound Endpoint** on the **Developer Studio Dashboard**. Alternatively, you can right-click on the ESB Config project and then click **New > Inbound Endpoint**.
3. Select **Import Inbound Endpoint** and click **Next**.
4. Specify the inbound endpoint file by typing its full path name or clicking **Browse** and navigating to the file.
5. In the **Save Inbound Endpoint In** field, specify an existing ESB Config project in your workspace where you

want to save the inbound endpoint, or click **Create new ESB Project** to create a new ESB Config project and save the endpoint there.

6. Click **Finish**. The inbound endpoint you selected is created in the `inbound-endpoints` folder under the ESB Config project you specified, and the inbound endpoint opens in the editor.

## Working with Inbound Endpoints via the Management Console

You can add, edit and delete inbound endpoints via the ESB Management Console. You can also enable/disable statistics as well as enable/disable tracing for inbound endpoints via the Management console.

See the following topics for information on how to manage inbound endpoints via the Management Console:

- Adding an inbound endpoint
- Editing an inbound endpoint
- Deleting an inbound endpoint
- Enabling/disabling statistics
- Enabling/disabling tracing

### Adding an inbound endpoint

Follow the steps given below to add an inbound endpoint.

1. Click the **Main** tab on the Management Console and then go to **Manage -> Service Bus** and click **Inbound Endpoints** to open the **Inbound Endpoints** page.
2. On the **Inbound Endpoints** page, click **Add Inbound Endpoint** to open the **New Inbound Endpoint** page.
3. Specify a name for the inbound endpoint, select the inbound protocol type for the new inbound endpoint , and click **Next**. This displays all the parameters applicable to the new inbound endpoint depending on the protocol type you selected.
4. Specify values for all the required parameters. Following are the available inbound protocol types. Click on a required protocol type for information on the parameters you need to specify as well as to view a sample configuration for an inbound endpoint of the relevant protocol type.
  - [HTTP Inbound Protocol](#)
  - [HTTPS Inbound Protocol](#)
  - [File Inbound Protocol](#)
  - [JMS Inbound Protocol](#)
  - [HL7 Inbound Protocol](#)
  - [Kafka Inbound Protocol](#)
  - [MQTT Inbound Protocol](#)
  - [RabbitMQ Inbound Protocol](#)
  - [Custom Inbound Protocol](#)
5. Click **Show Advanced Options** if you want to configure advanced settings for the inbound endpoint. The parameters displayed under advanced options also depend on the protocol type you selected for the inbound endpoint.
6. Click **Save**. You will see the new inbound endpoint you added displayed under **Available Defined Inbound Endpoints** on the **Inbound Endpoints** page.

### Editing an inbound endpoint

Follow the steps given below to edit an inbound endpoint.

1. Click the **Main** tab on the Management Console and then go to **Manage -> Service Bus** and click **Inbound Endpoints** to open the **Inbound Endpoints** page.
2. On the **Inbound Endpoints** page, you will see all the inbound endpoints that you have added under **Available Defined Inbound Endpoints**.
3. Click **Edit** to make necessary changes to a required inbound endpoint.
4. Change the existing parameter values based on your requirement and click **Update**. The changes will be applied to the inbound endpoint.

### Deleting an inbound endpoint

Follow the steps below to delete an inbound endpoint.

1. Click the **Main** tab on the Management Console and then go to **Manage -> Service Bus** and click **Inbound Endpoints** to open the **Inbound Endpoints** page.
2. On the **Inbound Endpoints** page, you will see all the inbound endpoints that you have added under **Available Defined Inbound Endpoints**.
3. Click **Delete** to discard an inbound endpoint that you no longer need. A message appears to confirm that you want to proceed with deleting the inbound endpoint.
4. Click **Yes**. The inbound endpoint will be deleted and will no longer appear under **Available Defined Inbound Endpoints** on the **Inbound Endpoints** page.

### **Enabling/disabling statistics**

Follow the steps below to enable or disable statistics for inbound endpoints.

1. Click the **Main** tab on the Management Console and then go to **Manage -> Service Bus** and click **Inbound Endpoints** to open the **Inbound Endpoints** page.
2. On the **Inbound Endpoints** page, you will see all the inbound endpoints that you have added under **Available Defined Inbound Endpoints**. You will see the **Enable Statistics** link displayed for inbound endpoints that have statistics currently disabled, and the **Disable Statistics** link displayed for inbound endpoints that have statistics currently enabled.
3. If you want to enable statistics for an inbound endpoint, click **Enable Statistics** for the particular inbound endpoint.
4. If you want to disable statistics for an inbound endpoint, click **Disable Statistics** for the particular inbound endpoint.

### **Enabling/disabling tracing**

Follow the steps below to enable or disable tracing for inbound endpoints.

1. Click the **Main** tab on the Management Console and then go to **Manage -> Service Bus** and click **Inbound Endpoints** to open the **Inbound Endpoints** page.
2. On the **Inbound Endpoints** page, you will see all the inbound endpoints that you have added under **Available Defined Inbound Endpoints**. You will see the **Enable Tracing** link displayed for inbound endpoints that have tracing disabled, and the **Disable Tracing** link displayed for inbound endpoints that have tracing enabled.
3. If you want to enable tracing for an inbound endpoint, click **Enable Tracing** for the particular inbound endpoint.
4. If you want to disable tracing for an inbound endpoint, click **Disable Tracing** for the particular inbound endpoint.

## **Working with Scheduled Tasks**

A scheduled task in WSO2 ESB allows you to run a piece of code triggered by a timer. WSO2 ESB provides a default scheduled task implementation, which you can use to inject a message to the ESB at a scheduled interval. To activate the built-in task, you have to add it to the *Tasks* list.

For information on working with scheduled tasks via the ESB tooling plug-in, see [Working with Scheduled Tasks via the ESB Tooling Plug-In](#).

For information on working with scheduled tasks via the Management Console, see [Working with Scheduled Tasks via the Management Console](#).

You can also write your own scheduled task by creating a custom Java class that implements the `org.apache.synapse.startup.Task` interface. This interface defines a single method named `execute()`. For example, you could create a task that will read a text file at a specified location and place orders for stocks that are given in the text file. For more information, see [Writing Tasks](#).

## **Working with Scheduled Tasks via WSO2 ESB Tooling**

You can create a new scheduled task or to import an existing scheduled task from the file system using WSO2 ESB tooling.

You need to have WSO2 ESB tooling installed to create a new task or to import an existing task via ESB tooling. For instructions on installing WSO2 ESB tooling, see [Installing WSO2 ESB Tooling](#).

### ***Creating a new scheduled task***

Follow these steps to create a new scheduled task. Alternatively, you can [import an existing scheduled task](#).

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Scheduled Task** on the **Developer Studio Dashboard**.
3. Leave the first option selected and click **Next**.
4. Type a unique name for this scheduled task and specify the group, implementation class, and other options. For more information, see [Adding and Scheduling Tasks](#).
5. Do one of the following:
  - To save the task in an existing ESB Config project in your workspace, click **Browse** and select that project.
  - To save the task in a new ESB Config project, click **Create new ESB Project** and create the new project.
6. Click **Finish**. The scheduled task is created in the `src/main/synapse-config/tasks` folder under the ESB Config project you specified. When prompted, you can open the file in the editor, or you can right-click the task in the project explorer and click **Open With > ESB Editor**. Click its icon in the editor to view its properties.

### ***Importing a scheduled task***

Follow these steps to import an existing scheduled task into an ESB Config project. Alternatively, you can [create a new scheduled task](#).

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Scheduled Task** on the **Developer Studio Dashboard**.
3. Select **Import Scheduled Task Artifact** and click **Next**.
4. Specify the XML file that defines the scheduled task by typing its full pathname or clicking **Browse** and navigating to the file.
5. In the **Save Task In** field, specify an existing ESB Config project in your workspace where you want to save the task, or click **Create new ESB Project** to create a new ESB Config project and save the task configuration there.
6. If there are multiple tasks definitions in the file, click **Create ESB Artifacts**, and then select the tasks you want to import.
7. Click **Finish**. The tasks you selected are created in the subfolders of the `src/main/synapse-config/tasks` folder under the ESB Config project you specified, and the first task appears in the editor.

## **Working with Scheduled Tasks via the Management Console**

You can use the management console to add, edit and delete scheduled tasks.

You can also schedule tasks to execute periodically. After deploying a task implementation to the ESB runtime, you can use the ESB management console to schedule various instances of the task by specifying the number of times the task needs to run along with the frequency, or you can use the cron syntax.

See the following topics for information on how to work with scheduled tasks via the Management Console:

- [Adding and Scheduling Tasks](#)
- [Editing Scheduled Tasks](#)
- [Deleting a Scheduled Task](#)

## Adding and Scheduling Tasks

A **task** runs a piece of code triggered by a timer, allowing you to run scheduled jobs at specified intervals. A task can be scheduled in the following ways:

1. Using `count` and `interval` attributes to run the task a `specified number of times` at a given `interval`.
2. Giving the scheduled time as a `cron style` entry.
3. Making the task run only `once` after the ESB starts by using the `once` attribute.

Having deployed a task implementation to the ESB runtime (see [Writing Tasks](#)), you can use the ESB Management Console to add a task to the "Tasks" list and schedule various instances of the task. You can use either [UI configuration](#) or [XML configuration](#) to add and schedule tasks.

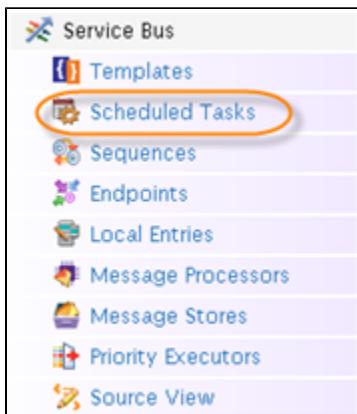
### UI Configuration

Follow the instructions below to add and schedule a task in ESB Management Console.

1. Sign in. Enter your user name and password to log in to the ESB Management Console.
2. Click **Main** in the left menu to access the Manage menu.



3. In the Manage menu, click **Scheduled Tasks** under Service Bus.



4. The Scheduled Tasks page appears, where you can add, [edit](#), and [delete](#) tasks.

**Scheduled Tasks**

Available Defined Scheduled Tasks

Task Name	Action
Task_1	Edit  Delete

Add Task

5. Click **Add Task**.

**Scheduled Tasks**

Available Defined Scheduled Tasks

Task Name	Action
Task_1	Edit  Delete

Add Task

6. The New Scheduled Task page appears. Enter the required details into the fields.

- **Task Name** - Name of a scheduled task.
- **Task Group** - The group name to grouping tasks. The group name `synapse.simple.quartz` belongs to ESB - Synapse. All available groups are displayed as a drop-down menu. If there are tasks belong to some other domains, for example WSO2 Mashups tasks, then those will be shown here as a separate group names.
- **Task Implementation** - The implementation class of the task. To use the default task implementation that is available with the ESB (and therefore can be used without downloading any third-party libraries or custom JARs), specify `org.apache.synapse.startup.tasks.MessageInjector`. This class simply injects a specified message into the Synapse environment at ESB startup. For more information on writing custom task implementations, see [Writing Tasks](#).
- **Trigger Type** - Trigger type for the task. This can be selected as either "Simple" or "Cron."
  - **Simple Trigger** - Defined by specifying a count and an interval, implying that the task will run a count number of times at specified intervals.
    - **Count** - The number of times the task will be executed.
    - **Interval** - The interval between consecutive executions of a task.
  - **Cron Trigger** - Defined using a cron expression.
- **Pinned Servers** - Provides a list of ESB server names, where this task should be started for the "Pinned Servers" value.

## New Scheduled Task

New Scheduled Task

Task Name\*

Task Group\*

Task Implementation\*  [Load Task Properties](#)

Trigger Information of the Task

Trigger Type  Simple  Cron

Count

Interval (in seconds)\*

Miscellaneous Information

Pinned Servers   
(separated by comma or space)

[Schedule](#) [Cancel](#)

7. Click **Load Task Properties** to see the instance properties of the task implementation.

New Scheduled Task

Task Name\*

Task Group\*

Task Implementation\*  [Load Task Properties](#)

8. Use the instance properties fields as follows:

- **Property Name** - The unique name of the task property.
- **Property Type** - The type of property, either Literal or XML.
- **Property Value** - The value of the property.
- **Action** - Allows you to delete a property.

For more information on setting the properties for the default task implementation, see [Examples](#) and [Injecting the message to a named sequence or proxy service](#) below.

Property Name	Property Type	Property Value	Action
format	Literal <input type="button" value="▼"/>	<input type="text"/>	Delete
message	Literal <input type="button" value="▼"/>	<input type="text"/>	Delete
soapAction	Literal <input type="button" value="▼"/>	<input type="text"/>	Delete
to	Literal <input type="button" value="▼"/>	<input type="text"/>	Delete
proxyName	Literal <input type="button" value="▼"/>	<input type="text"/>	Delete
sequenceName	Literal <input type="button" value="▼"/>	<input type="text"/>	Delete
injectTo	Literal <input type="button" value="▼"/>	<input type="text"/>	Delete

The org.apache.synapse.startup.tasks.MessageInjector implementation takes the following properties:

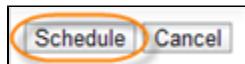
- **format** - defines the format of the message similar to Address Endpoint formats: soap11, soap12, pox, get
- **message** - you can provide an XML or literal value depending on message format.

**Note**

When you add a scheduled task, it is mandatory to provide a value for the message property. Therefore, even if you do not want to send a message body, you have to provide an empty payload as the value to avoid an exception being thrown.

- **soapAction** - specify the SOAP Action to use when sending the message to the endpoint.
- **to** - specify the endpoint address.
- **injectTo** - specify whether to inject a message to a proxy service or sequence. This field takes values 'sequence' or 'proxy' and 'main' to inject to main sequence.
- **proxyName** - if injectTo contains 'proxy' then the name of the proxy to inject the message to is specified here.
- **sequenceName** - if injectTo contains 'sequence' then the name of the sequence to inject the message to is specified here.

9. Click **Schedule** to apply the settings.



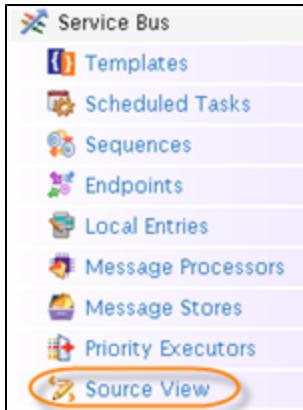
### **XML Configuration**

Follow the instructions below to add and schedule tasks using XML configuration.

1. Sign in. Enter your user name and password to log on to the ESB Management Console.
2. Click on "Main" in the left menu to access the "Manage" menu.



3. In the "Manage" menu, click on "Source View" under "Service Bus."



4. In the source view, add the task configuration based on your requirement.

**Service Bus Configuration**

Make the required modifications to the configuration and click 'Update' to apply the changes to the server. Use 'Reset' button to undo your changes.

Current Configuration

```

48 </out>
49 </sequence>
50 <task name="Task_1" class="org.apache.synapse.startup.tasks.MessageInjector" group="synapse.simple.quartz"
51 <trigger interval="5"/>...
52 </task>
53 <priority-executor name="Executor_1">
54 <queues>
55 <queue size="10" priority="5"/>
56 <queue size="15" priority="2"/>
57 </queues>
58 <threads max="100" core="20" keep-alive="5"/>
59 </priority-executor>
60 </definitions>

```

The syntax of the task configuration is as follows:

```

<task class="string" name="string" [group="string"] [pinnedServers="(serverName)+"]>
 <property name="string" value="String"/>
 <property name="string"><somexml>config</somexml></property>
 <trigger ([count="int"]? interval="int"] | [cron="string"] | [once=(true |
false)])/>
</task>

```

## Examples

Following are examples of configuring some common use cases. For an example of configuring a task with a simple trigger, see [Sample 300: Introduction to Tasks with a Simple Trigger](#). To see a complete example of writing a new task and configuring it in the UI, see [Writing Tasks Sample](#).

To run every 5 seconds continuously:

```

<task name="CheckPrice" class="org.wso2.esb.tutorial.tasks.PlaceStockOrderTask">
<trigger interval="5" />
</task>

```

To run every 5 seconds for 10 times:

```
<task name="CheckPrice" class="org.wso2.esb.tutorial.tasks.PlaceStockOrderTask">
<trigger interval="5" count="10"/>
</task>
```

You can also give cron-style values. To run daily at 1:30 AM:

```
<task name="CheckPrice" class="org.wso2.esb.tutorial.tasks.PlaceStockOrderTask">
<trigger cron="0 30 1 * * ?"/>
</task>
```

To run only once after ESB starts:

```
<task name="CheckPrice" class="org.wso2.esb.tutorial.tasks.PlaceStockOrderTask">
<trigger once="true"/>
</task>
```

### ***Injecting the message to a named sequence or proxy service***

By default, the message is sent to the Main sequence. To send it to a different sequence or to a proxy service, set the `injectTo` property to `sequence` or `proxy`, and then add the `sequenceName` or `proxyName` property to specify the name of the sequence or proxy service to use. For example:

## Injecting to a sequence other than Main

```
<task name="SampleInjectToSequenceTask"
 class="org.apache.synapse.startup.tasks.MessageInjector"
 group="synapse.simple.quartz">
 <trigger count="2" interval="5"/>

 <property xmlns:task="http://www.wso2.org/products/wso2commons/tasks"
 name="injectTo"
 value="sequence"/>

 <property xmlns:task="http://www.wso2.org/products/wso2commons/tasks"
 name="message">
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>IBM</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </property>

 <property xmlns:task="http://www.wso2.org/products/wso2commons/tasks"
 name="sequenceName"
 value="SampleSequence"/>
 </task>
```

## Injecting to a proxy service

```
<task name="SampleInjectToProxyTask"
 class="org.apache.synapse.startup.tasks.MessageInjector"
 group="synapse.simple.quartz">
 <trigger count="2" interval="5"/>
 <property xmlns:task="http://www.wso2.org/products/wso2commons/tasks"
 name="message">
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>IBM</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </property>

 <property xmlns:task="http://www.wso2.org/products/wso2commons/tasks"
 name="proxyName"
 value="SampleProxy" />

 <property xmlns:task="http://www.wso2.org/products/wso2commons/tasks"
 name="injectTo"
 value="proxy" />
 </task>
```

## Injecting messages to RESTful Endpoints

In order to use the Message Injector to inject a message to a RESTful endpoint, we can specify the injector with the required payload and inject the message to sequence or proxy service as defined above. The sample below shows a RESTful message injection through a ProxyService.

## Injecting to a sequence other than Main

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <registry provider="org.wso2.carbon.mediation.registry.WSO2Registry">
 <parameter name="cachableDuration">15000</parameter>
 </registry>
 <proxy name="SampleProxy"
 transports="https http"
 startOnLoad="true"
 trace="disable">
 <description/>
 <target>
 <inSequence>
 <property name="uri.var.city" expression="//request/location/city"/>
 <property name="uri.var.cc" expression="//request/location/country"/>
 <log>
 <property name="Which city?" expression="get-property('uri.var.city')"/>
 <property name="Which country?" expression="get-property('uri.var.cc')"/>
 </log>
 <send>
 <endpoint name="EP">
 <http method="get"
 uri-template="http://api.openweathermap.org/data/2.5/weather?q={uri.var.city},{uri.var.cc}"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <log level="full"/>
 <drop/>
 </outSequence>
 </target>
 </proxy>

 <task name="SampleInjectToProxyTask"
 class="org.apache.synapse.startup.tasks.MessageInjector"
 group="synapse.simple.quartz">
 <trigger count="2" interval="5"/>
 <property xmlns:task="http://www.wso2.org/products/wso2commons/tasks"
 name="injectTo"
 value="proxy"/>
 <property xmlns:task="http://www.wso2.org/products/wso2commons/tasks"
 name="message">
 <request xmlns="">
 <location>
 <city>London</city>
 <country>UK</country>
 </location>
 </request>
 </property>
 <property xmlns:task="http://www.wso2.org/products/wso2commons/tasks"
 name="proxyName"
 value="SampleProxy"/>
 </task>
</definitions>

```

## Editing Scheduled Tasks

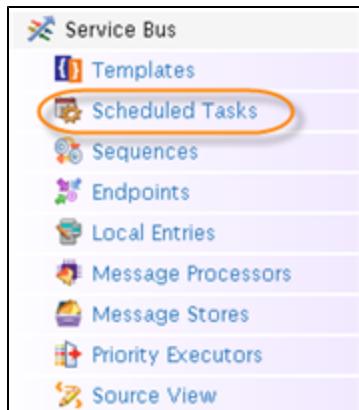
Having added a scheduled task to the WSO2 ESB, you can edit their schedule if necessary.

Follow the instructions below to edit a scheduled task via the ESB Management Console.

1. Sign in. Enter your user name and password to log on to the ESB Management Console.
2. Click on "Main" in the left menu to access the "Manage" menu.



3. In the "Manage" menu, click on "Scheduled Tasks" under "Service Bus."



4. The "Scheduled Tasks" page appears.

Task Name	Action
Task_1	Edit  Delete

**Add Task**

From here you can **add**, **edit** and **delete** tasks.

5. Click on the "Edit" link associated with a certain task.

**Scheduled Tasks**

Available Defined Scheduled Tasks

Task Name	Action
Task_1	Edit  Delete

Add Task

6. The "Edit Task" page appears. Here you can alter the options of a task except "Task Name" and "Task Group."

**Edit Task : Task\_1**

**Edit Task**

Task Name*	Task_1
Task Group*	synapse.simple.quartz
Task Implementation*	org.apache.synapse.startup.tasks.MessageIn

**Trigger Information of the Task**

Trigger Type	<input checked="" type="radio"/> Simple <input type="radio"/> Cron
Count	-1
Interval (in seconds)*	1500

**Miscellaneous Information**

Pinned Servers (separated by comma or space)	
----------------------------------------------	--

**Schedule Cancel**

To learn more information about tasks options, see [Adding and Scheduling Tasks](#).

7. Once setting up the options, click on the "Schedule" button to save alterations.

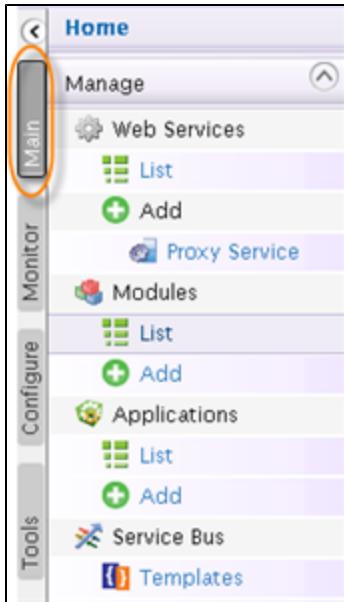


#### Deleting a Scheduled Task

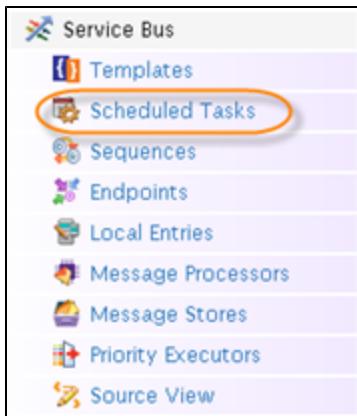
If there is no need to execute a particular task code any more, you can delete this task from the "Tasks" list in the ESB. A task will disappear from the list, but your [custom task class](#) will stay in the directory it was put in the ESB.

Follow the instructions below to delete a scheduled task from the "Tasks" list.

1. Sign in. Enter your user name and password to log on to the ESB Management Console.
2. Click on "Main" in the left menu to access the "Manage" menu.



3. In the "Manage" menu, click on "Scheduled Tasks" under "Service Bus."



4. The "Scheduled Tasks" page appears.

The screenshot shows the 'Scheduled Tasks' page. At the top, it says 'Available Defined Scheduled Tasks'. Below that is a table:

Task Name	Action
Task_1	Edit  Delete

At the bottom left of the table area is a green plus sign icon with the text 'Add Task'.

Here you can [add](#), [edit](#) and [delete](#) tasks.

5. Select a necessary task and click on the "Delete" link associated with the task.

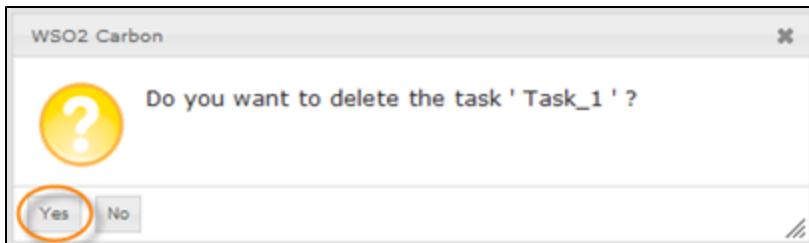
Scheduled Tasks

Available Defined Scheduled Tasks

Task Name	Action
Task_1	Edit  Delete

Add Task

6. Confirm your request in the "WSO2 Carbon" window. Click "Yes."



7. The task will disappear from the list and will not be executed till you add it and schedule again.

## Using REST

You can send and receive RESTful messages through the ESB using the HTTP and HTTPS transports (other transports such as the Local transport are not supported). The following topics describe the various scenarios for using REST.

- [Using REST with a Proxy Service](#)
- [Using REST with APIs](#)

You can also use the [HTTP endpoint](#) to define REST endpoints using [URI templates](#) similar to the REST API.

The following samples demonstrate using REST with the ESB:

- [Sample 50: POX to SOAP conversion](#)
- [Sample 152: Switching Transports and Message Format from SOAP to REST POX](#)
- [Sample 800: Introduction to REST API](#)

## Mediating Messages

A mediator is the basic message processing unit in the ESB. A mediator can take a message, carry out some predefined actions on it, and output the modified message. For example, the Clone mediator splits a message into several clones, the Send mediator sends the messages, and the Aggregate mediator collects and merges the responses before sending them back to the client. Message mediation is a fundamental part of any ESB.

WSO2 ESB ships with a range of mediators capable of carrying out various tasks on input messages, including functionality to match incompatible protocols, data formats and interaction patterns across different resources. Data can be split, cloned, aggregated, and enriched, allowing the ESB to match the different capabilities of services. XQuery and XSLT allow rich transformations on the messages. Rule-based message mediation allows users to cope with the uncertainty of business logic. Content-based routing using XPath filtering is supported in different flavors, allowing users to get the most convenient configuration experience. Built-in capability to handle Transactions allows message mediation to be done transactionally inside the ESB. With the [eventing](#) capabilities of ESB, EDA based components can be easily interconnected, allowing the ESB to be used in the front-end of an organisation's SOA infrastructure.

## Mediators

A **mediator** is a full-powered processing unit in the ESB. In run-time it has access to all parts of the ESB along with the current message. Usually, a mediator is configured using XML. Different mediators have their own XML configurations.

At the run-time, a message is injected into the mediator with the ESB run-time information. Then this mediator can do virtually anything with the message. A user can write a mediator and put it into the ESB. This custom mediator and any other built-in mediator will be exactly the same as the API and the privileges (Refer to more information in [Writing a WSO2 ESB Mediator](#)).

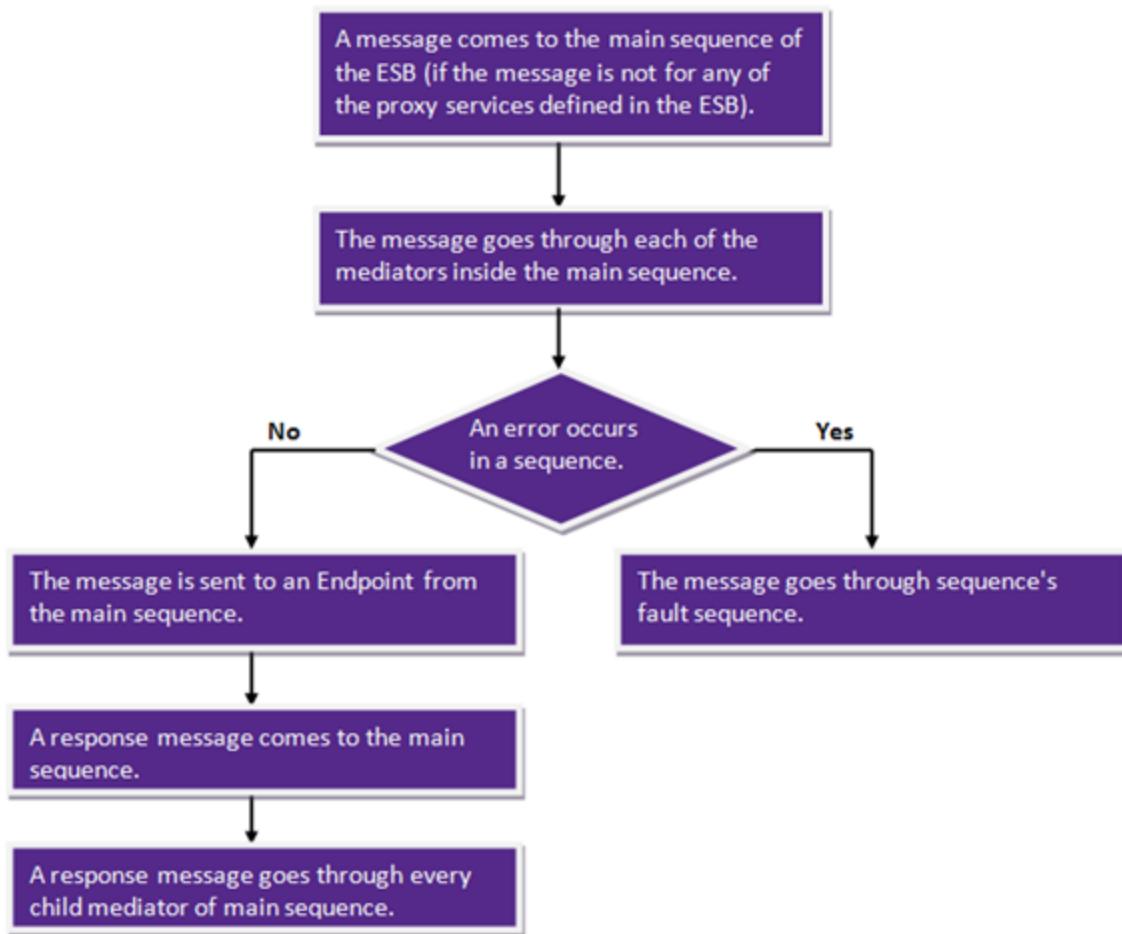
Refer to [Mediators](#).

## Mediation Sequence

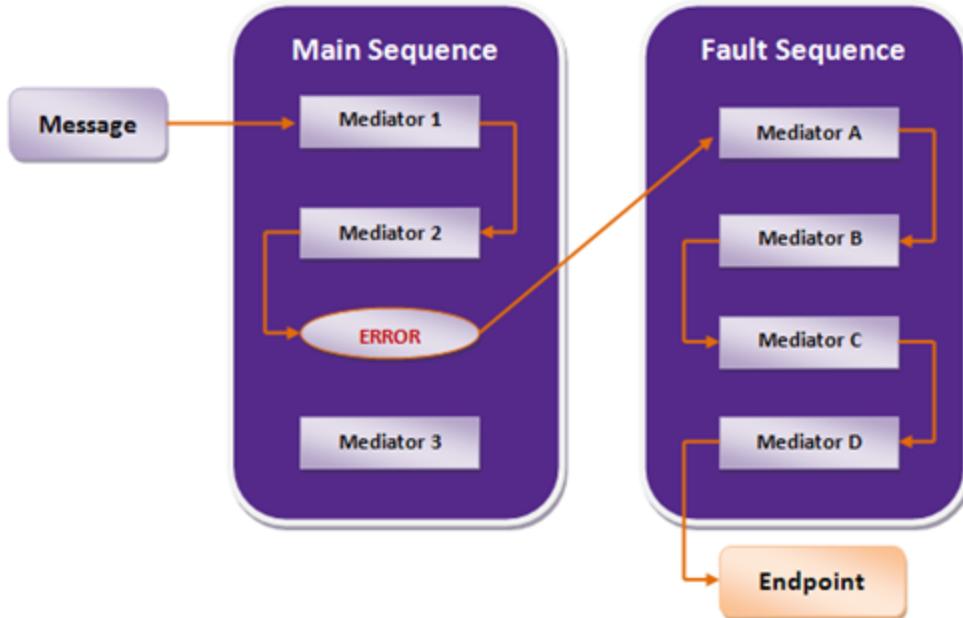
A mediation sequence, commonly called a "sequence", is a list of mediators. That means, it can hold other mediators and execute them. It is part of the ESB's core and message mediation cannot live without this mediator. When a message is delivered to a sequence, it sends the message through all its child mediators.

Read more in [Mediation Sequences](#).

### The Process of Message Mediation



In case an error occurs in the main sequence while processing, the message goes to the fault sequence.



For detailed information about mediators, refer to the following pages:

- [Mediators](#)
- [Working with Mediators via WSO2 ESB Tooling](#)
- [Working with Mediators via the Management Console](#)
- [Working with Local Registry Entries](#)
- [Creating Custom Mediators](#)
- [Best Practices for Mediation](#)
- [Mediation Sequences](#)
- [Working with Message Stores and Message Processors](#)
- [Working with Message Builders and Formatters](#)
- [Prioritizing Messages](#)
- [Transactions](#)
- [Debugging Mediation](#)

## Mediators

A **mediator** is a full-powered processing unit in the ESB. At run-time, a mediator has access to all the parts of the ESB along with the current message and can do virtually anything with the message.

When [adding a mediator to a sequence](#), you can configure the mediator in design view or in source view. Source view provides the XML representation of the configurations done in the UI. When you edit the source XML all changes immediately reflect in the design view as well.

Each mediator has its own XML configuration. In the source view of the following mediators, you can comment out required lines if necessary when you edit the XML.

- [Aggregate Mediator](#)
- [Cache Mediator](#)
- [Filter Mediator](#)
- [In Mediator](#)
- [Iterate Mediator](#)
- [Out Mediator](#)
- [Sequence Mediator](#)
- [Synapse Mediator](#)
- [Template Mediator](#)
- [Validate Mediator](#)

It is also possible to comment out required lines in the Synapse definition as well as in the source view of the complete ESB configuration.

Mediators in a sequence can be one of the following types:

- **Node mediators** - Contains child mediators.
- **Leaf mediators** - Does not hold any other child mediators.

The message content is accessed by some mediators in some mediation scenarios while it is not accessed in the other scenarios. Mediators can be classified as follows based on this aspect.

- **Content-aware mediators**: These mediators always access the message content when mediating messages (e.g., [Enrich mediator](#)).
- **Content-unaware mediators**: These mediators never access the message content when mediating messages (e.g., [Send mediator](#)).
- **Conditionally content-aware mediators**: These mediators could be either content-aware or content-unaware depending on their exact instance configuration. For an example a simple [Log Mediator](#) instance (i.e. configured as `<log/>`) is content-unaware. However a log mediator configured as `<log level="full"/>` would be content-aware since it is expected to log the message payload.

Mediators are considered to be one of the main mechanisms for [extending an ESB](#). You can [write a custom mediator](#) and add it to the ESB. This custom mediator and any other built-in mediator will be exactly the same as the API and the privileges.

The standard mediators in WSO2 ESB are listed in the table below. Click a link for details on that mediator. There are also many [samples](#) that demonstrate how to use mediators.

#### *The WSO2 ESB mediator catalog*

Category	Name	Description
Core	Call	Invoke a service in non blocking synchronous manner
	Enqueue	Uses a <a href="#">priority executor</a> to ensure high-priority messages are not dropped
	Send	Sends a message
	Loopback	Moves the message from the In flow to the Out flow, skipping all remaining configuration in the In flow
	Sequence	Inserts a reference to a sequence
	Respond	Stops processing on the message and sends it back to the client
	Event	Sends event notifications to an event source, publishes messages to predefined topics
	Drop	Drops a message
	Call Template	Constructs a sequence by passing values into a <a href="#">sequence template</a>
	Enrich	Enriches a message
	Property	Sets or remove properties associated with the message
	Log	Logs a message

Filter	Filter	Filters a message using XPath, if-else kind of logic
	Out	Applies to messages that are in the Out path of the ESB
	In	Applies to messages that are in the In path of the ESB
	Validate	Validates XML messages against a specified schema.
	Switch	Filters messages using XPath, switch logic
	Router	Routes messages based on XPath filtering
	Conditional Router	Implements complex routing rules (Header based routing, content based routing and other rules)
Transform	XSLT	Performs XSLT transformations on the XML payload
	FastXSLT	Performs XSLT transformations on the message stream
	URLRewrite	Modifies and rewrites URLs or URL fragments
	XQuery	Performs XQuery transformation
	Header	Sets or removes SOAP headers
	Fault (also called Makefault)	Create SOAP Faults
	PayloadFactory	Transforms or replaces message content in between the client and the backend server
Advanced	Cache	Evaluates messages based on whether the same message came to the ESB
	ForEach	Splits a message into a number of different messages by finding matching elements in an XPath expression of the original message.
	Clone	Clones a message
	Store	Stores messages in a predefined message store
	Iterate	Splits a message
	Aggregate	Combines a message
	Callout	Blocks web services calls
	Transaction	Executes a set of mediators transactionally
	Throttle	Limits the number of messages
	DBReport	Writes data to database
	DBLookup	Retrieves information from database
	EJB	Calls an external Enterprise JavaBean(EJB) and stores the result in the message payload or in a message context property.
	Rule	Executes rules

	Entitlement	Evaluates user actions against a XACML policy
	OAuth	2-legged OAuth support
	Smooks	Used to apply lightweight transformations on messages in an efficient manner.
Extension	Bean	Manipulates JavaBeans
	Class	Creates and executes a custom mediator
	POJOCommand	Executes a custom command
	Script	Executes a mediator written in Scripting language
	Spring	Creates a mediator managed by Spring
Agent	BAM	Captures data events and sends them to the BAM server

## Aggregate Mediator

The **Aggregate mediator** implements the [Aggregator enterprise integration pattern](#) and aggregates the **response messages** for messages that were split by the [Clone](#) or [Iterate](#) mediator and sent using the [Send](#) mediator.

Note that the responses will not necessarily be aggregated in the same order that the requests were sent, even if you set the `sequential` attribute to `true` on the [Iterate](#) mediator.

The Aggregate mediator is a [content-aware](#) mediator.

---

## Syntax | UI Configuration | Examples

---

### Syntax

```
<aggregate>
 <correlateOn expression="xpath" />?
 <completeCondition [timeout="time-in-seconds"]>
 <messageCount min="int-min" max="int-max" />?
 </completeCondition>?
 <onComplete expression="xpath" [sequence="sequence-ref"]>
 (mediator +)?
 </onComplete>
</aggregate>
```

---

### UI Configuration

Mediator  switch to source view

## Aggregate Mediator

Aggregate ID	<input type="text"/>	
Aggregation Expression *	<input type="text"/>	 NameSpaces
Completion Timeout(seconds)	<input type="text"/>	
Completion Max-messages	<input type="button" value="Value"/> <input type="text" value="-1"/>	
Completion Min-messages	<input type="button" value="Value"/> <input type="text" value="-1"/>	
Correlation Expression	<input type="text"/>	 NameSpaces
Enclosing Element Property	<input type="text"/>	
<input checked="" type="radio"/> Anonymous <input type="radio"/> Pick From Registry		
On Complete	<input type="radio"/> Anonymous <input type="radio"/> Pick From Registry	
<input type="button" value="Update"/>		

The parameters available for configuring the Aggregate mediator are as follows.

Parameter Name	Description
Aggregate ID	This optional attribute can be used to aggregate only responses for split messages that are created by a specific clone/iterate mediator. Aggregate ID should be the same as the ID of the corresponding clone/iterate mediator that creates split messages. This is particularly useful when aggregating responses for messages that are created using nested clone/iterate mediators.
Aggregation Expression	An XPath expression specifying which elements should be aggregated. A set of messages that are selected for aggregation is determined by the value specified in the <b>Correlation Expression</b> field.
Completion Timeout	The number of seconds taken by the Aggregate mediator to wait for messages. When this time duration elapses, the aggregation will be completed. If the number of response messages reaches the number specified in the <b>Completion Max-messages</b> field, the aggregation will be completed even if the time duration specified for the <b>Completion Timeout</b> field has not elapsed.
Completion Max-messages	Maximum number of messages that can exist in an aggregation. When the number of response messages received reaches this number, the aggregation will be completed.
Completion Min-messages	Minimum number of messages required for the aggregation to complete. When the time duration entered in the <b>Completion Timeout</b> field is elapsed, the aggregation will be completed even if the number of minimum response messages specified has not been received. If no value is entered in the <b>Completion Timeout</b> field, the aggregation will not be completed until the number of response messages entered in the <b>Completion Min-messages</b> field is received.

<b>Correlation Expression</b>	This is an XPath expression which provides the basis on which response messages should be selected for aggregation. This is done by specifying a set of elements for which the messages selected should have matching values. A specific aggregation condition is set via the <b>Aggregation Expression</b> field.
	<p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p>
<b>Enclosing Element Property</b>	This parameter is used to accumulate the aggregated messages inside a single property. The name of the relevant property is entered in this field.
<b>On Complete</b>	<p>The sequence to run when the aggregation is complete. You can select one of the following options:</p> <ul style="list-style-type: none"> <li>• <b>Anonymous:</b> Select this value if you want to specify the sequence to run by adding child mediators to the Aggregate mediator instead of selecting an existing sequence. For example, if you want to send the aggregated message via the <a href="#">Send mediator</a>, you can add the Send mediator as a child mediator.</li> <li>• <b>Pick from Registry:</b> Select this option if you want to specify a sequence which is already defined and saved in the registry. You can select the sequence from the Configuration Registry or Governance Registry. For more information, see <a href="#">Working with the Registry</a>.</li> </ul>

## Note

You can configure the mediator using XML. Click **switch to source** view in the **Mediator** window.

Mediator  switch to source view

## Examples

- [Example 1 - Sending aggregated messages through the send mediator](#)
- [Example 2 - Sending aggregated messages with the enclosing element](#)
- [Samples](#)

### **Example 1 - Sending aggregated messages through the send mediator**

```
<outSequence>
 <aggregate>
 <onComplete expression="//m0:getQuoteResponse"
 xmlns:m0="http://services.samples">
 <send/>
 </onComplete>
 </aggregate>
</outSequence>
```

In this example, the mediator aggregates the responses coming into the ESB, and on completion it sends the aggregated message through the Send mediator.

### **Example 2 - Sending aggregated messages with the enclosing element**

The following example shows how to configure the Aggregate mediator to annotate the responses sent from multiple backends before forwarding them to the client.

```
<outSequence>
 <property name="info" scope="default">
 <ns:Information xmlns:ns="www.asankatechtalks.com" />
 </property>
 <aggregate id="sa">
 <completeCondition />
 <onComplete expression="$body/*[1]" enclosingElementProperty="info">
 <send />
 </onComplete>
 </aggregate>
</outSequence>
```

The above configuration includes the following:

Parameter	Description
<property name="info" scope="default"> <ns:Information xmlns:ns="www.asankatechtalks.com" /> </property>	This creates the property named <code>info</code> of the OM type in which all the aggregated responses are accumulated.
<aggregate id="sa">	The ID of the corresponding Clone mediator that splits the messages to be aggregated by the Aggregate mediator.
<onComplete expression="\$body/*[1]" enclosingElementProperty="info">	This expression is used to add the <code>info</code> property (created earlier in this configuration) to be added to the payload of the message and for accumulating all the aggregated messages from different endpoints inside the tag created inside this property.
<send />	This is the Send mediator added as a child mediator to the Aggregate mediator in order to send the aggregated and annotated messages back to the client once the aggregation is complete.

### Samples

For more examples, see:

- [Sample 62: Routing a Message to a Dynamic List of Recipients and Aggregating Responses](#)
- [Sample 400: Message Splitting and Aggregating the Responses](#)
- [Sample 751: Message Split Aggregate Using Templates](#)

### BAM Mediator

BAM mediator will be deprecated with WSO2 ESB 5.0.0 release. You can use the [Publish Event Mediator](#) to obtain similar functionality.

The BAM mediator captures data events from the ESB mediation sequences and sends them to the [WSO2 Business Activity Monitor](#) server via its Thrift API. This API uses a binary protocol and enables fast data transmission between the ESB and BAM server.

For more information on BAM, see the [BAM documentation](#).

## Syntax

```
<bam xmlns="http://ws.apache.org/ns/synapse">
 <serverProfile name="string">
 <streamConfig name="string" version="string"></streamConfig>
 </serverProfile>
</bam>
```

## UI configuration

Before you can configure a BAM mediator, you must have [configured a BAM server profile and stream](#).

**BAM Mediator**

**BAM Server Profile**

Server Profile\*

Stream Configuration

Stream Name\*

Stream Version\*

Update

Save & Close Save in Registry Save Cancel

Parameters that can be configured for the BAM mediator are as follows.

Parameter Name	Description
<b>Server Profile</b>	The BAM server profile that contains the transport and credential data required to connect to the BAM Thrift server.
<b>Stream Name</b>	The stream used to identify the data to be extracted from the configuration context of the mediation sequence.
<b>Stream Version</b>	The version of the stream used.

## Example

See [Sending Messages to ESB](#) in the BAM documentation for an example.

## Bean Mediator

The **Bean Mediator** is used to manipulate a JavaBean that is bound to the Synapse message context as a property. Classes of objects manipulated by this mediator need to follow the JavaBeans specification.

The Bean mediator is a [content-aware](#) mediator.

## Syntax | UI Configuration | Example

### Syntax

```
<bean action="CREATE | REMOVE | SET_PROPERTY | GET_PROPERTY" var="string"
 [class="string"] [property="string"]
 [value="string | {xpath}"] />
```

### UI Configuration

The screenshot shows the 'Bean Mediator' configuration page. At the top, there's a 'switch to source view' link and a 'Help' button. The main area contains the following fields:

- Class\***: A text input field.
- Action \***: A dropdown menu set to **CREATE**.
- Var\***: A text input field.
- Property**: A text input field.
- Value**: A dropdown menu set to **Value**, followed by a text input field.
- Target**: A dropdown menu set to **Value**, followed by a text input field.

At the bottom, there are 'Update' and 'Save & Close' buttons.

The parameters available to configure the Bean mediator are as follows.

Field Name	Description
<b>Class</b>	The class on which the action selected for the <b>Action</b> parameter is performed by the Beanstalks manager.
<b>Action</b>	The action to be carried out by the Bean mediator. The possible values are: <ul style="list-style-type: none"> <li>• <b>CREATE</b>: This action creates a new JavaBean.</li> <li>• <b>REMOVE</b>: This action removes an existing JavaBean.</li> <li>• <b>SET_PROPERTY</b>: This action sets a property of an existing JavaBean.</li> <li>• <b>GET_PROPERTY</b>: This action retrieves a property of an existing JavaBean.</li> </ul>

<b>Var</b>	The variable which is created, removed, set or retrieved for the JavaBean based on the value selected for the <b>Action</b> parameter.
<b>Property</b>	The name of the property used to bind the JavaBean to the Synapse client.
<b>Value</b>	<p>The value of the property used to bind the JavaBean to the Synapse client. The property value can be entered using one of the following methods.</p> <ul style="list-style-type: none"> <li><b>Value:</b> If this is selected, the property value can be entered as a static value.</li> <li><b>Expression:</b> If this is selected, the property value can be entered as a dynamic value. You can enter the XPath expression to evaluate the relevant property value.</li> </ul> <div style="border: 1px solid #669966; padding: 10px; margin-top: 10px;"> <p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p> </div>

## Note

You can configure the mediator using XML. Click **switch to source** view in the **Mediator** window.



## Example

In this example, the Bean mediator first creates a JavaBean with the `loc` as the variable. The next Bean mediator creates property named `latitude` within the `loc` variable using the `SET_PROPERTY` action. The third Bean mediator creates another property named `longitude` in the same variable using the `SET_PROPERTY` action.

```
...
<bean action="CREATE" class="org.ejb.wso2.test.bean.Location" var="loc"></bean>
<bean action="SET_PROPERTY" property="latitude" value="//latitude" var="loc"
xmlns:ns3="http://org.apache.synapse/xsd"
xmlns:ns="http://org.apache.synapse/xsd"></bean>
<bean action="SET_PROPERTY" property="longitude" value="//longitude" var="loc"
xmlns:ns3="http://org.apache.synapse/xsd"
xmlns:ns="http://org.apache.synapse/xsd"></bean>
...

```

## Cache Mediator

When a message comes to the **Cache mediator**, it checks whether an equivalent message has been seen before based on message hashes. If such a message has existed before, the Cache mediator executes the `onCacheHit` sequence (if specified), fetches the cached response, and prepares the ESB to send the response. The `onCacheHit`

sequence can send back the response message using a [Send Mediator](#). If the `onCacheHit` sequence is not specified, the cached response is sent back to the requester.

The Cache mediator does not cache the response status code of the HTTP response in the cache table. Instead, it returns the "200 OK" status code on a cache hit, which is the default request success status response. If you want to return a different status code when the request gets a cache hit, you can update the response status code in the `onCacheHit` sequence.

The Cache mediator is a [content-aware](#) mediator.

---

## Syntax | UI Configuration | Examples

---

### Syntax

```
<cache [id="string"] [hashGenerator="class"] [timeout="seconds"] [scope=(per-host | per-mediator)]
 collector=(true | false) [maxMessageSize="in-bytes"]>
 <onCacheHit [sequence="key"]>
 (mediator)+
```

(mediator)+

```
</onCacheHit>?
 <implementation type=(memory | disk) maxSize="int"/>
</cache>
```

---

### UI Configuration

Click on the relevant tab to view the UI configuration depending on whether the cache type of the cache mediator is **Finder** or **Collector**.

[Finder](#)[Collector](#)

Mediator [switch to source view](#)

## Cache Mediator

Cache Id	<input type="text"/>
Cache Scope *	<input type="button" value="Per-Host"/>
Cache Type *	<input type="button" value="Finder"/>
Hash Generator *	<input type="text" value="org.wso2.carbon.mediator.cache.dig"/>
Cache Timeout (seconds) *	<input type="text"/>
Maximum Message Size	<input type="text"/>

### Cache Implementation Details

Implementation Type	<input type="button" value="In-Memory"/>
Maximum Size	<input type="text" value="1000"/>

### On Cache Hit

<input type="radio"/> Anonymous	<input type="radio"/> Sequence Reference <input type="text"/>	 Configuration Registry	 Governance Registry
<input type="button" value="Update"/>			

The parameters available to configure the Cache mediator are as follows.

Parameter Name	Description
<b>Cache ID</b>	The ID of the cache configuration.
<b>Cache Scope</b>	The scope of the cache. Possible values are as follows. <ul style="list-style-type: none"> <li>• <b>Per-Host:</b> The cache is kept only for the current host in a cluster.</li> <li>• <b>Per-Mediator:</b> The cache is kept once for the whole cluster.</li> </ul>
<b>Cache Type</b>	This parameter specifies whether the mediator should be in the incoming path (to check the request) or in the outgoing path (to cache the response). Possible values are as follows. <ul style="list-style-type: none"> <li>• <b>Finder:</b> If this is selected, the mediator is used to search for the request hash of incoming messages.</li> <li>• <b>Collector:</b> If this is selected, the mediator is used to collect response messages in the cache.</li> </ul>

<b>Hash Generator</b>	This parameter is used to define the logic used by the mediator to evaluate the has values of incoming messages. The value should be a class implementing the <code>org.wso2.carbon.mediator.cache.digest.DigestGenerator</code> class interface. The default hash generator is <code>org.wso2.carbon.mediator.cache.digest.DOMHASHGenerator</code> . If the generated hash value is found in the cache, then the cache mediator will execute the <code>onCacheHit</code> sequence which can be specified inline or referenced.  <code>org.wso2.carbon.mediator.cache.digest.DOMHASHGenerator</code> uses only the message body to generate the hash value. However, <code>org.wso2.carbon.mediator.cache.digest.REQUESTHASHGenerator</code> considers the recipient (To) address of the request, HTTP headers and XML Payload in generating the hash value. Therefore, this uniquely identifies the HTTP request with the same recipient (To) address, headers and payload.
<b>Cache Timeout (seconds)</b>	The time duration for which the cache is kept. The cache expires once this time duration elapses.
<b>Maximum Message Size</b>	The maximum size of the messages to be cached. This is specified in bytes.
<b>Implementation Type</b>	This parameter is used to specify whether the cache is memory-based or disk-based. Currently, <b>In-Memory</b> is the only value available to be selected.
<b>Maximum Size</b>	The maximum number of elements to be cached. The default size is 1000 bytes.
<b>Anonymous</b>	If this option is selected, an anonymous sequence is executed when an incoming message is identified as an equivalent to a previously received message based on the value defined for the <b>Hash Generator</b> field.
<b>Sequence Reference</b>	The reference to the <code>onCacheHit</code> sequence to be executed when an incoming message is identified as an equivalent to a previously received message based on the value defined for the <b>Hash Generator</b> field. This sequence should be specified in the <a href="#">Registry</a> in order to be selected for this field. You can click either <b>Configuration Registry</b> or <b>Governance Registry</b> as relevant to select the sequence from the resource tree.

Mediator [switch to source view](#)

### Cache Mediator

Cache Id	<input type="text"/>
Cache Scope	<input type="button" value="Per-Host"/>
Cache Type	<input type="button" value="Collector"/>
<input type="button" value="Update"/>	
<input type="button" value="Save &amp; Close"/> <input type="button" value="Save"/> <input type="button" value="Cancel"/>	

The parameters available to configure the Cache mediator are as follows.

Parameter Name	Description
<b>Cache ID</b>	The ID of the cache configuration.
<b>Cache Scope</b>	The scope of the cache. Possible values are as follows. <ul style="list-style-type: none"> <li><b>Per-Host:</b> The cache is kept only for the current host in a cluster.</li> <li><b>Per-Mediator:</b> The cache is kept once for the whole cluster.</li> </ul>
<b>Cache Type</b>	This parameter specifies whether the mediator should be in the incoming path (to check the request) or in the outgoing path (to cache the response). Possible values are as follows. <ul style="list-style-type: none"> <li><b>Finder:</b> If this is selected, the mediator is used to search for the request hash of incoming messages.</li> <li><b>Collector:</b> If this is selected, the mediator is used to collect response messages in the cache.</li> </ul>

## Note

You can configure the mediator using XML. Click **switch to source** view in the **Mediator** window.



## Examples

```
<sequence name="SEQ_CACHE">
<in>
 <cache scope="per-host"
 collector="false"
 hashGenerator="org.wso2.carbon.mediator.cache.digest.DOMHASHGenerator"
 timeout="20">
 <implementation type="memory" maxSize="100"/>
 </cache>

 <send>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
</in>

<out>
 <cache scope="per-host" collector="true"/>
 <send/>
</out>
</sequence>
```

In this example, the first message sent to the endpoint, and the cache is not hit. The Cache mediator configured in

the Out sequence caches the response to this message. When a similar message is sent to the second time, the previous response will be directly fetched from the cache and sent to the requester. This happens because no `onCacheHit` sequence is defined.

## Samples

[Sample 420: Simple Cache Implemented on ESB for the Actual Service.](#)

## Call Mediator

The **Call mediator** is used to send messages out of the ESB to an endpoint. You can invoke services either in blocking or non-blocking manner.

When you invoke a service in non-blocking mode, the underlying worker thread returns without waiting for the response. In blocking mode, the underlying worker thread gets blocked and waits for the response after sending the request to the endpoint. Call mediator in blocking mode is very much similar to the [Callout mediator](#).

In both blocking and non-blocking modes, Call mediator behaves in a synchronous manner. Hence, mediation pauses after the service invocation, and resumes from the next mediator in the sequence when the response is received. Call mediator allows you to create your configuration independent from the underlying architecture.

Non-blocking mode of the Call mediator leverages the non-blocking transports for better performance. Therefore, it is recommended to use it in non-blocking mode as much as possible. However, there are scenarios where you need to use the blocking mode. For example, when you implement a scenario related to JMS transactions, it is vital to use the underlying threads in blocking mode.

In blocking mode, Call mediator uses the `<ESB_HOME>/repository/conf/axis2/axis2_blocking_client.xml` file as the Axis2 configuration. For more information about the blocking transport related parameters that can be configured for the Call mediator, see [Configuring axis2\\_blocking\\_client.xml](#).

You can obtain the service endpoint for the Call mediator as follows:

- Pick from message-level information
- Pick from a pre-defined endpoint

If you do not specify an endpoint, the Call mediator tries to send the message using the `WSA:TO` address of the message. If you specify an endpoint, the Call mediator sends the message based on the specified endpoint.

The endpoint type can be Leaf Endpoint (i.e. Address/WSDL/Default/HTTP) or Group Endpoint (i.e. Failover/Load balance/Recipient list). Group Endpoint is only supported in non-blocking mode.

The Call mediator is a content-unaware mediator.

---

[Syntax](#) | [UI Configuration](#) | [Example](#)

---

## Syntax

```
<call [blocking="true"] />
```

If the message is to be sent to one or more endpoints, use the following syntax:

```
<call [blocking="true"]>
 (endpointref | endpoint) +
</call>
```

- The endpointref token refers to the following:

```
<endpoint key="name" />
```

- The endpoint token refers to an anonymous endpoint definition.

## UI Configuration

**Call Mediator**

Select Endpoint Type

None

Define Inline

Pick From Registry

XPath

Enable Blocking Calls

Blocking

Select one of the following options to define the endpoint to which, the message should be delivered.

Parameter Name	Description
<b>None</b>	Select this option if you do not want to provide an endpoint. The Call mediator will send the message using its <code>wsa:to</code> address.
<b>Define Inline</b>	If this is selected, the endpoint to which the message should be sent can be included within the Call mediator configuration. Click <b>Add</b> to add the required endpoint. For more information on Adding an endpoint, see <a href="#">Adding an Endpoint</a> .
<b>Pick From Registry</b>	If this is selected, the message can be sent to a pre-defined endpoint which is currently saved as a resource in the <a href="#">registry</a> . Click either <b>Configuration Registry</b> or <b>Governance Registry</b> as relevant to select the required endpoint from the resource tree.

<b>XPath</b>	If this is selected, the endpoint to which the message should be sent will be derived via an XPath expression. You are required to enter the relevant XPath expression in the text field that appears when this option is selected.
<p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p>	
<b>Blocking</b>	If set to <code>true</code> , you can use the call mediator in blocking mode.

## Example

### Example 1 - Service orchestration

In this example, the Call mediator invokes a backend service. An [Enrich mediator](#) stores the response received for that service invocation.

The [Filter Mediator](#) added after the Call mediator carries out a filter to determine whether the first call has been successful. If it is successful, second backend service is invoked. The payload of the request to the second backend is the response of the first service invocation.

After a successful second backend service invocation, response of the first service is retrieved by the [Enrich mediator](#) from the property where it was formerly stored. This response is sent to the client by the [Respond mediator](#).

If it is not successful, a custom JSON error message is sent with HTTP 500. If the first call itself is not successful, the output is just sent back with the relevant error code.

```
<target>
 <inSequence>
 <log/>
 <call>
 <endpoint>
 <http method="get"
uri-template="http://192.168.1.10:8088/mockaxis2service"/>
 </endpoint>
 </call>
 <enrich>
 <source type="body" clone="true"/>
 <target type="property" action="child" property="body_of_first_call"/>
 </enrich>
 <filter source="get-property('axis2', 'HTTP_SC')" regex="200">
 <then>
 <log level="custom">
 <property name="switchlog" value="Case: first call successful"/>
 </log>
 <call>
 <endpoint>
 <http method="get"
uri-template="http://localhost:8080/MockService1"/>
 </endpoint>
 </call>
 <filter source="get-property('axis2', 'HTTP_SC')" regex="200">
 <then>
 <log level="custom">
```

```
 <property name="switchlog" value="Case: second call
successful"/>
 </log>
 <enrich>
 <source type="property" clone="true"
property="body_of_first_call"/>
 <target type="body"/>
 </enrich>
 <respond/>
 </then>
 <else>
 <log level="custom">
 <property name="switchlog" value="Case: second call
unsuccessful"/>
 </log>
 <property name="HTTP_SC" value="500" scope="axis2"/>
 <payloadFactory media-type="json">
 <format>{ "status": "ERROR!" }</format>
 <args/>
 </payloadFactory>
 <respond/>
 </else>
 </filter>
 </then>
 <else>
 <log level="custom">
 <property name="switchlog" value="Case: first call unsuccessful"/>
 </log>
 <respond/>
 </else>
```

```

</filter>
</inSequence>
</target>

```

#### Example 2 - Continuing mediation without waiting for responses

In this example, the message will be cloned by the [Clone Mediator](#) and sent via the Call mediator. The Drop mediator drops the response so that no further mediation is carried out for the cloned message. However, since the `continueParent` attribute of the [Clone mediator](#) is set to `true`, the original message is mediated in parallel. Therefore, the [Log Mediator](#) at the end of the configuration will log the `After call mediator` log message without waiting for the Call mediator response.

```

...
<log level="full"/>
<clone continueParent="true">
<target>
<sequence>
<call>
<endpoint>
<address uri="http://localhost:8080/echoString"/>
</endpoint>
</call>
<drop/>
</sequence>
</target>
</clone>
<log level="custom">
<property name="MESSAGE" value="After call mediator"/>
</log>
...

```

#### Example 3 - Call mediator in blocking mode

In the following sample configuration, the [Header Mediator](#) is used to add the action, the [PayloadFactory Mediator](#) is used to store the the request message and the Call mediator is used to invoke a backend service. You will see that the payload of the request and header action are sent to the backend. After successful backend service invocation, you will see that the response of the service is retrieved by the ESB and sent to the client as the response using the [Respond Mediator](#).

```

<target>
 <inSequence>
 <header name="Action" value="urn:getQuote" />
 <payloadFactory media-type="xml">
 <format>
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>WSO2</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </format>
 <args />
 </payloadFactory>
 <call blocking="true">
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </call>
 <respond />
 </inSequence>
</target>

```

#### Example 4 - Receiving response headers in blocking mode

If you want to receive the response message headers, when you use the Call mediator in blocking mode, add the `BLOCKING_SENDER_PRESERVE_REQ_HEADERS` property within the proxy service, or in a sequence as shown in the sample proxy configuration below.

Set the value of the `BLOCKING_SENDER_PRESERVE_REQ_HEADERS` property to `false`, to receive the response message headers. If you set it to `true`, you cannot get the response headers, but the request headers will be preserved.

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="sample"
 transports="https"
 statistics="enable"
 trace="enable"
 startOnLoad="true">
 <target>
 <inSequence>
 <property name="FORCE_ERROR_ON_SOAP_FAULT"
 value="true"
 scope="default"
 type="STRING"/>
 <property name="HTTP_METHOD" value="POST" scope="axis2" type="STRING"/>
 <property name="messageType" value="text/xml" scope="axis2" type="STRING"/>
 <property name="BLOCKING_SENDER_PRESERVE_REQ_HEADERS" value="false" />
 <call blocking="true">
 <endpoint>
 <address uri="https://localhost:8243/services/sampleBE"
 trace="enable"
 statistics="enable"/>
 </endpoint>
 </call>
 </inSequence>
 <outSequence/>
 </target>
 <description/>
 </proxy>

```

## Samples

For another example, see [Sample 500: Call Mediator for Non-Blocking Service Invocation](#).

### Call Template Mediator

The Call Template mediator allows you to construct a sequence by passing values into a [sequence template](#).

This is currently only supported for special types of mediators such as the [Iterator](#) and [Aggregate Mediators](#), where actual XPath operations are performed on a different SOAP message, and not on the message coming into the mediator.

## Syntax

```

<call-template target="string">
 <!-- parameter values will be passed on to a sequence template -->
 (
 <!--passing plain static values -->
 <with-param name="string" value="string" /> |
 <!--passing xpath expressions -->
 <with-param name="string" value="{string}" /> |
 <!--passing dynamic xpath expressions where values will be compiled
dynamically-->
 <with-param name="string" value="{{string}}"/> |
) *
 <!--this is the in-line sequence of the template -->
</call-template>

```

You use the `target` attribute to specify the sequence template you want to use. The `<with-param>` element is used to parse parameter values to the target sequence template. The parameter names should be the same as the names specified in target template. The parameter value can contain a string, an XPath expression (passed in with curly braces `{ }`), or a dynamic XPath expression (passed in with double curly braces) of which the values are compiled dynamically.

## UI Configuration

The screenshot shows the configuration interface for the Call-Template Mediator. At the top, there's a header with 'Mediator' and a 'switch to source view' link. The main title is 'Call-Template Mediator'. Below the title, there are two input fields: 'Target Template\*' and 'Available Templates' with a dropdown menu labeled 'Select From Templates'. Underneath these, there's a section titled 'Parameters of the Template mediator' and a 'Update' button. At the very bottom, there are three buttons: 'Save & Close', 'Save', and 'Cancel'.

The parameters available to configure the Call-Template mediator are as follows.

Parameter Name	Description
<b>Target Template</b>	The sequence template to which values should be passed. You can select a template from the <b>A vailable Templates</b> list

When a target template is selected, the parameter section will be displayed as shown below if the sequence template selected has any parameters. This enables parameter values to be parsed into the sequence template selected.

**Parameters of the Template mediator**

Parameter Name	Parameter Type	Value / Expression	Action
<input type="text"/>	Value 	<input type="text"/>	 Delete
<b>Update</b>			
<b>Save &amp; Close</b> <b>Save</b> <b>Cancel</b>			

Parameter Name	Description
Parameter Name	The name of the parameter.
Parameter Type	The type of the parameter. Possible values are as follows. <ul style="list-style-type: none"> <li><b>Value:</b> Select this to define the parameter value as a static value. This value should be entered in the <b>Value / Expression</b> parameter.</li> <li><b>Expression:</b> Select this to define the parameter value as a dynamic value. The XPath expression to calculate the parameter value should be entered in the <b>Value / Expression</b> parameter.</li> </ul>
Value / Expression	The parameter value. This can be a static value, or an XPath expression to calculate a dynamic value depending on the value you selected for the <b>Parameter Type</b> parameter.
Action	Click <b>Delete</b> to delete a parameter.

## Example

The following four Call Template mediator configurations populate a sequence template named HelloWorld\_Logger with the "hello world" text in four different languages.

```
<call-template target="HelloWorld_Logger">
 <with-param name="message" value="HELLO WORLD!!!!!!" />
</call-template>
```

```
<call-template target="HelloWorld_Logger">
 <with-param name="message" value="Bonjour tout le monde!!!!!!" />
</call-template>
```

```
<call-template target="HelloWorld_Logger">
 <with-param name="message" value="Ciao a tutti!!!!!!" />
</call-template>
```

```
<call-template target="HelloWorld_Logger">
 <with-param name="message" value="??????!!!!!!" />
</call-template>
```

The sequence template can be configured as follows to log any greetings message passed to it by the Call

Template mediator. Thus, due to the availability of the Call Template mediator, you are not required to have the message entered in all four languages included in the sequence template configuration itself.

```
<template name="HelloWorld_Logger">
 <parameter name="message"/>
 <sequence>
 <log level="custom">
 <property name="GREETING_MESSAGE" expression="$func:message" />
 </log>
 </sequence>
</template>
```

See [Sequence Template](#) for a more information about this scenario.

### Callout Mediator

The **Callout** mediator performs a blocking external service invocation during mediation.

As the Callout mediator performs a blocking call, it cannot use the default non-blocking HTTP/S transports based on Java NIO. Instead, it defaults to using `<ESB_HOME>/repository/conf/axis2/axis2_blocking_client.xml` as the Axis2 configuration, and `<ESB_HOME>/repository/deployment/client` as the client repository unless these are specified separately. See [Configuring axis2\\_blocking\\_client.xml](#) for more information about the blocking transport related parameters that can be configured for the Callout mediator.

The **Call** mediator leverages the non-blocking transports for much greater performance than the Callout mediator, so you should use the Call mediator in most cases. However, the Callout mediator is recommended in situations where you need to execute the mediation flow in a single thread.

---

[Setup](#) | [Syntax](#) | [UI configuration](#) | [Examples](#)

---

### Setup

#### ***Enabling mutual SSL***

Since the Callout mediator is run based on the configuration of the `axis2_blocking_client.xml` file, its default https transport sender is `org.apache.axis2.transport.http.CommonsHTTPTransportSender`. Therefore, the Callout mediator does not have access to the required key store to handle mutual SSL. To enable the Callout mediator to handle mutual SSL, the following JVM settings should be added to the `<ESB_HOME>/bin/wso2server.sh` file.

```
-Djavax.net.ssl.keyStore="$CARBON_HOME/repository/resources/security/wso2carbon.jks" \
-Djavax.net.ssl.keyStorePassword="wso2carbon" \
-Djavax.net.ssl.keyPassword="wso2carbon" \
```

#### ***Disabling chunking***

The Callout mediator is not affected by the `DISABLE_CHUNKING` property. Instead, you can disable chunking for the Callout mediator by setting the `Transfer-Encoding` parameter to `none` in `CommonsHTTPTransportSender` of `axis2_blocking_client.xml` as follows:

```
<parameter name="Transfer-Encoding">none</parameter>
```

For example:

```

<transportSender name="http"
 class="org.apache.axis2.transport.http.CommonsHTTPTransportSender">
 <parameter name="PROTOCOL">HTTP/1.1</parameter>
 <parameter name="Transfer-Encoding">none</parameter>
 <parameter name="cacheHttpClient">true</parameter>
 <parameter name="defaultMaxConnectionsPerHost">200</parameter>
</transportSender>

```

This will disable chunking for all Callout mediators present in the ESB server.

If you want to disable chunking for only a single Callout mediator instance, create a new `axis2.xml` file by copying the `axis2_blocking_client.xml` file, set the `Transfer-Encoding` parameter as shown, and then configure that Callout mediator to use this new `axis2.xml` file as described below.

## Syntax

```

<callout [serviceURL="string"] [action="string"] [initAxis2ClientOptions="true|false"]
[endpointKey="string"]>
 <configuration [axis2xml="string"] [repository="string"]/>?
 <source xpath="expression" | key="string" | type="envelope"/>
 <target xpath="expression" | key="string"/>
 <enableSec policy="string" | outboundPolicy="String" | inboundPolicy="String"
/>?
</callout>

```

## UI configuration

There will be a slight difference in the UI depending on the option you select for the **Specify as** parameter. Click on the relevant tab to view the required UI.

URLAddress Endpoint

Mediator switch to source view

## Callout Mediator

### Service

Specify as :  URL  Address Endpoint

URL

Action

Axis2 Repository

Axis2 XML

initAxis2ClientOptions

### Source \*

Specify as :  XPath  Property  Envelope

Property

### Target \*

Specify as :  XPath  Property

Property

### WS-Security

WS-Security

Mediator [switch to source view](#)

### Callout Mediator

**Service**

Specify as :  URL  Address Endpoint

Address Endpoint  [Configuration Registry](#) [Governance Registry](#)

Action

Axis2 Repository

Axis2 XML

initAxis2ClientOptions

**Source \***

Specify as :  XPath  Property  Envelope

Property

**Target \***

Specify as :  XPath  Property

Property

**WS-Security**

WS-Security

[Update](#)

[Save & Close](#) [Save](#) [Cancel](#)

The parameters available for configuring the Callout mediator are as follows.

Parameter Name	Description
<b>Specify As</b>	<p>This parameter determines whether the target external service should be configured by using the <code>endpointKey</code> attribute.</p> <p>Callout mediator does not support endpoint configurations such as <code>timeout</code>, <code>suspend</code> etc. Instead the <code>endpointKey</code> attribute is used to specify an existing endpoint.</p> <ul style="list-style-type: none"> <li>• <b>URL:</b> Select <b>URL</b> if you want to call the external service by specifying its URL in the Configuration Registry.</li> <li>• <b>Address Endpoint:</b> Selected <b>Address Endpoint</b> if you want to call the external service using the Configuration Registry. This option should be selected if you want to make use of the WSO2 function conversions, security etc. Note that only Leaf endpoint types (i.e. Address , WSDL , Callout mediator).</li> </ul> <p>If neither a URL or an address endpoint is specified, the <code>To</code> header on the request is used.</p>
<b>URL</b>	If you selected <b>URL</b> for the <b>Specify As</b> parameter, use this parameter to enter the URL of the external service. The URL will be used as the End Point Reference (EPR) of the external service.
<b>Address Endpoint</b>	If you selected <b>Address Endpoint</b> for the <b>Specify As</b> parameter, use this parameter to enter the URL of the external service. Click <b>Configuration Registry</b> or <b>Governance Registry</b> to browse the resource tree.

Action	The SOAP action which should be appended to the service call.						
Axis2 Repository	The path to Axis2 client repository where the services and modules are located. The purp mediator initialize with the required client repository.						
Axis2 XML	The path to the location of the Axis2.xml configuration file. The purpose of this parameter relevant Axis2 configurations.						
initAxis2ClientOptions	If this parameter is set to <code>false</code> , the existing Axis2 client options available in the Synaps mediator is invoked. This is useful when you want to use NTLM authentication. The defau						
Source	<p>This parameter defines the payload for the request. It can be defined using one of the foll</p> <ul style="list-style-type: none"> <li>• <b>XPath</b> - This option allows you to specify an expression that defines the location in th</li> </ul> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expres appear where you can provide any number of namespace prefixes and URLs u</p> </div> <ul style="list-style-type: none"> <li>• <b>Property</b> - This option allows you to specify the payload for a request via a property i</li> <li>• <b>Envelope</b> - This option allows you to select the entire envelope which is available in t</li> </ul>						
Target	<p>The node or the property of the request message to which the payload (resulting from the be attached. The target can be specified using one of the following options.</p> <ul style="list-style-type: none"> <li>• <b>XPath</b> - This option allows you to specify an expression that defines the location in th</li> </ul> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expres appear where you can provide any number of namespace prefixes and URLs u</p> </div> <ul style="list-style-type: none"> <li>• <b>Property</b> - This option allows you to specify a property included in the mediation flow</li> </ul>						
WS-Security	<p>If this check box is selected, WS-Security is enabled for the Callout mediator. This sectio this check box.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p><b>WS-Security</b></p> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 30%;">WS-Security</td> <td style="text-align: center;"><input checked="" type="checkbox"/></td> </tr> <tr> <td>Specify as Inbound and Outbound Policies</td> <td style="text-align: center;"><input type="checkbox"/></td> </tr> <tr> <td>Policy Key</td> <td style="text-align: right; vertical-align: bottom;"> <input style="width: 150px;" type="text"/> </td> </tr> </table> </div>	WS-Security	<input checked="" type="checkbox"/>	Specify as Inbound and Outbound Policies	<input type="checkbox"/>	Policy Key	<input style="width: 150px;" type="text"/>
WS-Security	<input checked="" type="checkbox"/>						
Specify as Inbound and Outbound Policies	<input type="checkbox"/>						
Policy Key	<input style="width: 150px;" type="text"/>						

<b>Specify as Inbound and Outbound Policies</b>	If this check box is selected, you can define separate security policies for the inbound and outbound messages by entering the required policy keys in the <b>Outbound Policy Key</b> and <b>Inbound Policy Key</b> fields. If this check box is not selected, a single security policy which will be applied to both inbound and outbound messages. You can click <b>Configuration Registry</b> or <b>Governance Registry</b> from the resource tree.
<b>Policy Key</b>	If the <b>Specify as Inbound and Outbound Policies</b> check box is not selected, this parameter defines a security policy which will be applied to both inbound and outbound messages. You can click <b>Configuration Registry</b> or <b>Governance Registry</b> from the resource tree to select a security policy saved in the <b>Registry</b> from the resource tree.

## Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.



## Examples

```
<callout serviceURL="http://localhost:9000/services/SimpleStockQuoteService"
 action="urn:getQuote">
 <source xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
 xpath="s11:Body/child::*[fn:position()=1] |
s12:Body/child::*[fn:position()=1]" />
 <target xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
 xpath="s11:Body/child::*[fn:position()=1] |
s12:Body/child::*[fn:position()=1]" />
</callout>
```

In this example, the Callout Mediator does the direct service invocation to the StockQuoteService using the client request, gets the response, and sets the response as the first child of the SOAP message body. You can then use the **Send Mediator** to send the message back to the client.

## Samples

[Sample 430: Callout Mediator for Synchronous Service Invocation](#) .

## Class Mediator

The **Class Mediator** creates an instance of a custom-specified class and sets it as a mediator. The class must implement the `org.apache.synapse.api.Mediator` interface. If any properties are specified, the corresponding setter methods are invoked once on the class during initialization.

Use the Class mediator for user-specific, custom developments only when there is no built-in mediator that already provides the required functionality. Maintaining custom classes incurs a high overhead. Therefore avoid using them unless the scenario is frequently re-used and very user-specific.

For best results, use [WSO2 Developer Studio](#) for debugging Class mediators.

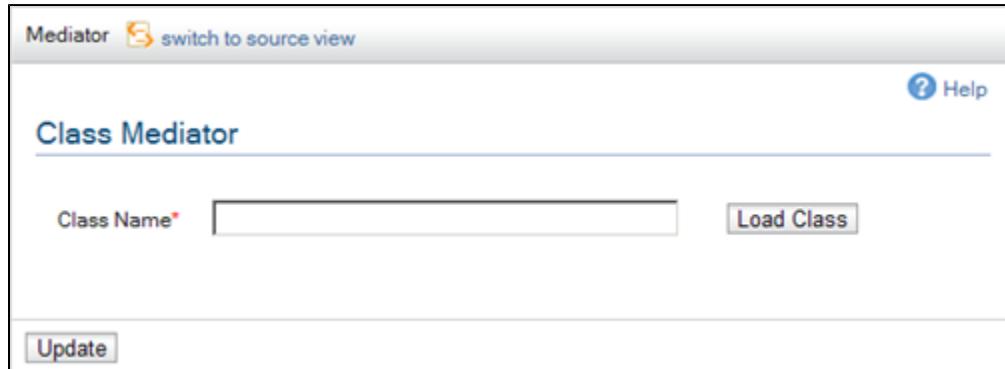
## Syntax | UI Configuration | Examples

---

### Syntax

```
<class name="class-name">
 <property name="string" value="literal">
 </property>
</class>
```

### UI Configuration



The screenshot shows the 'Class Mediator' configuration page. At the top, there is a 'Mediator' tab and a 'switch to source view' button. Below the tabs, there is a 'Help' button. The main area has a title 'Class Mediator'. It contains a 'Class Name\*' input field with a placeholder 'Enter class name' and a 'Load Class' button. At the bottom, there is a 'Update' button.

**Class Name:** The name of the class. To load a class, enter the qualified name of the relevant class in this parameter and click **Load Class**.

### Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.



A screenshot of the 'Mediator' configuration window. The 'switch to source view' button is highlighted with a red oval.

### Examples

In this configuration, the ESB sends the requested message to the endpoint specified via the [Send mediator](#). This endpoint is the Axis2server running on port 9000. The response message is passed through a Class mediator before it is sent back to the client. Two parameters named `variable1` and `variable2` are passed to the instance mediator implementation class (`SimpleClassMediator`).

```

<definitions xmlns="http://ws.apache.org/ns/synapse">

 <sequence name="fault">
 <makefault>
 <code value="tns:Receiver"
xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason value="Mediation failed."/>
 </makefault>
 <send/>
 </sequence>

 <sequence name="main" onError="fault">
 <in>
 <send>
 <endpoint name="stockquote">
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </in>
 <out>
 <class name="samples.mediators.SimpleClassMediator">
 <property name="variable1" value="10"/>
 <property name="variable2" value="5"/>
 </class>
 <send/>
 </out>
 </sequence>

</definitions>

```

See the following sample Class Mediator and note the `SynapseMessageContext` and the full Synapse API in there.

```

package samples.mediators;

import org.apache.synapse.MessageContext;
import org.apache.synapse.mediators.AbstractMediator;
import org.apache.axiom.om.OMElement;
import org.apache.axiom.om.OMAbstractFactory;
import org.apache.axiom.om.OMFactory;
import org.apache.axiom.soap.SOAPFactory;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import javax.xml.namespace.QName;

public class SimpleClassMediator extends AbstractMediator {

 private static final Log log = LogFactory.getLog(SimpleClassMediator.class);

 private String variable1="10";

 private String variable2="10";

 private int variable3=0;
}

```

```
public SimpleClassMediator(){}

public boolean mediate(MessageContext mc) {
 // Do something useful..
 // Note the access to the Synapse Message context
 return true;
}

public String getType() {
 return null;
}

public void setTraceState(int traceState) {
 traceState = 0;
}

public int getTraceState() {
 return 0;
}

public void setVariable1(String newValue) {
 variable1=newValue;
}

public String getVariable1() {
 return variable1;
}

public void setVariable2(String newValue){
 variable2=newValue;
}

public String getVariable2(){
 return variable2;
```

```

 }
}

```

## Samples

For more examples, see:

- Sample 380: Writing your own Custom Mediation in Java
- Sample 381: Class Mediator to CBR Binary Messages

## Clone Mediator

The **Clone Mediator** can be used to clone a message into several messages. It resembles the Scatter-Gather enterprise integration pattern. The Clone mediator is similar to the [Iterate mediator](#). The difference between the two mediators is, the Iterate mediator splits a message into different parts, whereas the Clone mediator makes multiple identical copies of the message.

The Clone mediator is a [content-aware mediator](#).

## Syntax | UI Configuration | Example

### Syntax

```

<clone [continueParent=(true | false)]>
 <target [to="uri"] [soapAction="qname"] [sequence="sequence_ref"]
 [endpoint="endpoint_ref"]>
 <sequence>
 (mediator)+?
 </sequence>?
 <endpoint>
 endpoint
 </endpoint>?
 </target>+
</clone>

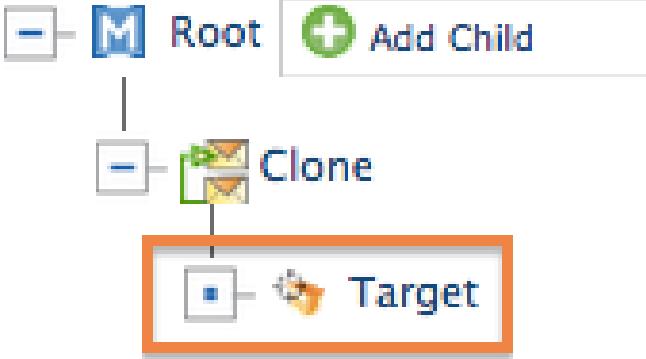
```

### UI Configuration

The screenshot shows the 'Clone Mediator' configuration screen in the WSO2 ESB UI. The interface includes a header with 'Mediator' and a 'switch to source view' link, and a 'Help' button. The main form contains the following fields:

- Clone ID:** A text input field.
- Sequential Mediation:** A dropdown menu set to "No".
- Continue Parent:** A dropdown menu set to "No".
- Number of clones:** A text input field showing "0".
- Add Clone Target:** A link to add more targets.
- Update:** A button at the bottom left.

The parameters available to configure the Clone mediator is as follows.

Parameter Name	Description
<b>Clone ID</b>	Identification of messages created by the clone mediator. This is particularly useful when aggregating responses of messages that are created using nested Clone mediators.
<b>Sequential Mediation</b>	This parameter is used to specify whether the cloned messages should be processed sequentially or not. The processing is carried based on the information relating to the sequence and endpoint specified in the <a href="#">target configuration</a> . The possible values are as follows. <ul style="list-style-type: none"> <li>• <b>Yes:</b> If this is selected, the cloned messages will be processed sequentially. Note that selecting <b>True</b> might cause delays due to high resource consumption.</li> <li>• <b>No:</b> If this is selected, the cloned messages will not be processed sequentially. This is the default value and it results in better performance.</li> </ul>
<b>Continue Parent</b>	This parameter is used to specify whether the original message should be preserved or not. Possible values are as follows. <ul style="list-style-type: none"> <li>• <b>Yes:</b> If this is selected, the original message will be preserved.</li> <li>• <b>No:</b> If this is selected, the original message will be discarded. This is the default value.</li> </ul>
<b>Number of Clones</b>	The parameter indicates the number of targets which currently exist for the Clone mediator. Click <b>Add Clone Target</b> to add a new target. Each time you add a target, it will be added as a child of the Clone mediator in the mediator tree as shown below.  <p>Click <b>Target</b> to add the target configuration as described below.</p>

Target configuration

The following section is displayed in the mediator page when you click **Target** as mentioned above.

Mediator switch to source view

[? Help](#)

## Target Configuration

SOAP Action

To Address

**Sequence**

None

Anonymous

Pick From Registry

**Endpoint**

None

Anonymous

Pick From Registry

**Update**

**Save & Close** **Save** **Cancel**

The parameters available to configure the target are as follows.

Parameter Name	Description
<b>SOAP Action</b>	The SOAP action of the message.
<b>To Address</b>	The target endpoint address.
<b>Sequence</b>	This parameter is used to specify whether cloned messages should be mediated via a <a href="#">sequence</a> or not, and to specify the sequence if they are to be further mediated. Possible options are as follows. <ul style="list-style-type: none"> <li>• <b>None:</b> If this is selected, no further mediation will be performed for the cloned messages.</li> <li>• <b>Anonymous:</b> If this is selected, you can define an anonymous <a href="#">sequence</a> for the cloned messages by adding the required mediators as children to <b>Target</b> in the mediator tree.</li> <li>• <b>Pick From Registry:</b> If this is selected, you can refer to a pre-defined <a href="#">sequence</a> that is currently saved as a resource in the <a href="#">registry</a>. Click either <b>Configuration Registry</b> or <b>Governance Registry</b> as relevant to select the required <a href="#">sequence</a> from the resource tree.</li> </ul>
<b>Endpoint</b>	The <a href="#">endpoint</a> to which the cloned messages should be sent. Possible options are as follows. <ul style="list-style-type: none"> <li>• <b>None:</b> If this is selected, the cloned messages are not sent to an <a href="#">endpoint</a>.</li> <li>• <b>Anonymous:</b> If this is selected, you can define an anonymous <a href="#">endpoint</a> within the iterate target configuration to which the cloned messages should be sent. Click the <b>Add</b> link which appears after selecting this option to add the anonymous <a href="#">endpoint</a>. See <a href="#">Adding an Endpoint</a> for further information.</li> <li>• <b>Pick from Registry:</b> If this is selected, you can refer to a pre-defined <a href="#">endpoint</a> that is currently saves as a resource in the registry. Click either <b>Configuration Registry</b> or <b>Governance Registry</b> as relevant to select the required <a href="#">endpoint</a> from the resource tree.</li> </ul>

## Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.



Mediator  switch to source view

### Example

In this example, the Clone Mediator clones messages and redirects them to a default endpoint and an existing sequence.

```
<clone xmlns="http://ws.apache.org/ns/synapse">
 <target>
 <endpoint name="endpoint_urn_uuid_73A47733EB1E6F30812921609540392-849227072">
 <default />
 </endpoint>
 </target>
 <target sequence="test1" />
</clone>
```

### Conditional Router Mediator

The **Conditional Router Mediator** specifies how a message should be routed based on given conditions. The specified target **sequence** is applied if the condition of the mediator evaluates to **true**.

The Conditional Router mediator is a **content-aware** mediator.

---

[Syntax](#) | [UI Configuration](#) | [Example](#)

---

### Syntax

```
<conditionalRouter continueAfter="(true|false)">
 <route breakRoute="(true|false)">
 <condition/>
 <target/>
 </route>+
</conditionalRouter>
```

---

[UI Configuration](#)

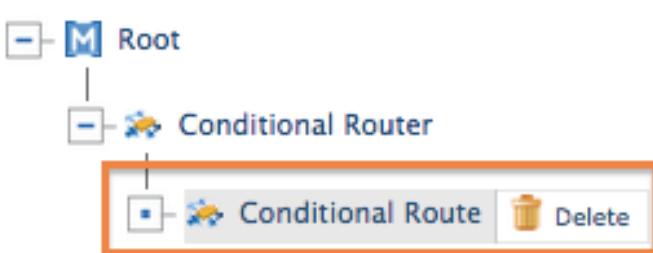
Mediator [switch to source view](#)

## Conditional Router Mediator

Continue after routing

[+ Add Route](#)

The parameters available to configure the Conditional Router mediator are as follows.

Parameter Name	Description
<b>Continue after Routing</b>	This parameter specifies whether routing should be continued if any of the child routes are executed. Possible values are as follows. <ul style="list-style-type: none"> <li><b>Yes:</b> If this is selected, routing is continued after child routes are executed.</li> <li><b>No:</b> If this is selected, routing is not continued after the child routes are executed. This is the default value.</li> </ul>
<b>Add Route</b>	Click this link to add a route. The conditional route will be added as a child to the Conditional Router mediator in the mediator tree as shown below.   You can add multiple conditional routes to a Conditional Router mediator by clicking on this link.

Click on the conditional route in the mediator tree to configure it. The parameters available to configure a conditional route are as follows.

Mediator [switch to source view](#)

## Conditional Route Configuration

Break after route

Evaluator Expression\*

Target Sequence\*

[Configuration Registry](#) [Governance Registry](#)

Parameter Name	Description
<b>Break after route</b>	Possible values for this parameter are as follows. <ul style="list-style-type: none"> <li><b>True</b>: If this is selected, a matching route would break the router.</li> <li><b>False</b>: If this is selected, a matching route would not break the router.</li> </ul>
<b>Evaluator Expression</b>	The expression to evaluate the condition based on which the target mediation sequence should be applied.
<b>Target Sequence</b>	The mediation sequence to be applied if the expression entered in the <b>Evaluator Expression</b> parameter evaluates to true.

## Example

See [Sample 157: Conditional Router for Routing Messages based on HTTP URL, HTTP Headers and Query Parameters](#) for an example of the Conditional Router mediator.

## Data Mapper Mediator

Data Mapper mediator is a data mapping solution that can be integrated into a mediation sequence. It converts and transforms one data format to another, or changes the structure of the data in a message. It provides a WSO2 Developer Studio-based tool to create a graphical mapping configuration and generates the files required to execute this graphical mapping configuration by the WSO2 Data Mapper engine.

WSO2 Data Mapper is an independent component that does not depend on any other WSO2 product. However, other products can use the Data Mapper to achieve/offer data mapping capabilities. Data Mapper Mediator is the intermediate component you need for that, which gives the data mapping capability into WSO2 ESB.

Data Mapper mediator finds the configuration files from the Registry and configures the Data Mapper Engine with the input message type (XML/JSON/CSV) and output message type (XML/JSON/CSV). Then it takes the request message from the WSO2 ESB message flow and uses the configured Data Mapper Engine to execute the transformation and adds the output message to the ESB message flow.

The Data Mapper mediator is a [content-aware](#) mediator.

- Prerequisites
- Syntax
- UI configuration
- Components of Data Mapper
- Data Mapper element and attribute types
- Data Mapper operations
- Examples
- Samples

## Prerequisites

You need to install the WSO2 Developer Studio ESB Tool 5.0.0 to use the Data Mapper mediator. For instructions on installing this WSO2 ESB Tooling Plugin, see [Installing WSO2 ESB Tooling](#).

## Syntax

```
<datamapper config="gov:datamapper/FoodMapping.dmc"
inputSchema="gov:datamapper/FoodMapping_inputSchema.json" inputType="XML"
outputSchema="gov:datamapper/FoodMapping_outputSchema.json" outputType="XML" />
```

## UI configuration

The screenshot shows the configuration interface for the DataMapper Mediator. At the top, there's a 'Mediator' tab and a 'switch to source view' button. Below the tabs, the title 'DataMapper Mediator' is displayed. The configuration parameters are listed in a table-like structure:

Mapping Configuration	gov:datamapper/FoodMapping.dmc	Configuration Registry	Governance Registry
Input Schema	gov:datamapper/FoodMapping_inputSchema.js	Configuration Registry	Governance Registry
Output Schema	gov:datamapper/FoodMapping_outputSchema.js	Configuration Registry	Governance Registry
Input Type	XML		
Output Type	XML		

At the bottom left is a 'Update' button.

The parameters available for configuring the Data Mapper mediator are as follows.

Parameter name	Description
<b>Mapping Configuration</b>	The file, which contains the script file that is used to execute the mapping. You need to create a mapping configuration file using the Dev Studio-based Tooling plugin, and store it either in the Configuration Registry or Governance Registry, to select and upload it from here.
<b>Input Schema</b>	JSON schema, which represents the input message format. You need to create an input schema file using the Dev Studio-based Tooling plugin, and store it either in the Configuration Registry or Governance Registry to select and upload it from here.
<b>Output Schema</b>	JSON schema, which represents the output message format. You need to create an output schema file using the Dev Studio-based Tooling plugin, and store it either in the Configuration Registry or Governance Registry to select and upload it from here.
<b>Input Type</b>	Expected input message type (XML/JSON/CSV)
<b>Output Type</b>	Target output message type (XML/JSON/CSV)

### Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.

## Components of Data Mapper

WSO2 Data Mapper consists of two components. They are Data Mapper Tooling and Data Mapper Engine.

### Data Mapper Tooling

Data Mapper Tooling component is the interface used to create configuration files that are required by the Data

Mapper Engine to execute the mapping. Following three configuration files are needed by the Data Mapper engine.

- Input schema file
- Output schema file
- Mapping configuration file

These three files are generated by the Data Mapper Tool and saved in a Registry Resource project, which you deploy in a WSO2 server as shown in the example below.

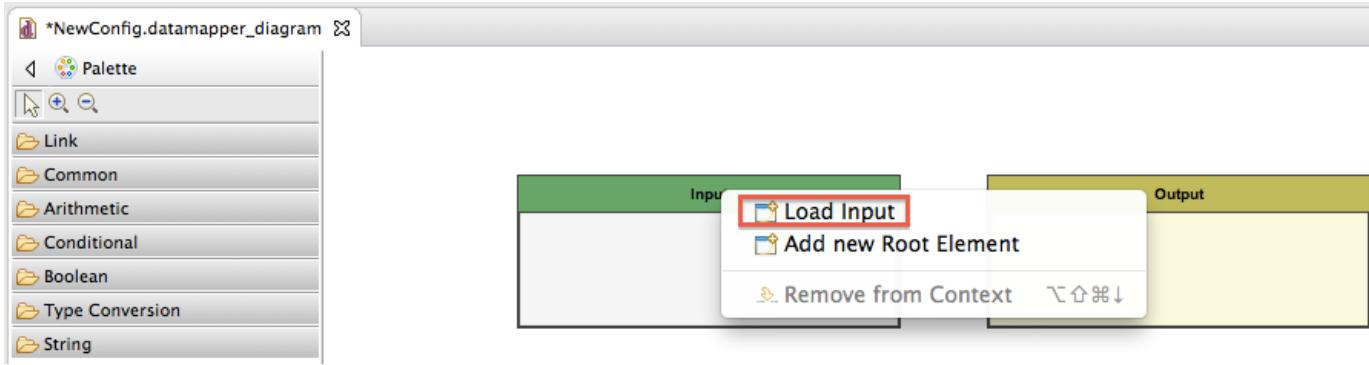


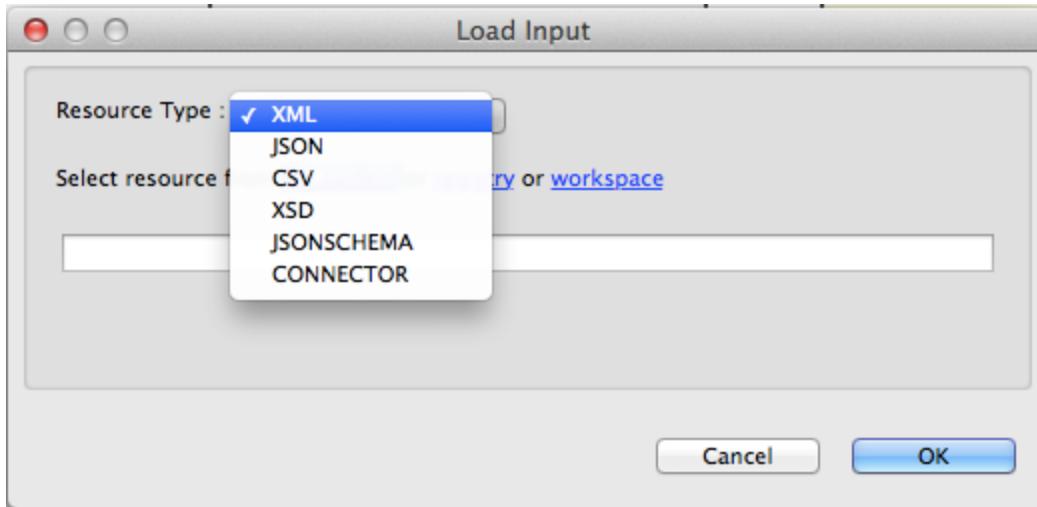
The .datamapper and .datamapper\_diagram files as shown in the example above contain meta data related to the Data Mapper diagram. They are ignored when you deploy the project to a server to be used by the Data Mapper Engine. Only the two schema files and the .dmc (Data Mapper Configuration) get deployed.

#### Input and output schema files

Input and output schema files are custom-defined JSON schemas that define the input/output format of input/output messages. The Data Mapper tool generates them when loading the input and output files as shown below.

You can also create the input and output JSON Schemas manually using the Data Mapper Diagram Editor. For instructions, see [Creating a JSON Schema Manually](#).





You can load the following input/output message formats:

When loading a sample input XML file, you cannot have the default namespace (i.e. without a prefix in the namespace element). Also, you need to use the same prefix in all occurrences that refer to the same namespace within one XML file. For example, see the use of the prefix axis2ns11 in the example below.

▼ Sample input XML file

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Header>
 <axis2ns11:LimitInfoHeader xmlns:axis2ns11="urn:partner.soap.sforce.com">
 <axis2ns11:limitInfo>
 <axis2ns11:current>42336</axis2ns11:current>
 <axis2ns11:limit>83000</axis2ns11:limit>
 <axis2ns11:type>API REQUESTS</axis2ns11:type>
 </axis2ns11:limitInfo>
 </axis2ns11:LimitInfoHeader>
</soapenv:Header>
<soapenv:Body>
 <axis2ns11:records xmlns:axis2ns11="urn:partner.soap.sforce.com"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="sf:sObject">
 <sf:type xmlns:sf="urn:sobject.partner.soap.sforce.com">Account</sf:type>
 <sf:Id
 xmlns:sf="urn:sobject.partner.soap.sforce.com">001E0000002SF02IA0</sf:Id>
 <sf:CreatedDate
 xmlns:sf="urn:sobject.partner.soap.sforce.com">2011-03-15T00:15:00.000Z</sf:CreatedDate>
 <sf:Id
 xmlns:sf="urn:sobject.partner.soap.sforce.com">001E0000002SF02IA0</sf:Id>
 <sf:Name xmlns:sf="urn:sobject.partner.soap.sforce.com">WSO2</sf:Name>
 </axis2ns11:records>
</soapenv:Body>
</soapenv:Envelope>
```

- **XML:** to load a sample XML file
- **JSON:** to load a sample JSON file
- **CSV:** to load a sample CSV file with column names as the first record
- **XSD:** to load an XSD file, which is an actual schema for XML files

- **JSONSCHEMA:** to load a WSO2 Data Mapper JSON schema
- **CONNECTOR:** to use Data Mapper with WSO2 ESB Connectors. Connectors will contain JSON schemas for each operation that defines the message formats to which it will respond and expect. Therefore, when you integrate connectors in a project this Connector option searches through the workspace and find the available Connectors. Then, you can select the respective Connector in the operation, so that the related JSON schema will be loaded for the Data Mapper by the tool.

Mapping configuration file

This is a JavaScript file generated by looking at the diagram you draw in the Data Mapper Diagram Editor by connecting input elements to output elements. Every operation you define in the diagram gets converted to a JavaScript operation.

### **Data Mapper Engine**

You need the following information to configure the Data Mapper Engine:

- Input message type
- Output message type
- Input schema Java Scripting API
- Output schema
- Mapping configuration

At the runtime, the Data Mapper Engine gets the input message and the runtime variable map object and outputs the transformed message. The Data Mapper Engine uses the Java Scripting API, to execute the mapping configuration. Therefore, if your runtime is JAVA 7, it uses the Rhino JS Engine and if your runtime is JAVA 8, it uses the Nashorn JS engine.

When you use JAVA 7, there are several limitations in the Rhino engine that directly affects the Data mapper Engine. There are several functions that Rhino does not support. For example, String object functions like `startsWith()` and `endsWith()`. Therefore, the Rhino engine may have limitations in executing those when using custom functions and operators.

Using product-specific runtime variables

Also, the Data Mapper engine allows you to use runtime product-specific variables in the mapping. The intermediate component should construct a map object containing runtime product-specific variables and send it to the Data Mapper Engine, thereby, when the mapping happens in the Data Mapper Engine, these variables become available.

For example, the Data Mapper mediator provides ESB properties like `axis2/transport/synapse/axis2client/operation/..` In the Data Mapper diagram, you can use the **Property operator** and define the scope and the property name and use it in the mapping. Then, the Data Mapper mediator will identify the required properties to execute the mapping and populate a map with the required properties and will send it to the Data Mapper Engine.

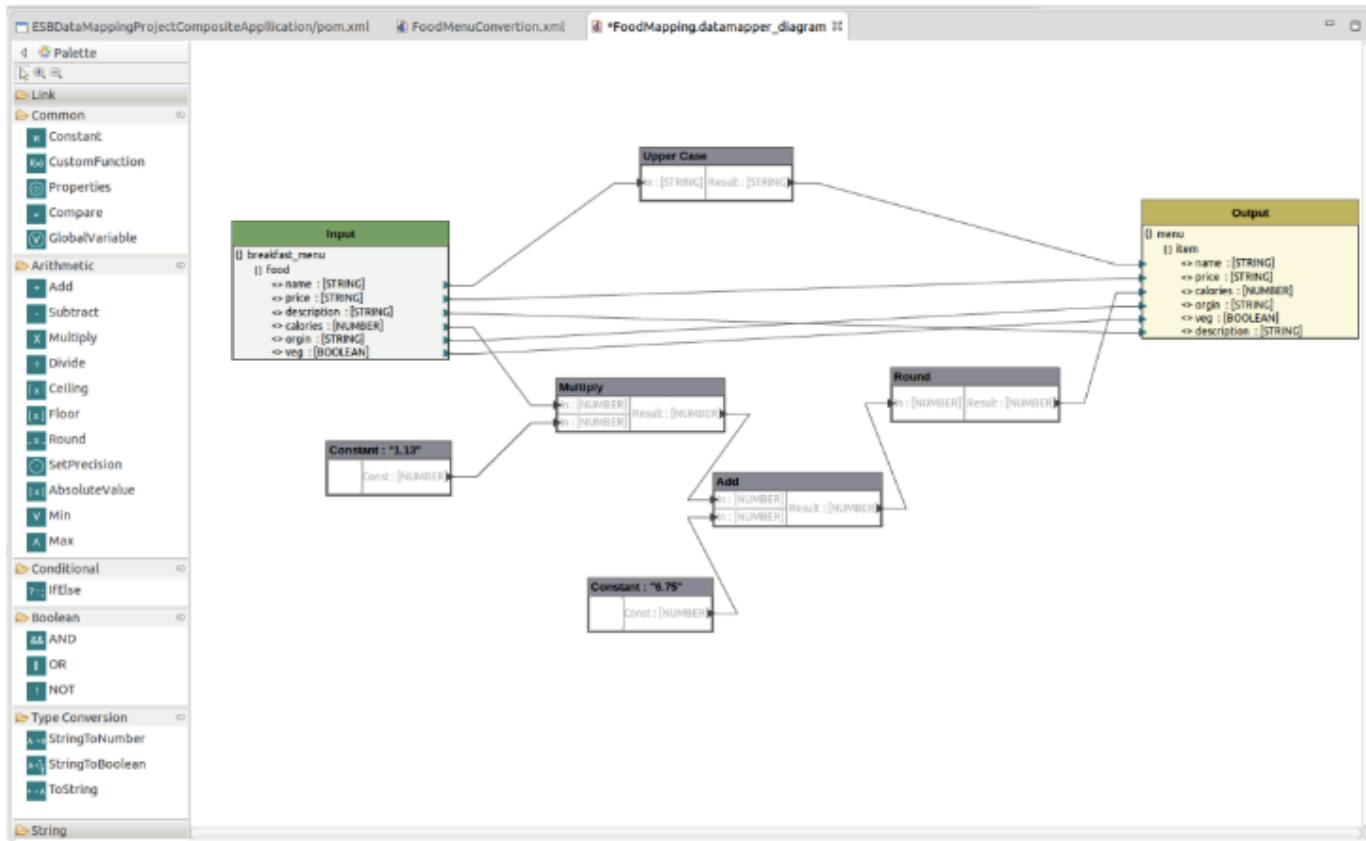
### **Data Mapper element and attribute types**

Following are the element and attribute types that are supported by the Data Mapper.

- {} - represents object elements
- [] - represents array elements
- <> - represents primitive field values
- A - represents XML attribute values

### **Data Mapper operations**

The operations palette placed in the left-hand side of the WSO2 Data Mapping Diagram Editor displays the operations that the Data Mapper supports as shown below.



You can drag and drop these operations to the Editor area. There are six categories of operations as follows:

- Links
- Common
- Arithmetic
- Conditional
- Boolean
- Type Conversion
- String

## Links



**Data Mapping Link:** maps elements with other operators and elements.

## Common



**Constant:** defines String, number or boolean constant values.



**Custom Function:** defines custom functions to use in the mapping.



**Properties:** uses product-specific runtime variables.



**Global Variable:** instantiates global variables that you can access from anywhere.

**Compare:** compares two inputs in the mapping.

### *Arithmetic*

**Add:** adds two numbers.

**Subtract:** subtracts two or more numbers.

**Multiply:** multiplies two or more numbers.

**Divide:** divides two numbers.

**Ceiling:** derives the ceiling value of a number (closest larger integer value).

**Floor:** derives the floor value of a number (closest lower integer value).

**Round:** derives the nearest integer value.

**Set Precision:** formats a number into a specified length.

**Absolute Value:** derives the absolute value of a rational number.

**Min:** derives the minimum number from given inputs

**Max:** derives the maximum number from given inputs

### *Conditional*

**IfElse:** uses a condition and selects one input from given two.

### *Boolean*

**AND:** performs the boolean AND operation on inputs.

**OR:** performs the boolean OR operation on inputs.

**NOT:** performs the boolean NOT operation on inputs.

### *Type conversion*

 A → 0

**StringToNumber:** converts a String value to number (“0” -> 0).

 A → t

**StringToBoolean:** converts a String value to boolean (“true” -> true).

 \* → A

**ToString:** converts a number or a boolean value to String.

## String

 □ →

**Concat:** concatenates two or more Strings.

 ↗ ↘

**Split:** splits a String by a matching String value.

 A ↑

**Uppercase:** converts a String to uppercase letters.

 a ↓

**Lowercase:** converts a String to lowercase letters.

 X ↴

**String Length:** gets the length of the String.

 X ...

**StratsWith:** checks whether a String starts with a specific value. (This is not supported in Java 7.)

 ... X

**EndsWith:** checks whether String ends with a specific value. (This is not supported in Java 7.)

 x ☐

**Substring:** extracts a part of the String value.

 → x ←

**Trim:** removes white spaces from the beginning and end of a String.

 a ↩ b

**Replace:** replaces the first occurrence of a target String with another.

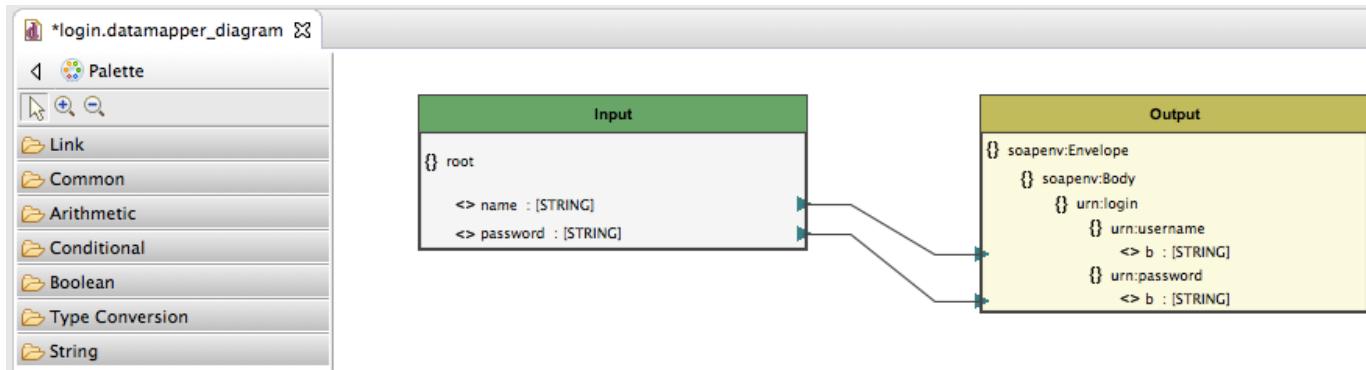
 x = x

**Match** – check whether the input match with a (JS) Regular Expression

## Examples

### Example 1 - Creating a SOAP payload with namespaces

This example creates a Salesforce login SOAP payload using a JSON payload. The login payload consists of XML namespaces. Even though the JSON payload does not contain any namespace information, the output JSON schema will be generated with XML namespace information using the provided SOAP payload.



The sample input JSON payload is as follows.

```
{
 "name": "Watson",
 "password": "watson@123"
}
```

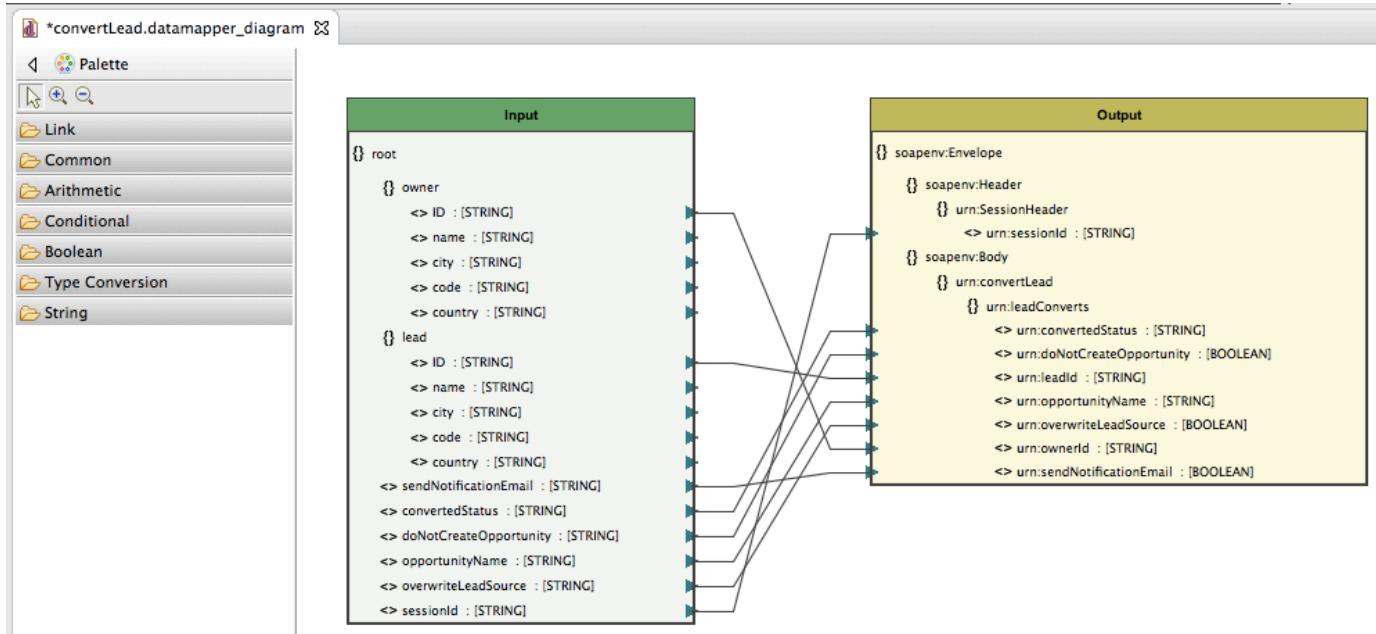
The sample output XML is as follows.

```
<soapenv:Envelope xmlns:urn="urn:enterprise.soap.sforce.com"
 xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
 <soapenv:Body>
 <urn:login>
 <urn:username>user@domain.com</urn:username>
 <urn:password>secret</urn:password>
 </urn:login>
 </soapenv:Body>
</soapenv:Envelope>
```

### **Example 2 - Mapping SOAP header elements**

This example demonstrates how to map SOAP header elements along with SOAP body elements to create a certain SOAP payload, by creating a Salesforce convertLead SOAP payload using a JSON payload. The Convert Lead SOAP payload needs mapping SOAP header information.

E.g. <urn:sessionId>QwWsHJyTPW.1pd0\_jXlNKOSU</urn:sessionId>



The sample input JSON payload is as follows.

```
{
 "owner": {
 "ID": "005D000000nVYVIA2",
 "name": "Smith",
 "city": "CA",
 "code": "94041",
 "country": "US"
 },
 "lead": {
 "ID": "00QD000000FP14JMAT",
 "name": "Carl",
 "city": "NC",
 "code": "97788",
 "country": "US"
 },
 "sendNotificationEmail": "true",
 "convertedStatus": "Qualified",
 "doNotCreateOpportunity": "true",
 "opportunityName": "Partner Opportunity",
 "overwriteLeadSource": "true",
 "sessionId": "QwWshJyTPW.1pd0_jXlnKOSU"
}
```

The sample output XML is as follows.

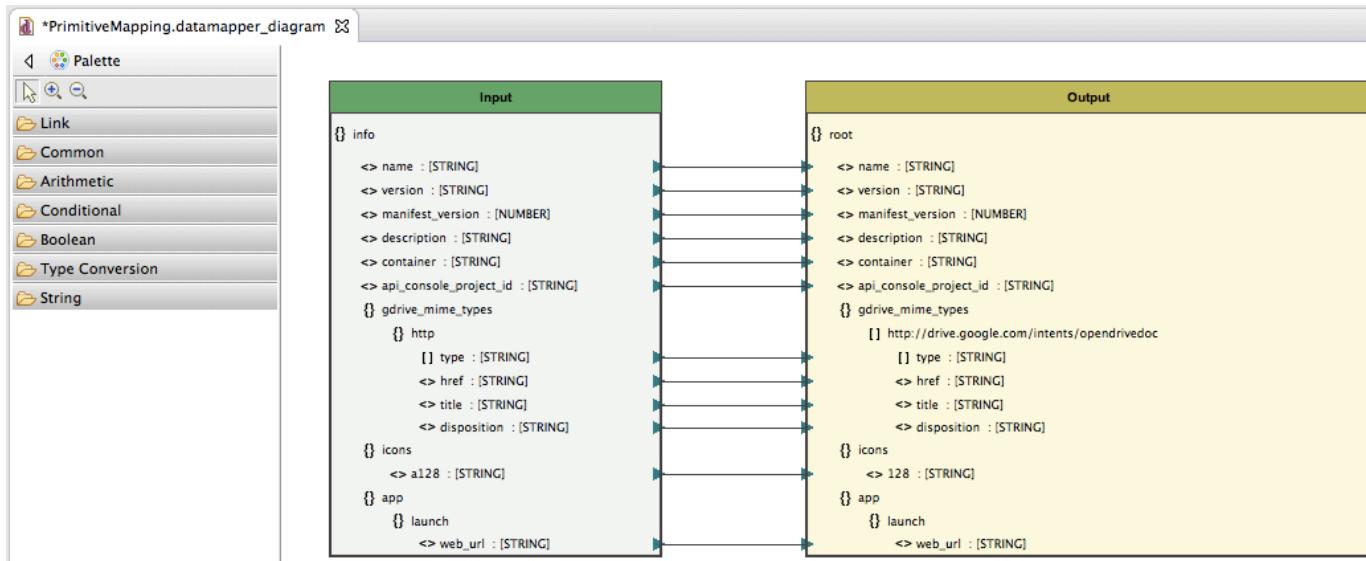
```

<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:urn="urn:enterprise.soap.sforce.com"
xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope/">
<soapenv:Header>
<urn:SessionHeader>
<urn:sessionId>QwWsHJyTPW.1pd0_jXlNKOSU</urn:sessionId>
</urn:SessionHeader>
</soapenv:Header>
<soapenv:Body>
<urn:convertLead >
<urn:leadConverts> <!-- Zero or more repetitions -->
<urn:convertedStatus>Qualified</urn:convertedStatus>
<urn:doNotCreateOpportunity>false</urn:doNotCreateOpportunity>
<urn:leadId>00QD000000FP14JM</urn:leadId>
<urn:opportunityName>Partner Opportunity</urn:opportunityName>
<urn:overwriteLeadSource>true</urn:overwriteLeadSource>
<urn:ownerId>005D000000nVYVIA2</urn:ownerId>
<urn:sendNotificationEmail>true</urn:sendNotificationEmail>
</urn:leadConverts>
</urn:convertLead>
</soapenv:Body>
</soapenv:Envelope>

```

### Example 3 - Mapping primitive types

This example demonstrates how you can map an XML payload with integer, boolean etc. values, into a JSON payload with required primitive types, by specifying the required primitive type in the JSON schema.



The sample input XML payload is as follows.

```
<?xml version="1.0" encoding="UTF-8" ?>
<name>app_name</name>
<version>version</version>
<manifest_version>2</manifest_version>
<description>description_text</description>
<container>GOOGLE_DRIVE</container>
<api_console_project_id>YOUR_APP_ID</api_console_project_id>
<gdrive_mime_types>
 <http://drive.google.com/intents/opendrivedoc>
 <type>image/png</type>
 <type>image/jpeg</type>
 <type>image/gif</type>
 <type>application/vnd.google.drive.ext-type.png</type>
 <type>application/vnd.google.drive.ext-type.jpg</type>
 <type>application/vnd.google.drive.ext-type.gif</type>
 <href>http://your_web_url/</href>
 <title>Open</title>
 <disposition>window</disposition>
</http://drive.google.com/intents/opendrivedoc>
</gdrive_mime_types>
<icons>
 <128>icon_128.png</128>
</icons>
<app>
 <launch>
 <web_url>http://yoursite.com</web_url>
 </launch>
</app>
```

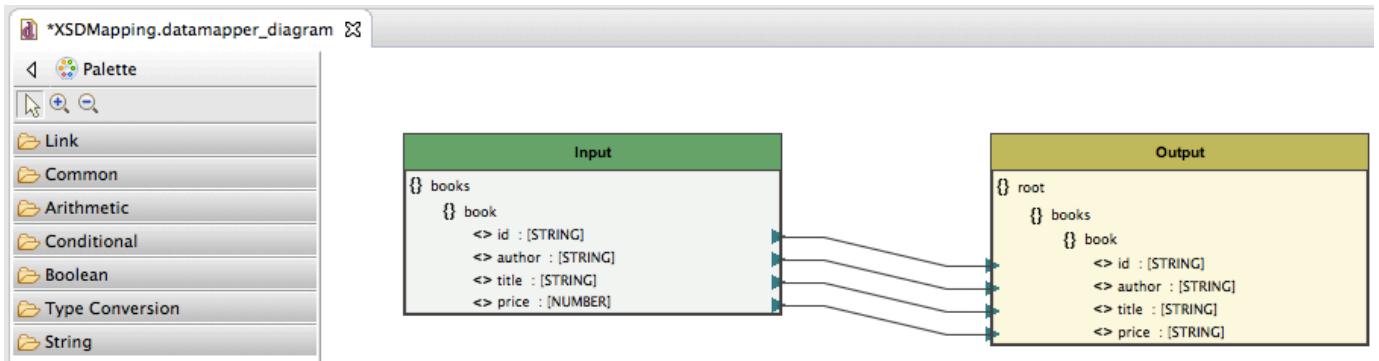
The sample output JSON is as follows.

```
{
 "name" : "app_name",
 "version" : "version",
 "manifest_version" : 2,
 "description" : "description_text",
 "container" : "GOOGLE_DRIVE",
 "api_console_project_id" : "YOUR_APP_ID",
 "gdrive_mime_types": {
 "http://drive.google.com/intents/opendrivedoc": [
 {
 "type": ["image/png", "image/jpeg", "image/gif",
"application/vnd.google.drive.ext-type.png",

"application/vnd.google.drive.ext-type.jpg", "application/vnd.google.drive.ext-type.gif
"] ,
 "href": "http://your_web_url/",
 "title" : "Open",
 "disposition" : "window"
 }
]
 },
 "icons": {
 "128": "icon_128.png"
 },
 "app" : {
 "launch" : {
 "web_url" : "http://yoursite.com"
 }
 }
}
```

#### **Example 4 - Mapping XML to CSV**

This example demonstrates how you can map an XML payload to CSV format.



The sample input XML payload is as follows.

```

<?xml version="1.0"?>
<PurchaseOrder PurchaseOrderNumber="001">
<Address>
 <Name>James Yee</Name>
 <Street>Downtown Bartow</Street>
 <City>Old Town</City>
 <State>PA</State>
 <Zip>95819</Zip>
 <Country>USA</Country>
</Address>
<Address>
 <Name>Elen Smith</Name>
 <Street>123 Maple Street</Street>
 <City>Mill Valley</City>
 <State>CA</State>
 <Zip>10999</Zip>
 <Country>USA</Country>
</Address>
<DeliveryNotes>Please leave packages in shed by driveway.</DeliveryNotes>
</PurchaseOrder>

```

The sample output CSV is as follows.

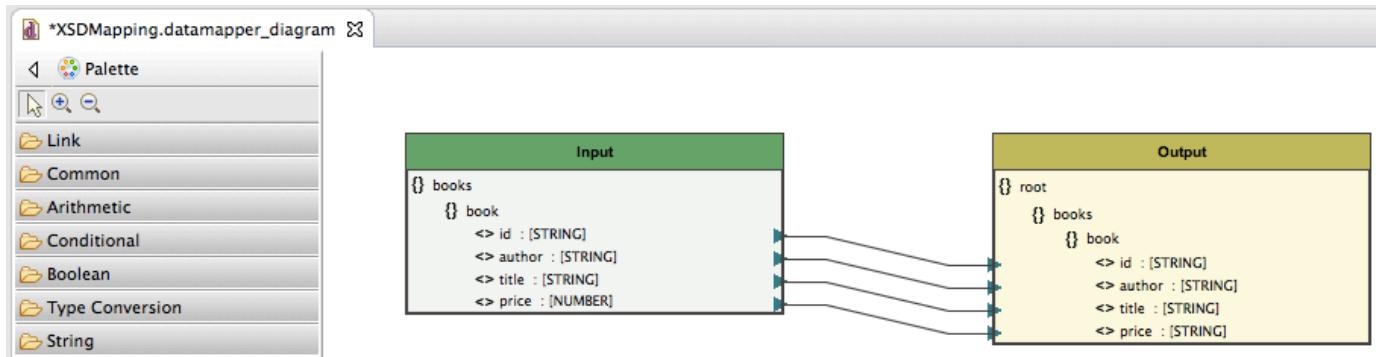
```

Name,Street,City,State,Zip,Country
James Yee,Downtown Bartow,Old Town,PA,95819,USA
Ellen Smith,123 Maple Street,Mill Valley,CA,10999,USA

```

### Example 5 - Mapping XSD to JSON

This example demonstrates how you can map an XSD payload to JSON format.



The sample input XSD payload is as follows.

```

<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="books">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="book">
 <xs:complexType>
 <xs:sequence>
 <xs:element type="xs:string" name="id"/>
 <xs:element type="xs:string" name="author"/>
 <xs:element type="xs:string" name="title"/>
 <xs:element type="xs:float" name="price"/>
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 </xs:sequence>
 </xs:complexType>
 </xs:element>
</xs:schema>

```

The sample output JSON is as follows.

```
{
 "books": {
 "book": {
 "id": "001",
 "author": "Writer",
 "title": "Great book on nature",
 "price": "44.95"
 }
 }
}
```

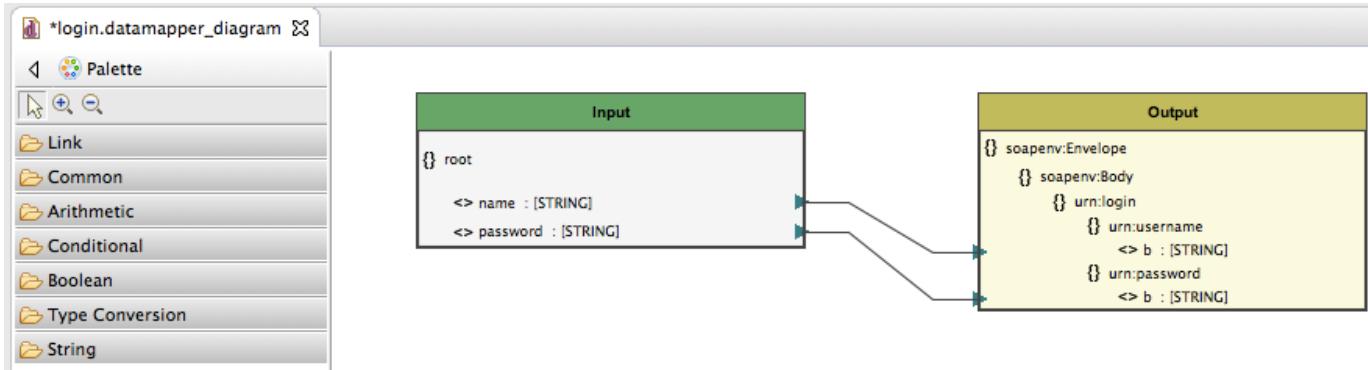
## Samples

For a sample that demonstrates how to use the Data Mapper mediator, see:

- [Using Data Mapper Mediator in WSO2 ESB](#)

### Creating a JSON Schema Manually

You need to have two input and output JSON Schema files for a [Data Mapping configuration](#) as shown in the example below.



Therefore, you can use the WSO2 Data Mapper Diagram Editor to create a JSON Schema manually by adding elements to the Data Mapper tree view as explained below.

- Components of a JSON Schema
- Adding the root element
- Adding a child element
- Setting a nullable element
- Editing an element
- Deleting an element

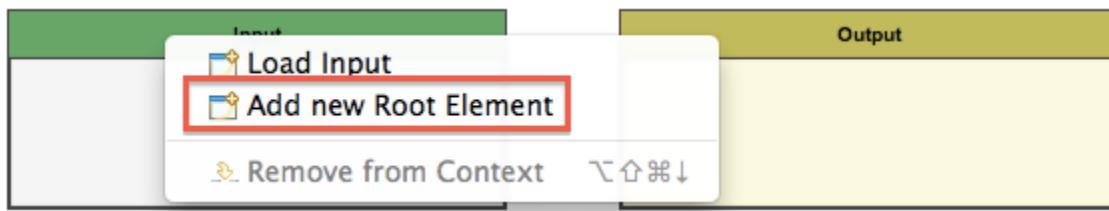
### **Components of a JSON Schema**

There are the following four types of components in a JSON Schema:

- Arrays
- Objects
- Fields
- Attributes

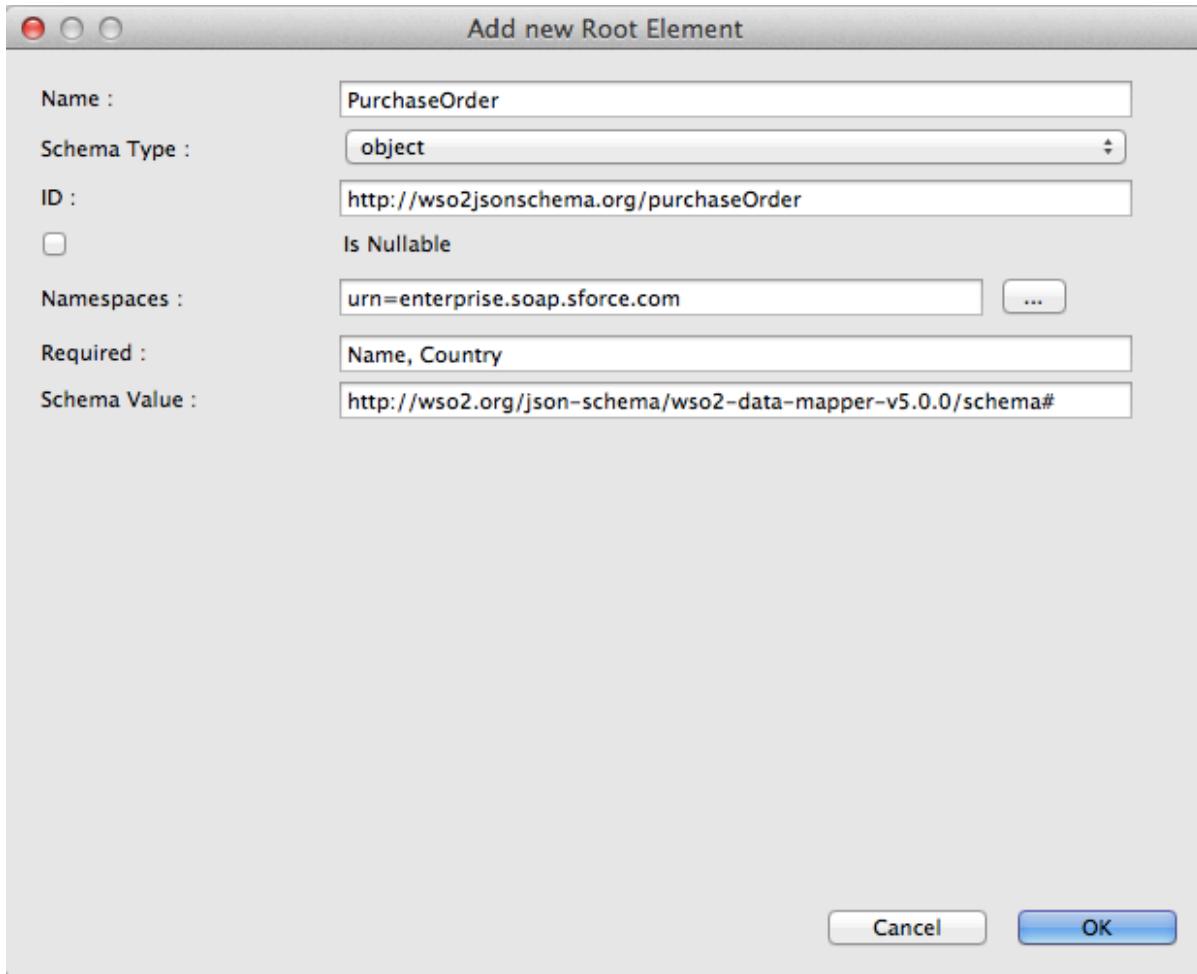
### **Adding the root element**

When creating the tree you need to first add the root element. The root element can be either an object or an array. Right-click on the **Input** or **Output** box and then click **Add new Root Element** as shown below, to add the root element.



Add the following details to create the root element.

- **Name:** name of the root element
- **Schema Type:** type of the element (i.e. array or object)
- **ID:** ID of the root element to uniquely identify it
- **isNullable:** whether the element can be a nullable (i.e. not available in the payload) (optional)
- **Namespaces:** prefix and URL of the namespace (optional)
- **Required:** child elements required to be there in the payload (optional)
- **Schema Value:** custom URI of the Schema



You view the root element added to the **Input** box as shown below.

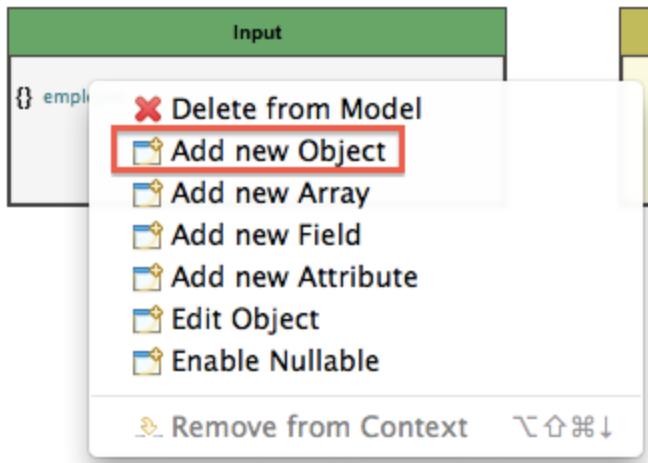
Input
{ } urn:PurchaseOrder

### ***Adding a child element***

You can add Arrays, Objects, Fields and Attributes as child elements as explained below.

[Adding an Object as a Child element](#)
[Adding an Array as a child element](#)
[Adding a Field as a child element](#)  
[Adding an Attribute as a child element](#)

Right-click on the parent element and click **Add new Object**, to add an Object as shown below.



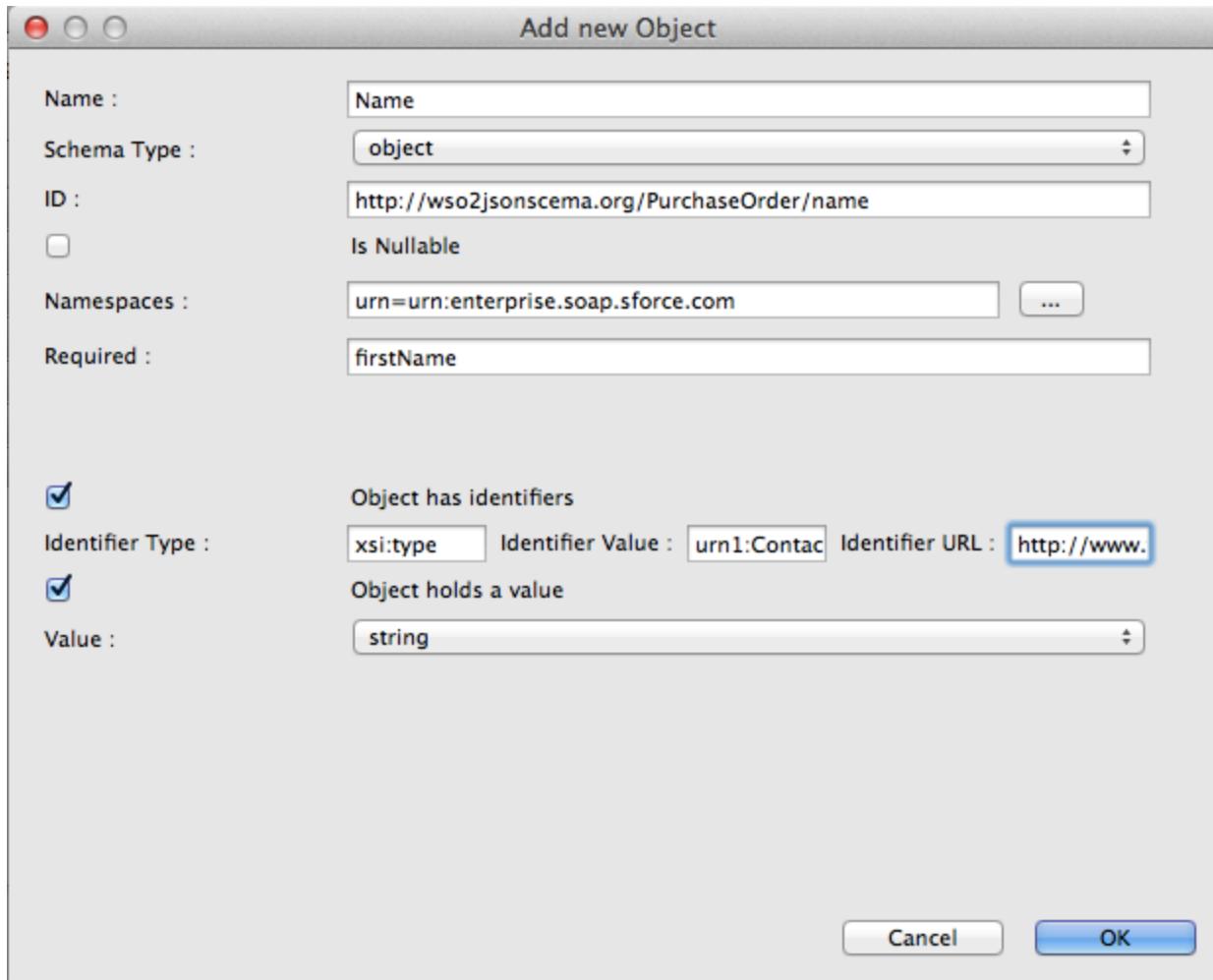
Add the following details to create an Object as a child element.

- **Name:** name of the child element
- **Schema Type:** type of the element (i.e. object)
- **ID:** ID of the child element to uniquely identify it
- **isNullable:** whether the element can be a nullable (i.e. not available in the payload) (optional)
- **Namespaces:** prefix and URL of the namespace (optional)
- **Required:** child elements required to be there in the payload (optional)
- **Schema Value:** custom URI of the Schema

If the object has element identifiers, then select the checkbox and add the value, type, and URL of the identifier.

- **Object holds a value:** if the object holds a value or not

If the object holds a value, then select the checkbox and select the data type.

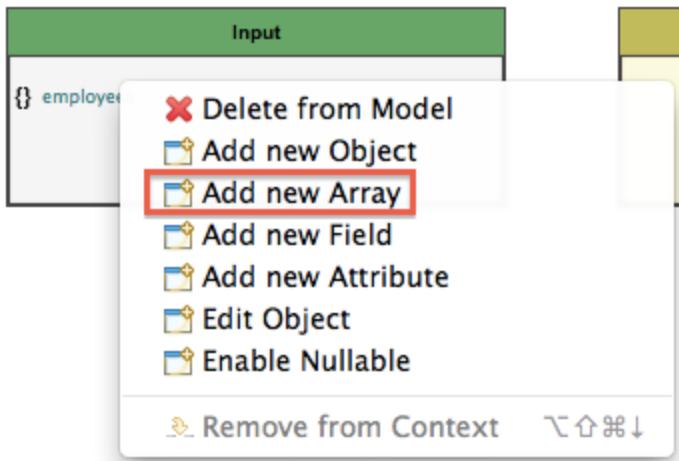


You view the child Object element added to the root element as shown below.

The element identifier is added as an attribute to the element.

Input
<pre>{ } urn:PurchaseOrder   { } Name,xsi:type=urn1&gt;Contact : [STRING]     A xsi:type : [STRING]</pre>

Right-click on the parent element and click **Add new Array**, to add an Array as shown below.



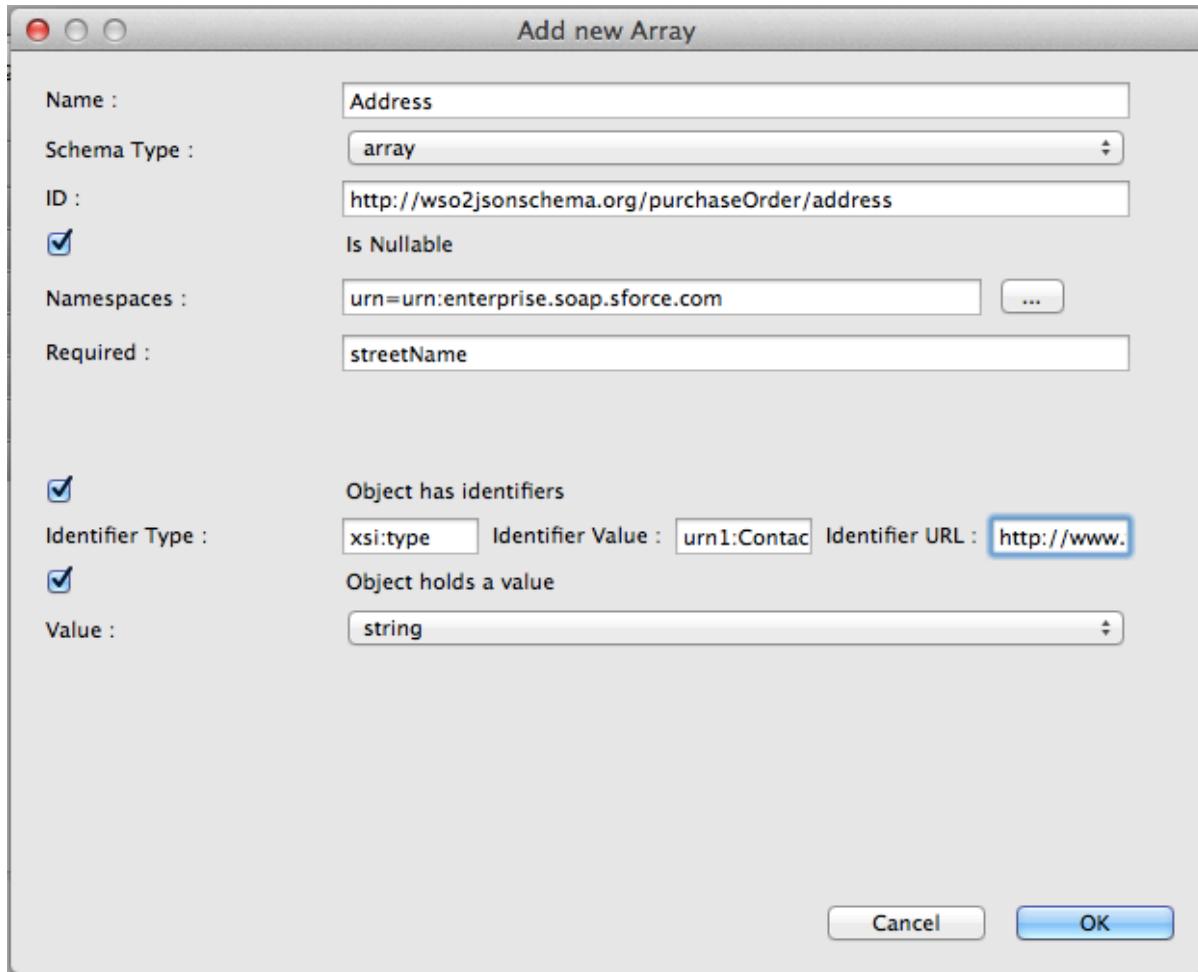
You need to add the following details to create an Array as a child element.

- **Name:** name of the child element
- **Schema Type:** type of the element (i.e. array)
- **ID:** ID of the child element to uniquely identify it
- **isNullable:** whether the element can be a nullable (i.e. not available in the payload) (optional)  
**Namespaces:** prefix and URL of the namespace (optional)
- **Required:** child elements required to be there in the payload (optional)
- **Schema Value:** custom URI of the Schema

If the array has element identifiers, then select the checkbox and add the value, type, and URL of the identifier.

- **Object holds a value:** if the object holds a value or not

If the array holds a value, then select the checkbox and select the data type.

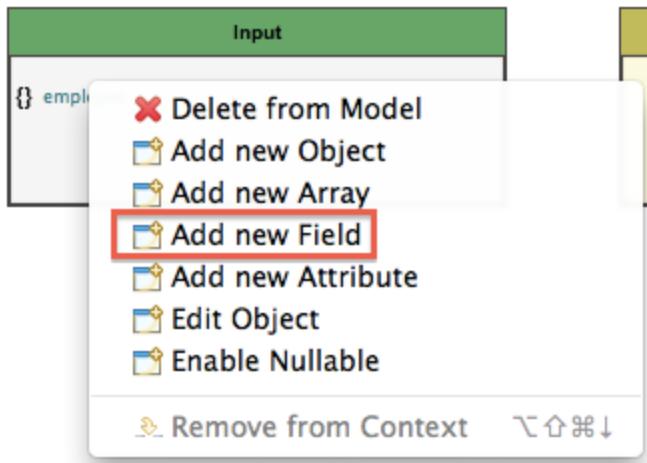


You view the child Array element added to the root element as shown below.

The element identifier is added as an attribute to the element.

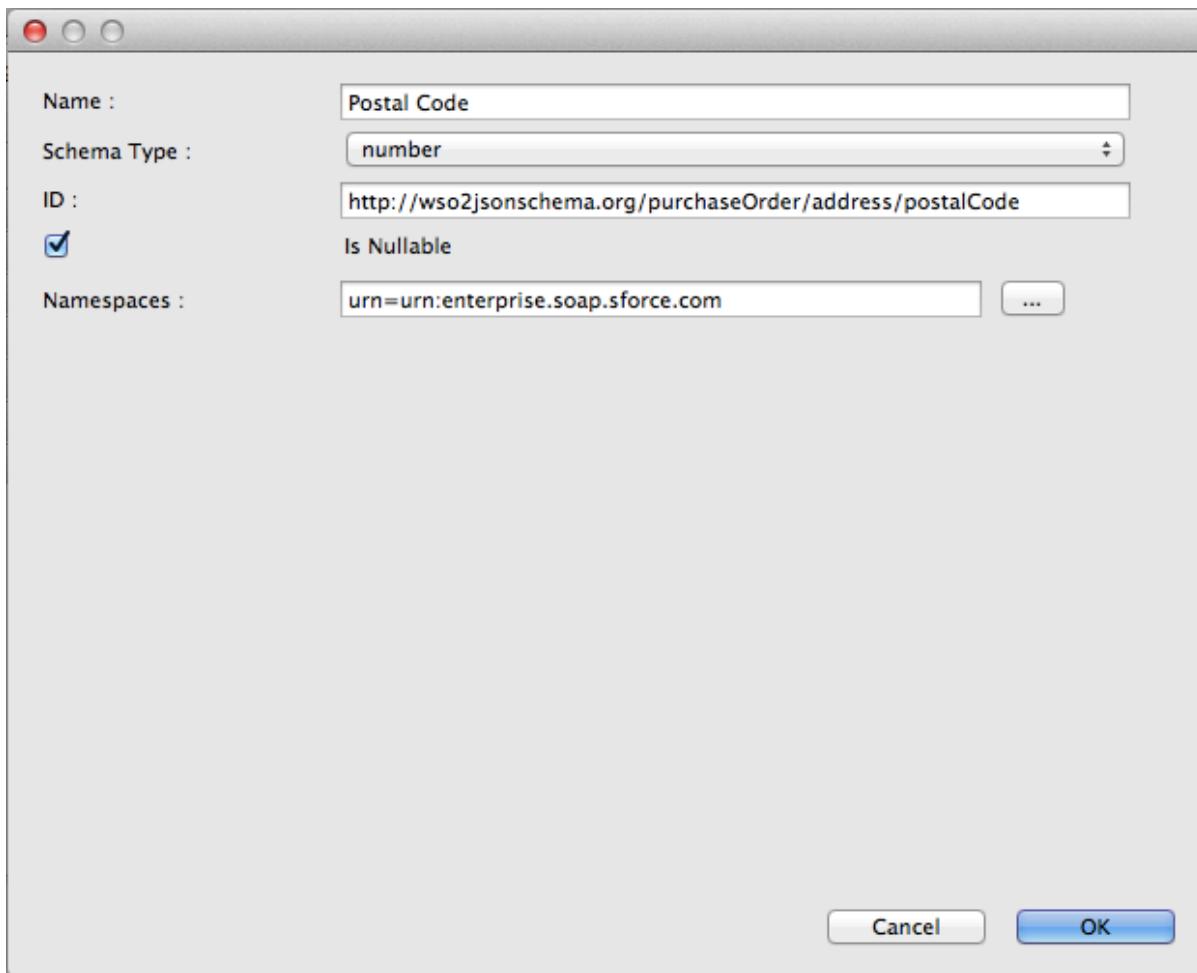
Input	
{}	urn:PurchaseOrder
	{}
	Name,xsi:type=urn1>Contact : [STRING]
	A xsi:type : [STRING]
	[1] Address,xsi:type=urn1>Contact : [STRING]
	A xsi:type : [STRING]

Right-click on the parent element and click **Add new Field**, to add a Field as shown below.

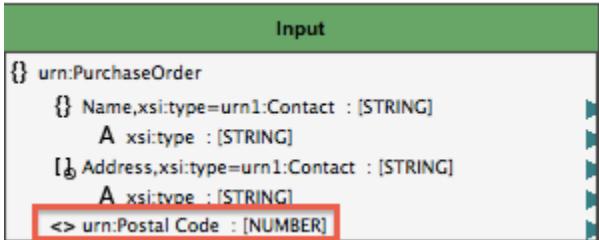


You need to add the following details to create a Field as a child element.

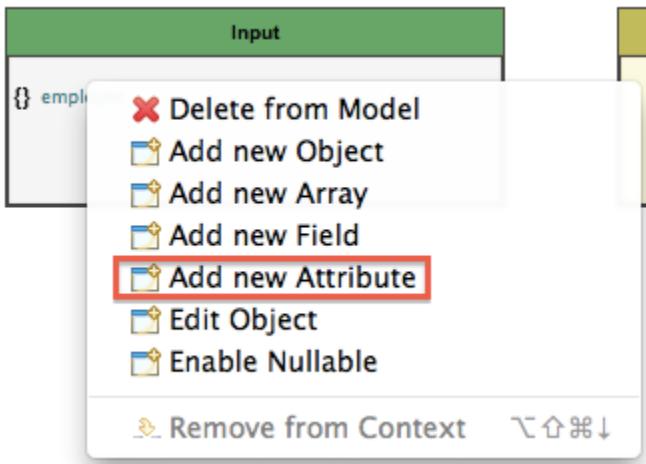
- Name:** name of the child element
- Schema Type:** type of the element
- ID:** ID of the child element to uniquely identify it
- isNullable:** whether the element can be a nullable (i.e. not available in the payload) (optional)
- Namespaces:** prefix and URL of the namespace (optional)
- Required:** child elements required to be there in the payload (optional)
- Schema Value:** custom URI of the Schema



You view the child Field element added to the root element as shown below.

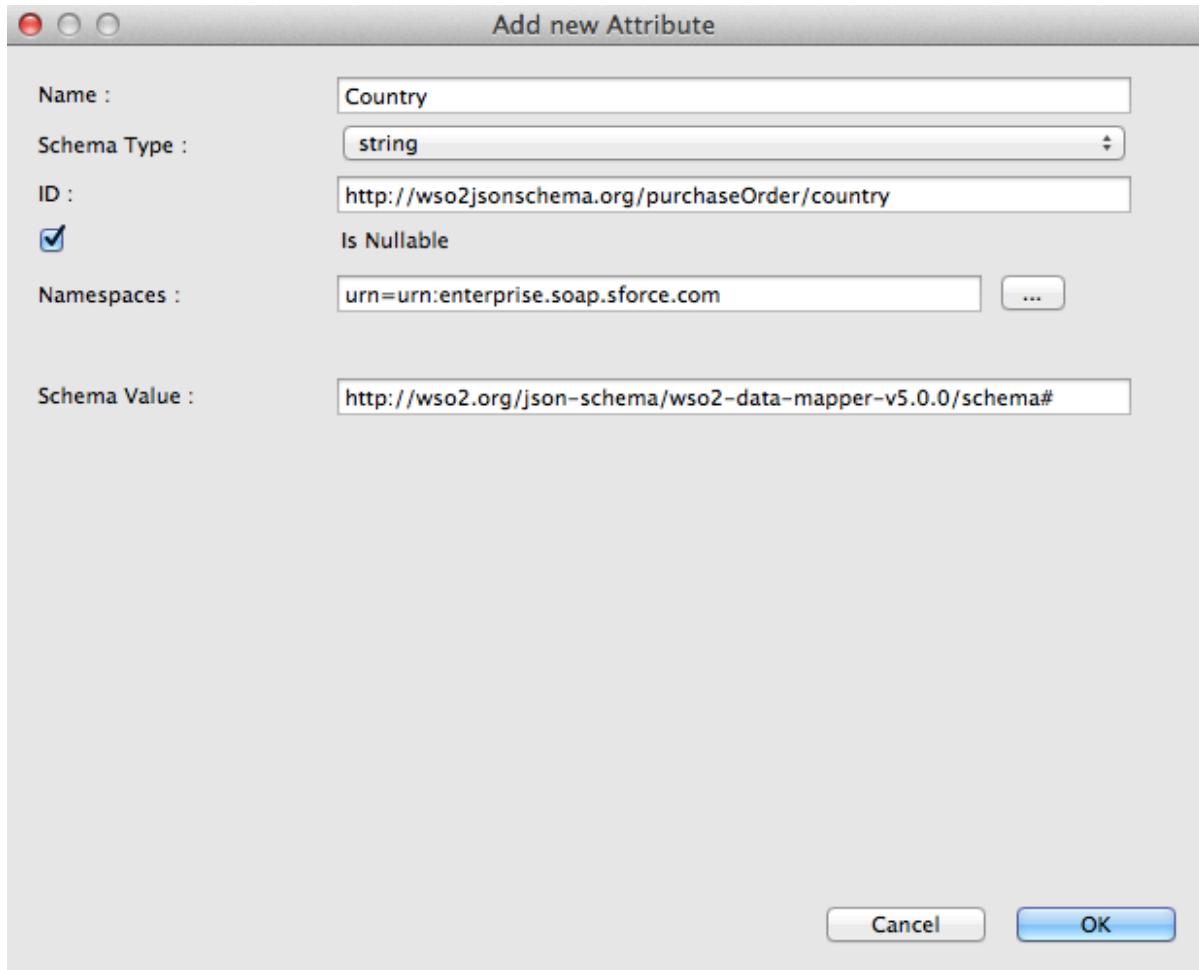


Right-click on the parent element and click **Add new Attribute**, to add an attribute as shown below.



You need to add the following details to create an Attribute as a child element.

- **Name:** name of the child element
- **Schema Type:** type of the element
- **ID:** ID of the child element to uniquely identify it
- **isNullable:** whether the element can be a nullable (i.e. not available in the payload (optional))
- **Namespaces:** prefix and URL of the namespace (optional)
- **Required:** child elements required to be there in the payload (optional)
- **Schema Value:** custom URI of the Schema

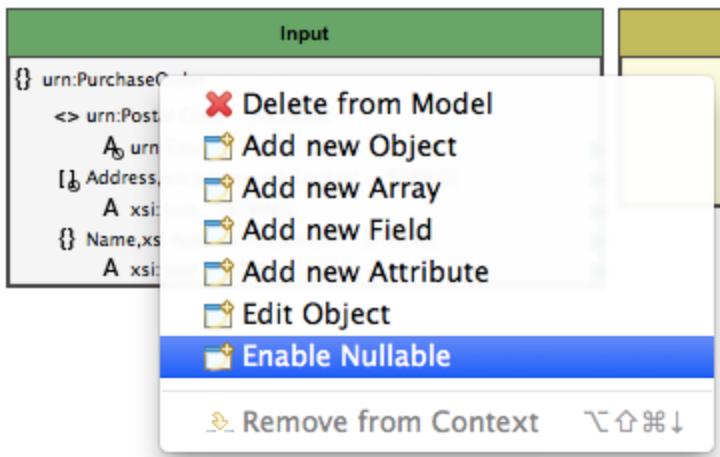


You view the child Attribute element added to the root element as shown below.

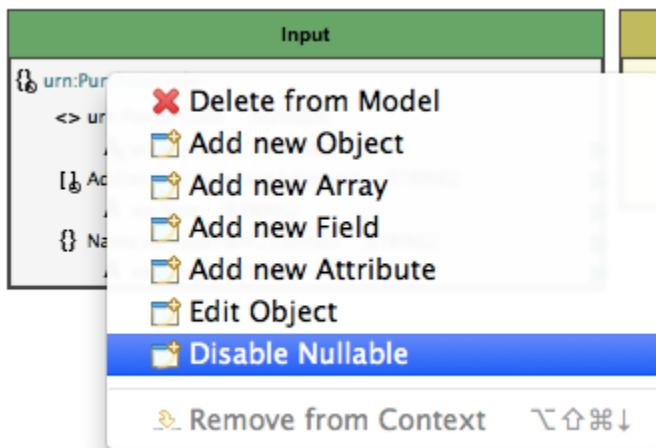
```
Input
{} urn:PurchaseOrder
 <> urn:Postal Code : [NUMBER]
 A urn:Country : [STRING]
 [1] Address,xsi:type=urn1>Contact : [STRING]
 A xsi:type : [STRING]
 {} Name,xsi:type=urn1>Contact : [STRING]
 A xsi:type : [STRING]
```

## ***Setting a nullable element***

You can set an element as a nullable element, so that it is not required to have that element in the payload. Right-click on the parent element and click **Enable Nullable**, to enable an element to make it nullable as shown below.

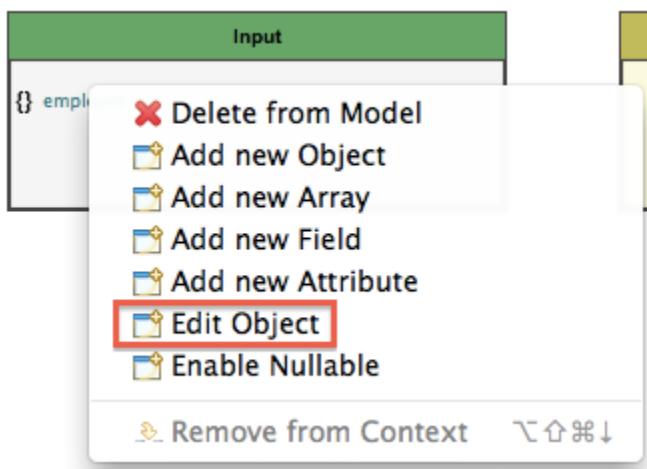


Right-click on the parent element and click **Disable Nullable**, to make it required in the payload as shown below.



### **Editing an element**

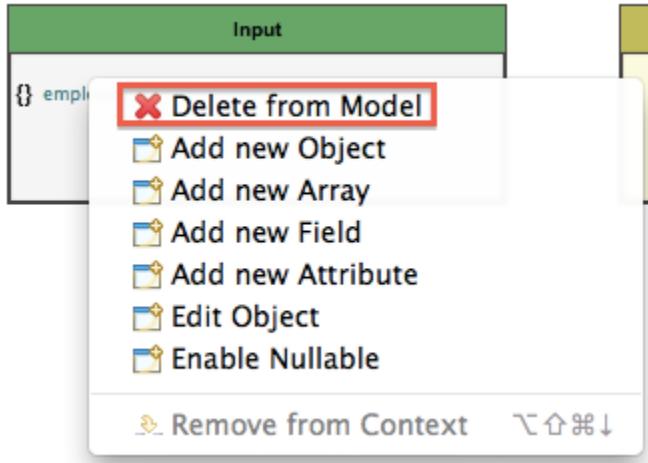
Right-click on any element and click **Edit Object**, to edit and update the field values as required as shown below.



### **Deleting an element**

Right-click on any element and click **Delete from Model**, to delete it as shown below.

This deletes the element with its child nodes.



## WSO2 ESB Data Mapper JSON Schema Specification

The following specification defines the Data Mapper JSON schema of WSO2 ESB. It is intended to be the authoritative specification. Implementations of schemas for the Data Mapper mediator must adhere to this.

- Schema declaration
- Primitive types
- Complex types
- Defining WSO2 schemas to represent an XML payload

### **Schema declaration**

A schema is represented in JSON by one of:

- A JSON string, naming a defined type.
- A JSON object, of the form: { "type" : "typeName" ... attributes... }, where typeName is either a primitive or a derived type name, as defined below.
- A JSON array, representing a union of embedded types.

A WSO2 ESB Data Mapper schema should start with the \$schema attribute with the WSO2 Data Mapper schema version. For example: { "\$schema": "http://wso2-data-mapper-json-schema/1.0v" }

Also, it can contain following optional attributes that will define more information about the schema.

- “**id**”: a JSON string declaring a unique identifier for the schema.
- “**title**”: a JSON string defining the root element name.
- “**description**”: a JSON string providing a detailed description about the schema.
- “**type**”: a JSON string providing the element type.
- “**namespaces**”: a JSON array of JSON objects defining namespaces and prefix values used in the schema as shown in the following example.

```
{
 "$schema" : "http://wso2-data-mapper-json-schema/1.0v",
 "id": "http://wso2-data-mapper-json-schema-sample-o1",
 "title": "RootElement",
 "type": "object",
 "description": "This schema represent any form of object without any restriction",
 "namespaces": [
 { "prefix": "ns1", "url": "http://ns1.com" },
 { "prefix": "ns2", "url": "http://ns2.com" }
]
}
```

## Primitive types

Primitive types have no specified attributes. The set of primitive type names are as follows.

- **null**: no value
- **boolean**: a binary value
- **integer**: integer value
- **number**: rational numbers
- **string**: unicode character sequence

Primitive type names are also defined type names. Thus, for example, the schema "string" is equivalent to: { "type": "string" }

## Complex types

WSO2 ESB Data Mapper schema supports the following complex types: object and array.  
Object

Object uses the type name "object", and supports the following attributes.

- **"id"** : a JSON string declaring a unique identifier for the object (required).
- **"type"**: a JSON string providing the element type.
- **"description"**: a JSON string providing documentation to the user of this schema.
- **"properties"**: a JSON object listing fields (required). Each field is a JSON object.
- **"attributes"**: a JSON object listing XML attribute fields. Each field is a JSON object.

### Arrays

Arrays use the type name "array", and support a single attribute out of the following.

- **"items"**: the schema representing the items of the array.
- **"id"** : a JSON string declaring a unique identifier for the object (required).
- **"attributes"**: a JSON object listing XML attribute fields. Each field is a JSON object.
- **"description"**: a JSON string providing documentation to the user of this schema

For example, an array of an object containing a field named `firstname` is declared as shown below.

```
{
 "type": "array",
 "items": [
 {
 "id": "http://jsonschema.net/employee/0",
 "type": "object",
 "properties": {
 "firstname": {
 "id": "http://jsonschema.net/employee/0/firstname",
 "type": "string"
 }
 }
 }]
 }
}
```

### ***Defining WSO2 schemas to represent an XML payload***

There are differences between XML and JSON message specifications. Therefore, to represent XML message formats in JSON schemas, you need to introduce a few more configurations as explained below.  
Representing XML attributes and namespaces in WSO2 JSON schemas

For example, you can build a JSON schema, which follows the WSO2 specification using the following XML code as described below.

```

<?xml version="1.0" encoding="UTF-8"?>
<ns:employees xmlns:ns="http://wso2.employee.info"
 xmlns:sn="http://wso2.employee.address">
 <ns:employee>
 <ns:firstname>Mark</ns:firstname>
 <ns:lastname>Taylor</ns:lastname>
 <sn:addresses>
 <sn:address location="home">
 <sn:city postalcode="30000">LA</sn:city>
 <sn:road>baker street</sn:road>
 </sn:address>
 <sn:address location="office">
 <sn:city postalcode="10003">Colombo 03</sn:city>
 <sn:road>duplication road</sn:road>
 </sn:address>
 </sn:addresses>
 </ns:employee>
 <ns:employee>
 <ns:firstname>Mathew</ns:firstname>
 <ns:lastname>Hayden</ns:lastname>
 <sn:addresses>
 <sn:address location="home">
 <sn:city postalcode="60000">Sydney</sn:city>
 <sn:road>101 street</sn:road>
 </sn:address>
 <sn:address location="office">
 <sn:city postalcode="10003">Colombo 03</sn:city>
 <sn:road>duplication road</sn:road>
 </sn:address>
 </sn:addresses>
 </ns:employee>
</ns:employees>

```

WSO2 Data Mapper supports only single rooted XML messages. In the above example, `employees` is the root element of the payload, and it should be the value of the `title` element.

Also, there are two namespace values used. Those should be listed under the `namespaces` field with any prefix value.

Prefix value can be any valid string that contains only [a-z,A-Z,0-1] characters. You need not match them with the prefix values of the sample.

When you include above information, the schema will be as follows.

The "required" field specifies the fields that are mandatory to contain in that level of schema.

```
{
 "$schema" : "http://wso2-data-mapper-json-schema/1.0v",
 "id": "http://wso2-data-mapper-json-schema-sample-o1",
 "title": "employees",
 "type": "object",
 "description": "This schema represent wso2 employee xml message format",
 "required": [
 "employees"
],
 "namespaces": [
 { "prefix": "ns1", "url": "http://wso2.employee.info" },
 { "prefix": "ns2", "url": "http://wso2.employee.address" }
]
}
```

Including the child elements and attribute values

Define child elements under the “`properties`” field as a JSON object with fields to describe the child element. In the above employee example, the `employees` element contains an array of similar employee elements. Hence, this can be represented as the following schema.

```
{
 "$schema" : "http://wso2-data-mapper-json-schema/1.0v",
 "id": "http://wso2-data-mapper-json-schema-sample-employees",
 "title": "employees",
 "type": "object",
 "description": "This schema represent wso2 employee xml message format",
 "properties": {
 "employee": {
 "id": "http://wso2-data-mapper-json-schema-sample-employees/employee",
 "type": "array",
 "Items": [
 {
 "required": ["arrayRequired"]
 }
],
 "required": [
 "employees"
],
 "namespaces": [
 { "prefix": "ns1", "url": "http://wso2.employee.info" },
 { "prefix": "ns2", "url": "http://wso2.employee.address" }
]
 }
 }
}
```

Since the `employee` element is an array type element, it contains a field named “`items`”, which defines the element format of the array of employee elements. It contains three child fields as `firstname`, `lastname`, and `address` with string, string, and object types accordingly. Hence, when you include these elements into the schema, it will look as the following schema.

```
{
 "$schema" : "http://wso2-data-mapper-json-schema/1.0v",
 "id": "http://wso2-data-mapper-json-schema-sample-employees",
 "title": "employees",
 "type": "object",
 "description": "This schema represent wso2 employee xml message format",
 "properties": {
 "employee": {
 "id": "http://....employees/employee",
 "type": "array",
 "Items": [
 {
 "id": "http://jsonschema.net/employee/0",
 "type": "object",
 "properties": {
 "firstname": {
 "id": "http://....employee/firstname",
 "type": "string"
 },
 "lastname": {
 "id": "http://....employee/lastname",
 "type": "string"
 },
 "addresses": {
 "id": "http://....employee//addresses",
 "type": "object",
 "properties": {
 "address": {
 "id": "http://....employee/
addresses/address",
 "type": "array",
 "Items": [...]
 }
 }
 }
 }
 },
 "required": [
 "firstname",
 "lastname",
 "address"
]
],
 "required": ["arrayRequired"]
 }
 },
 "required": ["employees"],
 "namespaces": [
 { "prefix": "ns1", "url": "http://wso2.employee.info" },
 { "prefix": "ns2", "url": "http://wso2.employee.address" }
]
}
```

Define the XML attributes under the "attributes" field similar to the "properties" in the element definition. In the above employees example, address array element and city element contain attributes, and those can be represented as follows.

```

"addresses": {
 "id": "http://.../addresses",
 "type": "object",
 "properties": {
 "address": {
 "id": "http://.../addresses/address",
 "type": "array",
 "items": [
 {
 "id": "http://.../addresses/address/element",
 "type": "object",
 "properties": {
 "city": {
 "id": "http://.../addresses/address/element/city",
 "type": "string",
 "attributes": {
 "postalcode": {
 "id": ".../element/city/postalcode",
 "type": "string"
 }
 }
 },
 "road": {
 "id": ".../addresses/address/element/road",
 "type": "string"
 }
 }
 }
],
 "attributes": {
 "location": {
 "id": ".../addresses/address/element/location",
 "type": "string"
 }
 }
 }
 }
}

```

Now, the format of the XML payload is complete. However, you need to define namespaces. You have defined the namespaces used in the payload before with prefix values in the root element under the "namespaces" tag. To assign the namespace to each element, you should only add the prefix before the element name with a colon as "ns1:employees", "ns1:employee" etc.

The complete schema to represent the employee payload is as follows.

```
{
 "$schema": "http://wso2-data-mapper-json-schema/1.0v",
 "id": "http://wso2-data-mapper-json-schema-sample-employees",
 "title": "ns2:employees",
 "type": "object",
 "description": "This schema represent wso2 employee xml message format",
 "properties": {
 "ns2:employee": {
 "id": "http://.../employee",
 "type": "array",
 "items": [
 {

```

```

"id": "http://.../employee/element",
"type": "object",
"properties": {
 "ns2:firstname": {
 "id": "http://.../employee/element/firstname",
 "type": "string"
 },
 "ns2:lastname": {
 "id": "http://.../employee/element/lastname",
 "type": "string"
 },
 "ns1:addresses": {
 "id": "http://.../employees/employee/element/addresses",
 "type": "object",
 "properties": {
 "ns1:address": {
 "id": "http://.../addresses/address",
 "type": "array",
 "items": [
 {
 "id": "http://.../addresses/address/0",
 "type": "object",
 "properties": {
 "ns1:city": {
 "id": "http://.../addresses/address/element/city",
 "type": "string",
 "attributes": {
 "postalcode": {
 "id": "http://.../city/-postalcode",
 "type": "string"
 }
 }
 },
 "ns1:road": {
 "id": "http://.../addresses/address/element/road",
 "type": "string"
 }
 }
 }
],
 "attributes": {
 "location": {
 "id": "http://jsonschema.net/employees/employee/0/addresses/address/0/-location",
 "type": "string"
 }
 }
 }
 }
 },
 "required": [
 "firstname",
 "lastname",
 "address"
]
}

```

```
],
 "required":[
 "arrayRequired"
]
},
},
"required":[
 "employees"
],
"namespaces": [{ "prefix":"ns1",
"url":"http://wso2.employee.address" }, { "prefix":"ns2" ,
```

```

"url": "http://wso2.employee.info"]
}

```

## Using Data Mapper Mediator in WSO2 ESB

- Prerequisites
- Introduction
- Creating the ESB configuration project
- Deploying the configurations
- Invoking the created REST API

### **Prerequisites**

Set up the following prerequisites before you begin.

- Download and run **WSO2 ESB**. For instructions on running the WSO2 ESB server, see [Running the Product](#).
- Install the WSO2 Developer Studio ESB Tool 5.0.0 to use the Data Mapper mediator, which supports the data mapping editor. For instructions on installing this, see [Installing WSO2 ESB Tooling](#).
- Download and launch a REST client into your web browser. For example, this guide uses the [Postman REST client](#) to send the requests to WSO2 ESB and receive the responses.

### **Introduction**

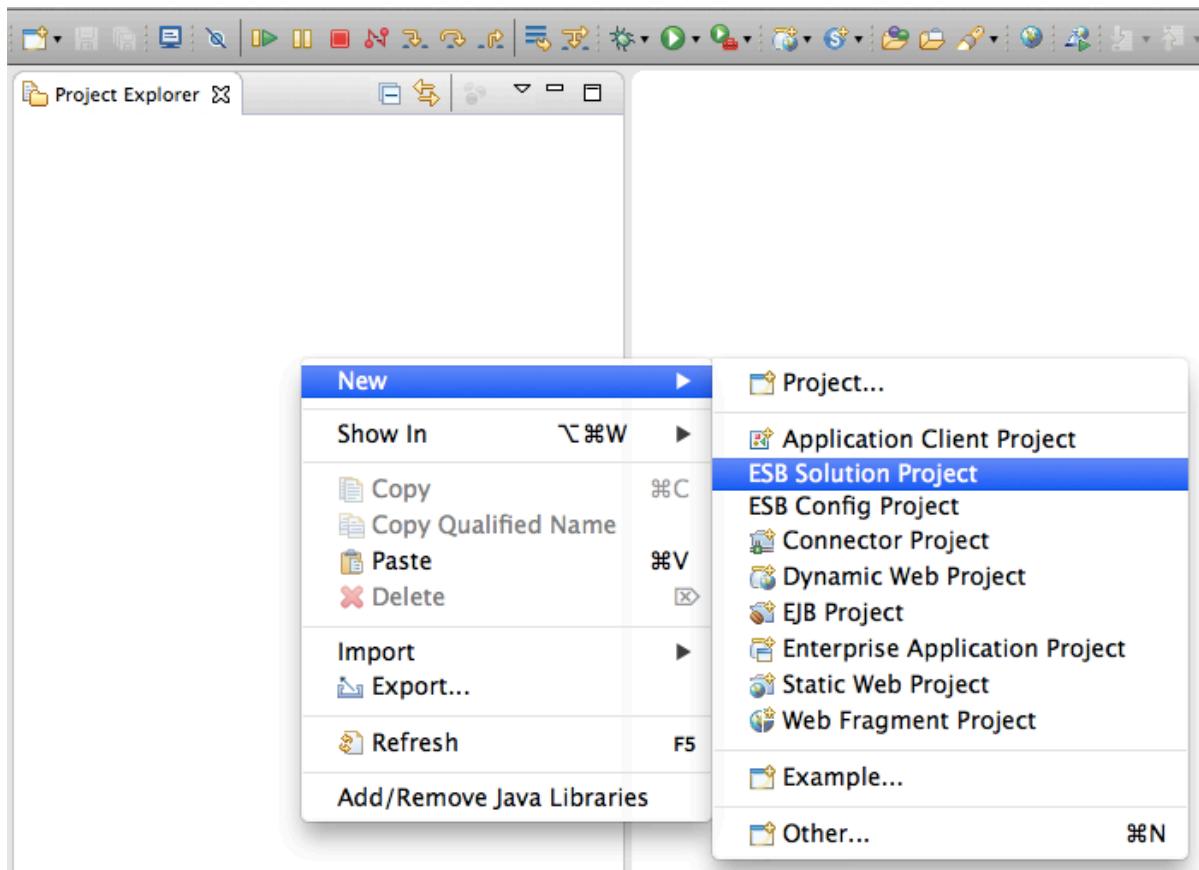
This sample demonstrates how to create a mapping configuration for different data formats using the Data Mapper mediator. It uses a simple WSO2 ESB configuration with only a Data Mapper mediator, and a Respond mediator to check the converted message. The input employee message in XML format, and the output engineer message in JSON format, which is sent to the client as the response.

### **Creating the ESB configuration project**

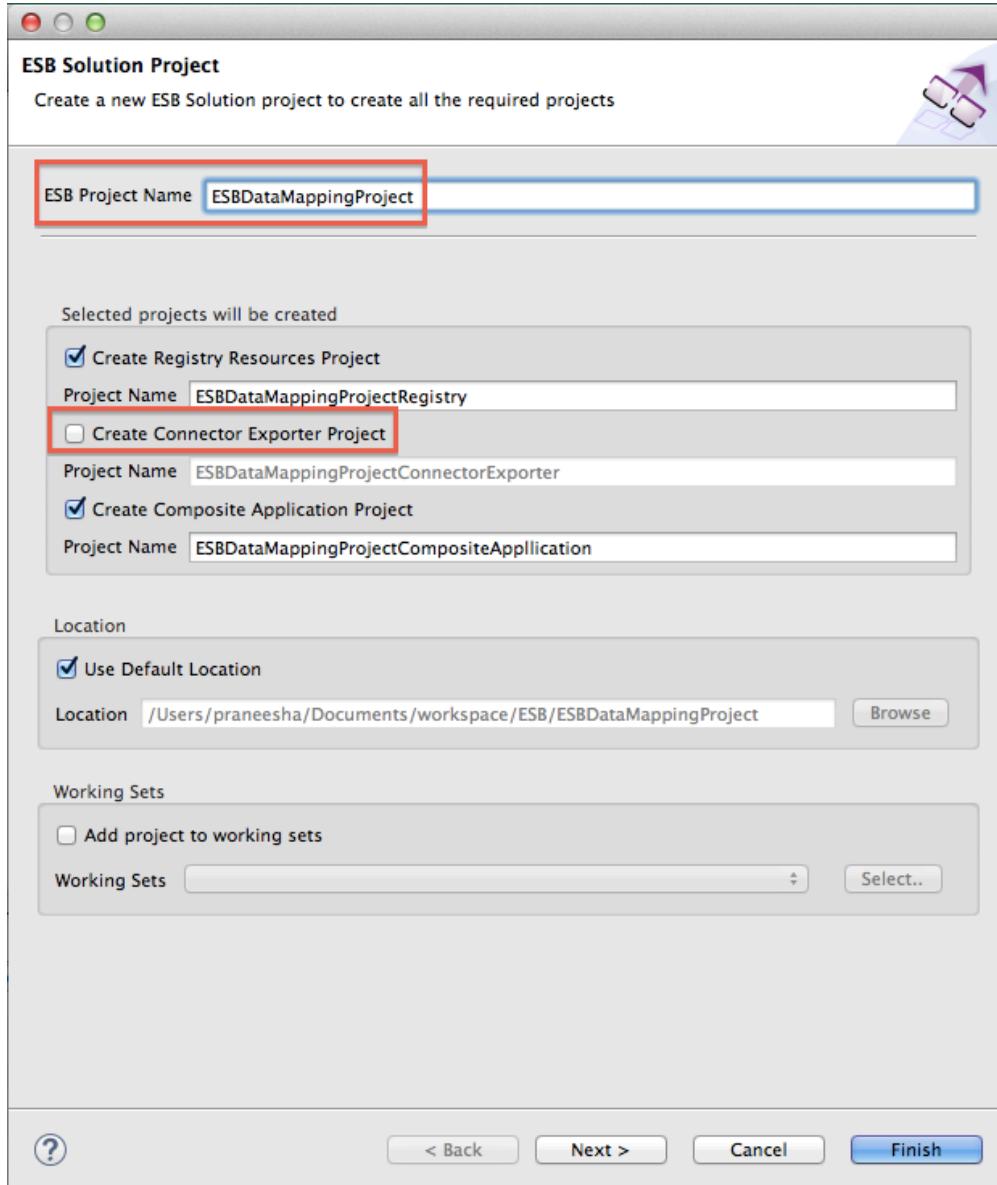
Follow the steps below to create an ESB configuration project to contain the Data Mapping configurations using the WSO2 Developer Studio ESB Tool.

1. Open the WSO2 Developer Studio ESB Tool.
2. Right click on the **Project Explorer** area, click **New**, and then click **ESB Solution Project** as shown below.

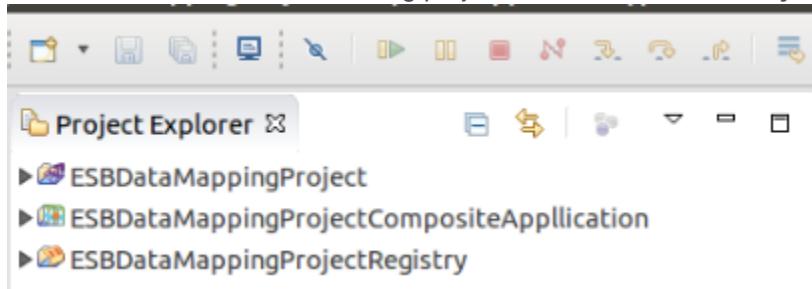
WSO2 ESB Tooling now provides this new option to create an **ESB Solution Project** for you to define all different configurations you need for the project using a wizard.



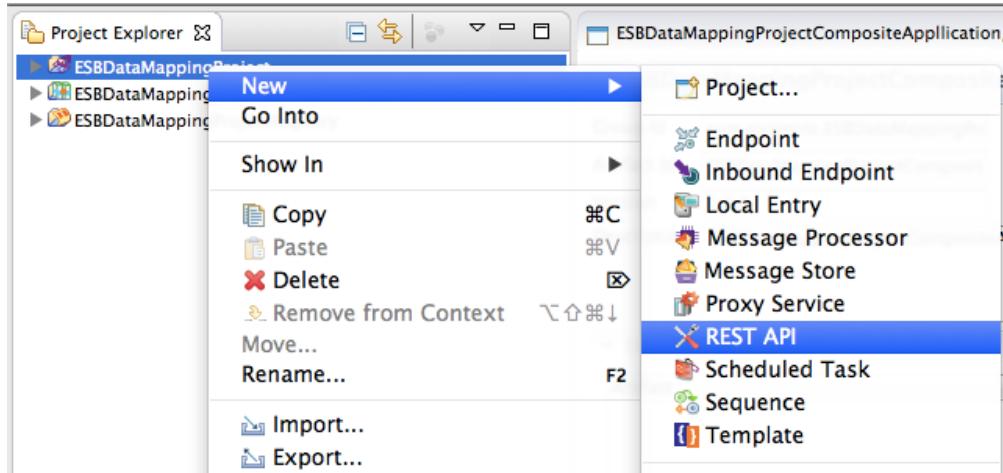
3. Enter a name for the project, and untick **Create Connector Exporter Project** (since you do not need Connectors in your configuration) in the following wizard page.



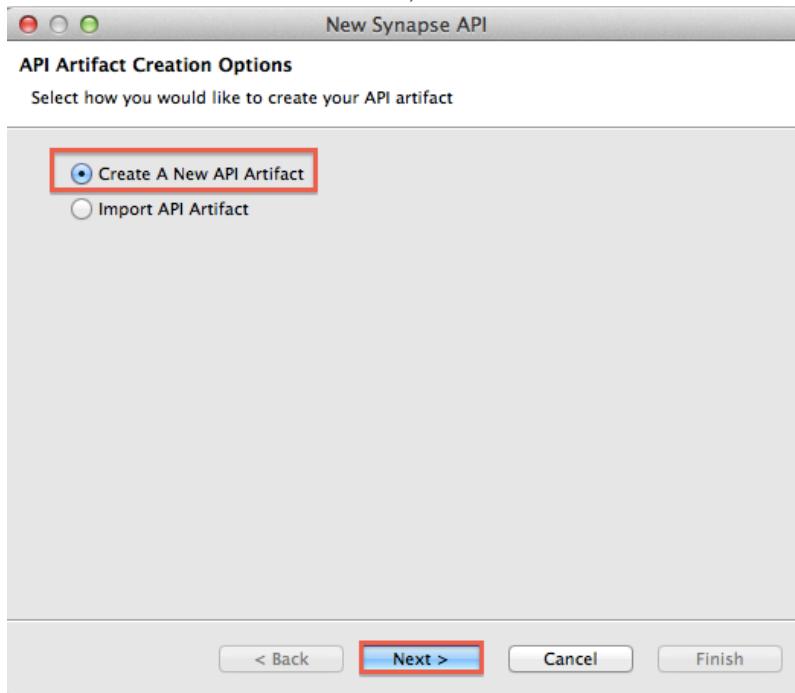
- Click **Finish**. You view the following project files created in the **Project Explorer** tab.



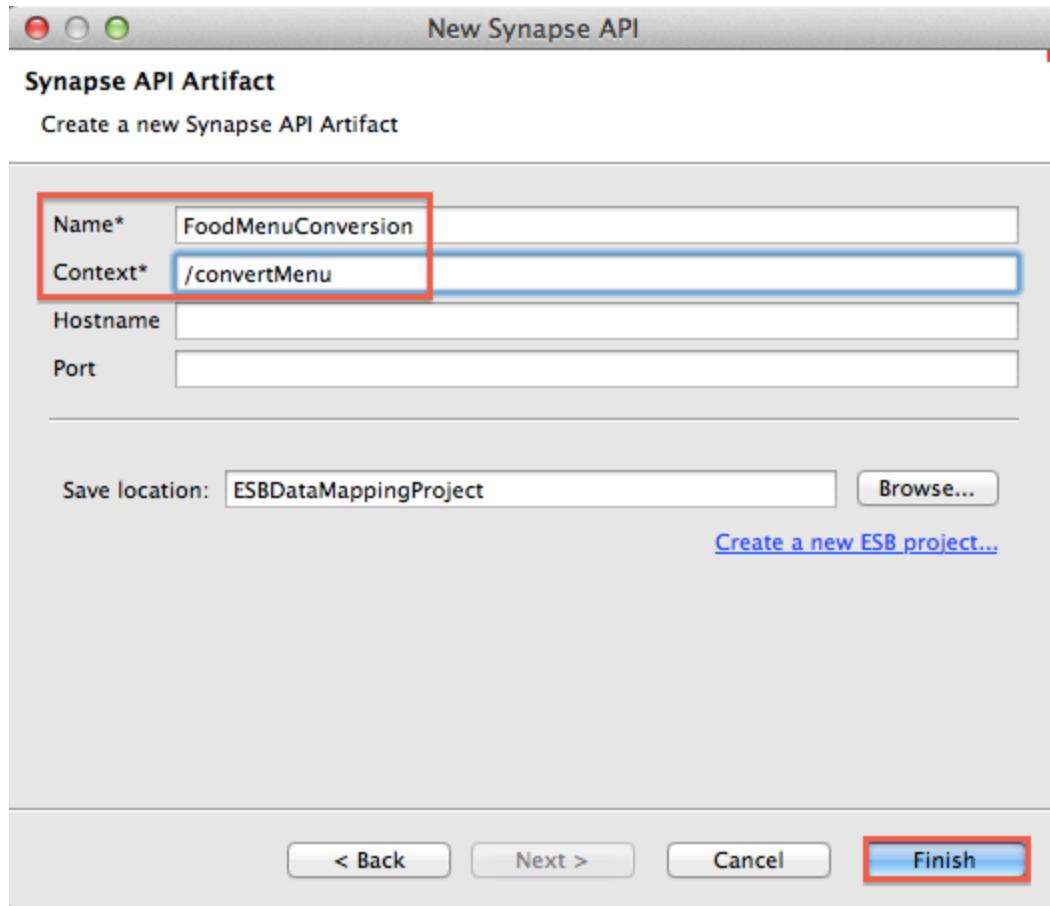
- Right click **ESBDataMappingProject** workspace file, click **New**, and then click **REST API** as shown below, to create a new REST API project in WSO2 ESB.



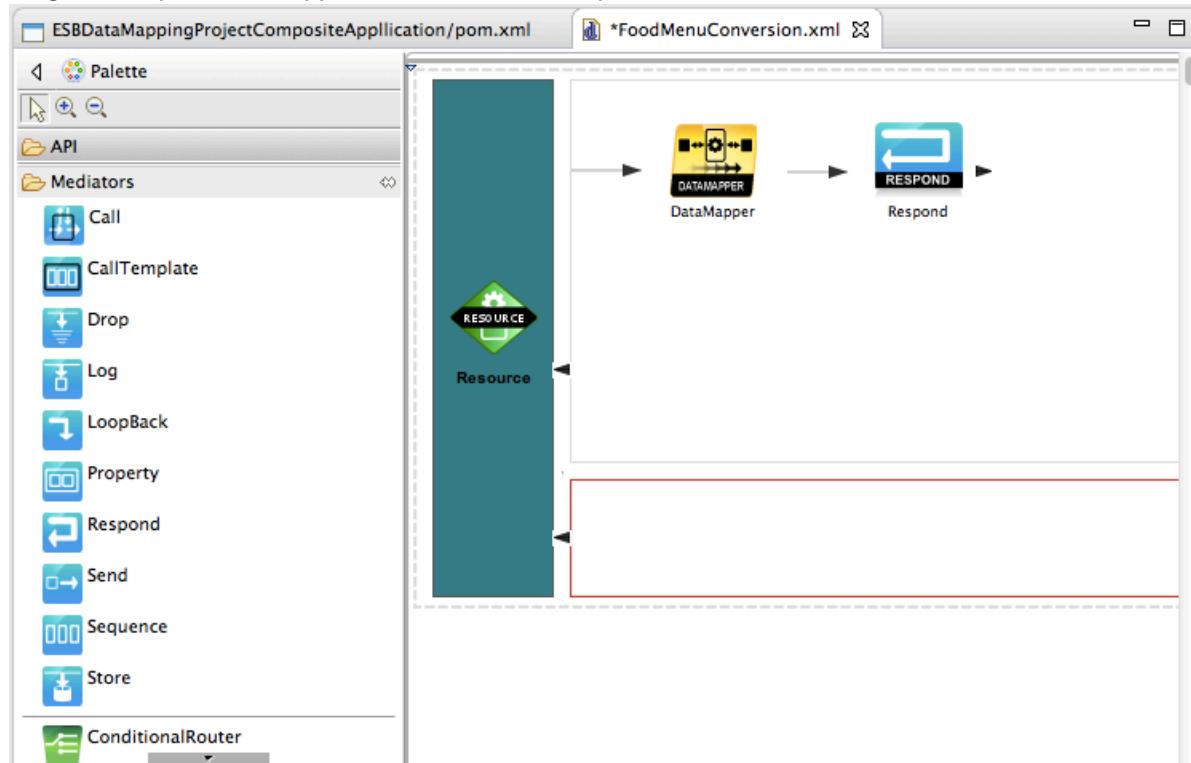
Select **Create A New API Artifact**, and then click **Finish** as shown below.



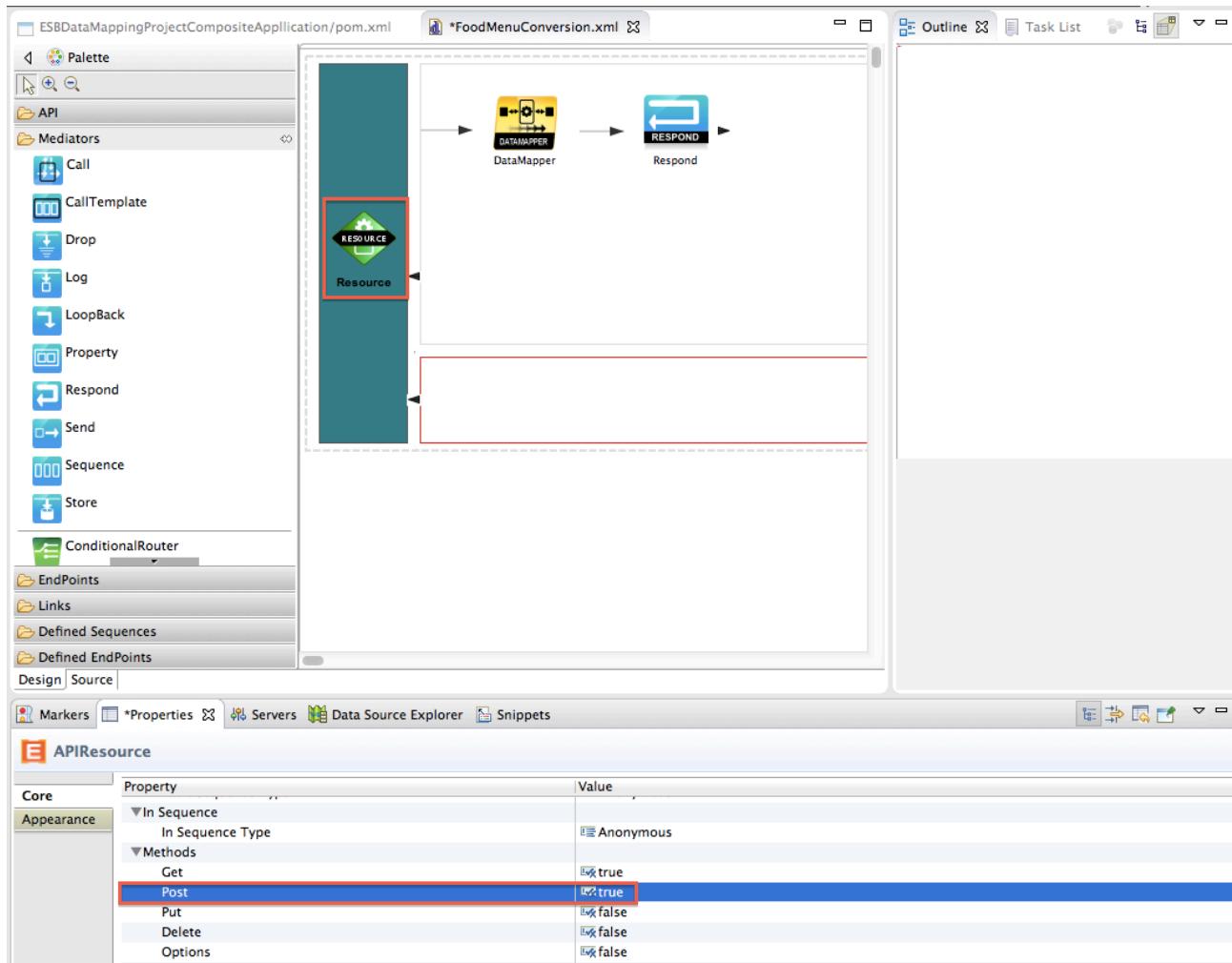
6. Enter a name for the Synapse API Artifact, enter /convertMenu for **Context** to configure the REST API project to listen for POST requests on the /convertMenu URL, and then click **Finish** as shown below.



7. Drag and drop a Data Mapper mediator and a Respond mediator as shown below.

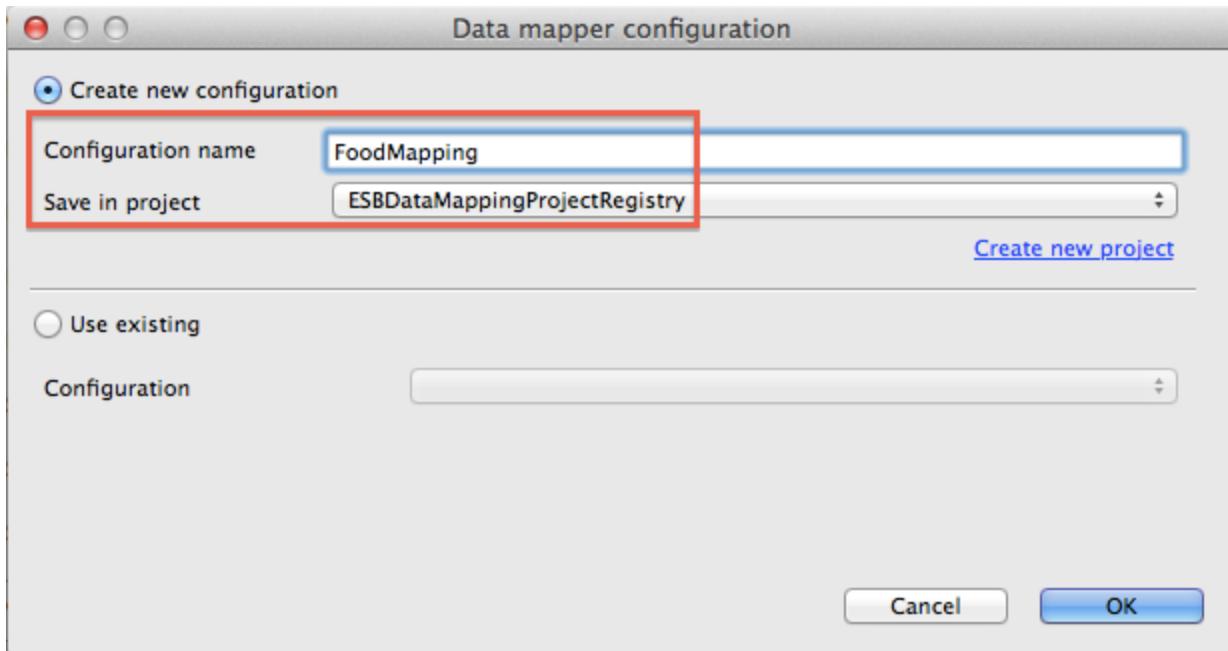


8. Click on the API Resource, and then click on its **Properties** tab, and select **True** as the value for the **Post** method as shown below, to create the API resource listening to POST requests.



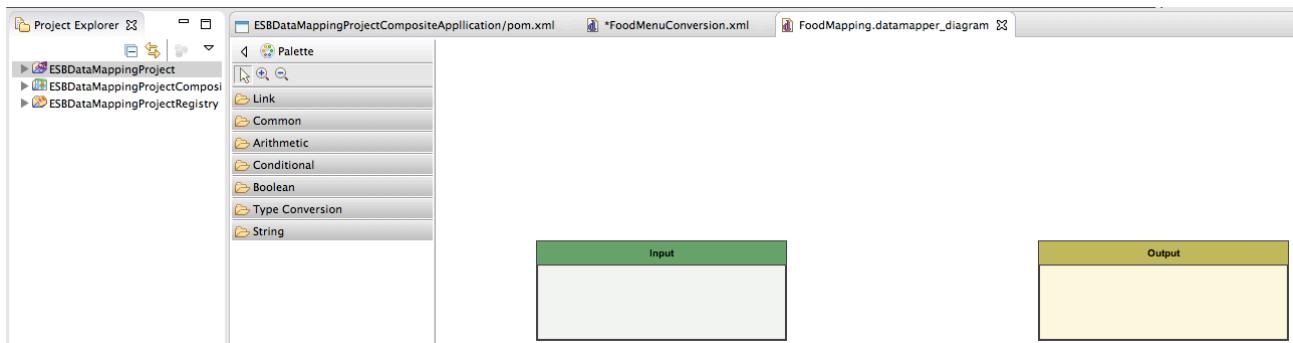
9. Double click on the Data Mapping mediator, to configure it. You view a dialog box to create a registry resource project.
10. Enter a name for the configuration, and point the Registry Resource project to save it as shown below.

This configuration name is the prefix used for the configuration files that you deploy to the ESB server related to the Data Mapper. Since you created an ESB Solution project, it directly points you to that project to save in it. Otherwise, you need to click the **Create new project** link, to create a new Registry Resource project and then point to it.



11. Click **OK**. You view the following Data Mapper diagram editor in the new **WSO2 Data Mapper Graphical** perspective.

You can switch to another perspective by either selecting another in top toolbar tags or by clicking **Window->Perspective->Open Perspective->Other** in the top menu bar.



12. Create an XML file by copying the following sample content of a food menu, and save it in your local file system.

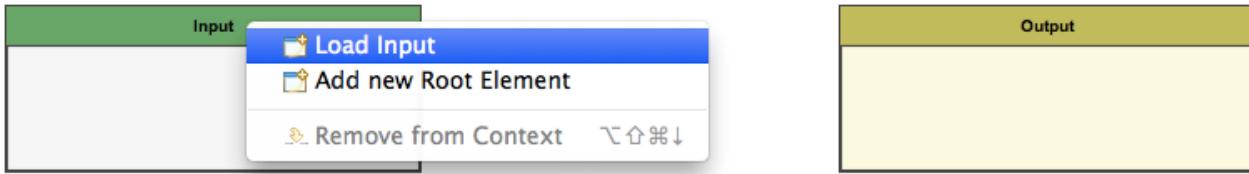
Use this sample XML message to load the input format to the Data Mapper editor.

```

<?xml version="1.0" encoding="UTF-8"?>
<breakfast_menu>
 <food>
 <name>Belgian Waffles</name>
 <price>$5.95</price>
 <description>Two of our famous Belgian Waffles with plenty of real maple syrup</description>
 <calories>650</calories>
 <origin>Belgian</origin>
 <veg>true</veg>
 </food>
 <food>
 <name>Strawberry Belgian Waffles</name>
 <price>$7.95</price>
 <description>Light Belgian waffles covered with strawberries and whipped cream</description>
 <calories>900</calories>
 <origin>Belgian</origin>
 <veg>true</veg>
 </food>
 <food>
 <name>Berry-Berry Belgian Waffles</name>
 <price>$8.95</price>
 <description>Light Belgian waffles covered with an assortment of fresh berries and whipped cream</description>
 <calories>900</calories>
 <origin>Belgian</origin>
 <veg>true</veg>
 </food>
 <food>
 <name>French Toast</name>
 <price>$4.50</price>
 <description>Thick slices made from our homemade sourdough bread</description>
 <calories>600</calories>
 <origin>French</origin>
 <veg>true</veg>
 </food>
 <food>
 <name>Homestyle Breakfast</name>
 <price>$6.95</price>
 <description>Two eggs, bacon or sausage, toast, and our ever-popular hash browns</description>
 <calories>950</calories>
 <origin>French</origin>
 <veg>false</veg>
 </food>
</breakfast_menu>

```

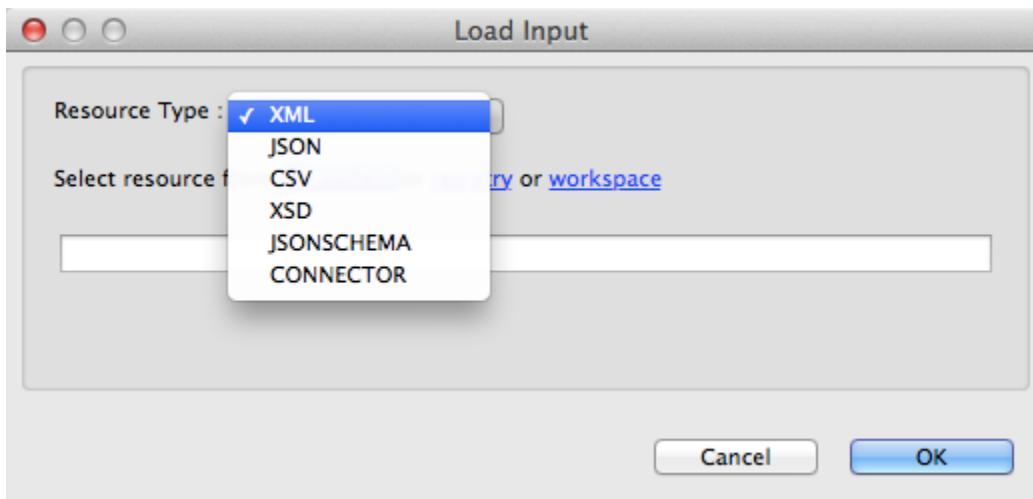
13. Right-click on the top title bar of the **Input** box and, click **Load Input** as shown below. The operation palettes that appear on the left-hand side allows you to provide the input message format to begin the mapping.



14. Select **XML** as the **Resource Type** as shown below.

You can select one out of the following resource types, to load the input and output message formats to Data Mapper.

- **XML**: to load a sample XML message and WSO2 Data Mapper Editor will generate the JSON schema to represent the XML according to the WSO2 Data Mapper Schema specification.
- **JSON**: to load a sample JSON message.
- **CSV**: to load a sample JSON/CSV message. **For CSV you need to provide the column names as the first record.**
- **XSD**: to load an XSD schema file, which defines your XML message format.
- **JSONSCHEMA**: to load a JSON schema for your message according to the WSO2 Data Mapper schema specification.
- **CONNECTOR**: to map a message, which is an output of a Connector. Select the **Connector Type** in the **Input** box, and it will list down all available connectors. Then, select the operation from the menu that appears in front of Data Mapper mediator.



15. Click the **file system** link in **Select resource from**, select the XML file you saved in your local file system in step 12, and click **Open**.

You view the input format loaded in the **Input** box in the editor as shown below.

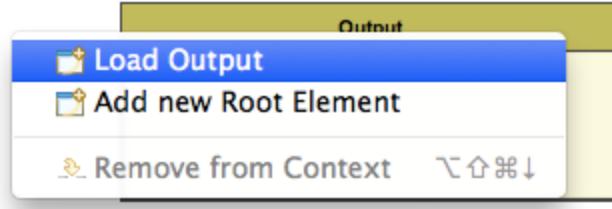


16. Create another XML file by copying the following sample content of a food menu, and save it in your local file system.

Use this sample XML message to load the output format to the Data Mapper editor.

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
 <item>
 <name>Belgian Waffles</name>
 <price>$5.95</price>
 <calories>650</calories>
 <origin>Belgian</origin>
 <veg>true</veg>
 <description>Two of our famous Belgian Waffles with plenty of real maple
syrup</description>
 </item>
 <item>
 <name>Strawberry Belgian Waffles</name>
 <price>$7.95</price>
 <calories>900</calories>
 <origin>Belgian</origin>
 <veg>true</veg>
 <description>Light Belgian waffles covered with strawberries and whipped
cream</description>
 </item>
 <item>
 <name>Berry-Berry Belgian Waffles</name>
 <price>$8.95</price>
 <calories>900</calories>
 <origin>Belgian</origin>
 <veg>true</veg>
 <description>Light Belgian waffles covered with an assortment of fresh
berries and whipped cream</description>
 </item>
 <item>
 <name>French Toast</name>
 <price>$4.50</price>
 <calories>600</calories>
 <origin>French</origin>
 <veg>true</veg>
 <description>Thick slices made from our homemade sourdough
bread</description>
 </item>
 <item>
 <name>Homestyle Breakfast</name>
 <price>$6.95</price>
 <calories>950</calories>
 <origin>French</origin>
 <veg>false</veg>
 <description>Two eggs, bacon or sausage, toast, and our ever-popular hash
browns</description>
 </item>
</menu>
```

17. Right-click on the top title bar of the **Output** box and, click **Load Output** as shown below. The operation palettes that appear on the left-hand side allows you to provide the output message format.



18. Click the **file system** link in **Select resource from**, select the XML file you saved in your local file system in **step 16**, and click **Open**.

You view the input format loaded in the **Output** box in the editor as shown below.

```

Output
{} menu
[] item
 <> name : [STRING]
 <> price : [STRING]
 <> calories : [NUMBER]
 <> origin : [STRING]
 <> veg : [BOOLEAN]
 <> description : [STRING]
```

19. Check the **Input** and **Output** boxes with the sample messages, to see if the element types (i.e. (Arrays, Objects and Primitive values) are correctly identified or not. Following signs will help you to identify them correctly.

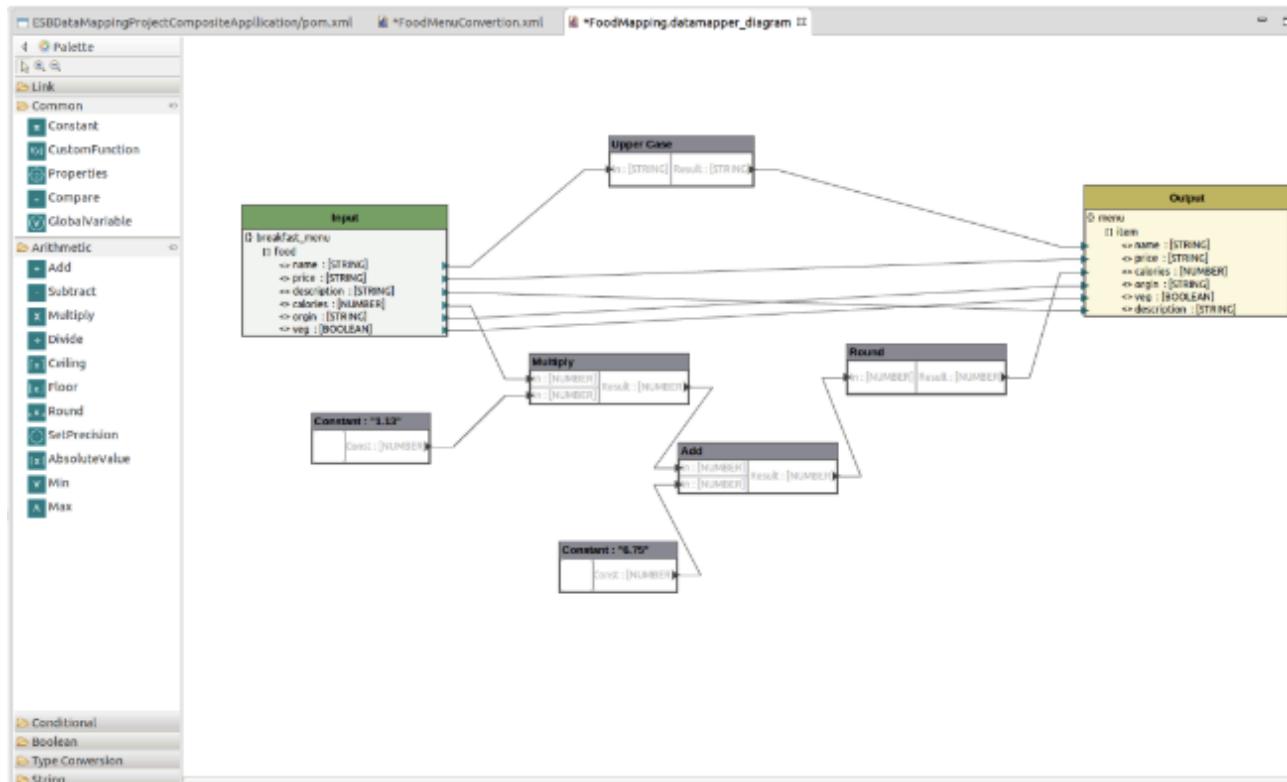
- {} - represents object elements
- [] - represents array elements
- <> - represents primitive field values
- A - represents XML attribute values

20. Do the mapping as preferred using operators as shown in the example below.

You can only connect primitive data values such as Strings, numbers, boolean and etc. You cannot map Array and object values.

The mapping done in the below example is that, name is mapped via uppercase operator and calories undergoes a mathematical calculation to get the output as follows:

`output calories =Round( (calories*1.13) + 6.75)`



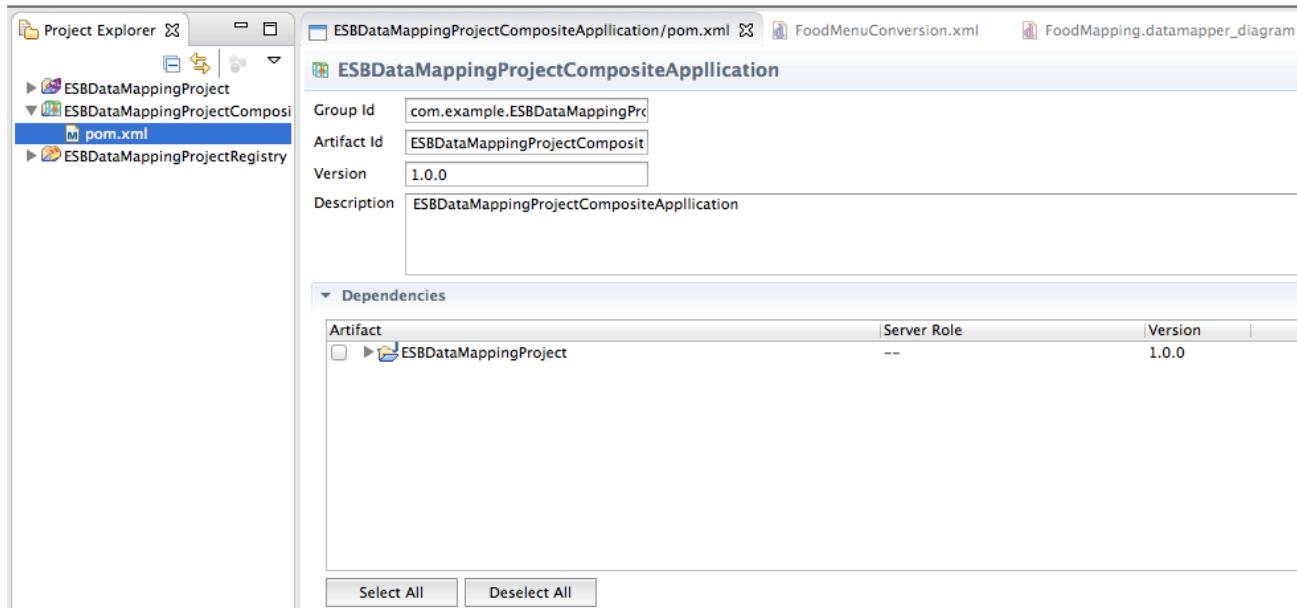
21. Press Ctrl+S keys in each tab, to save all the configurations.

### **Deploying the configurations**

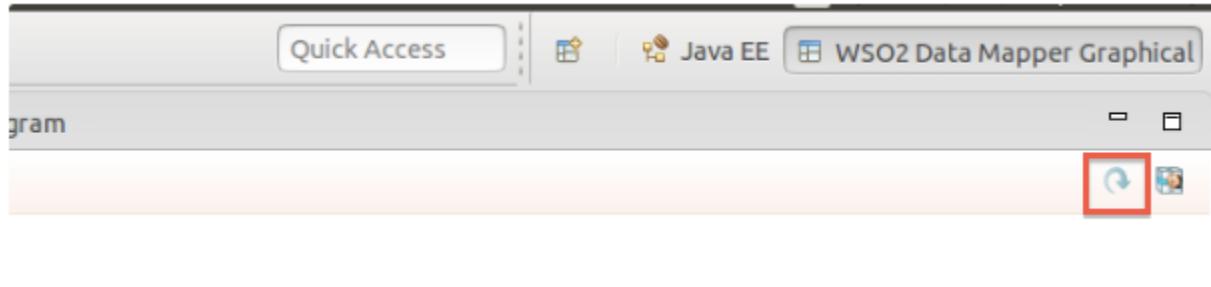
After creating the Data Mapper configurations, follow the steps below to deploy the created REST API and the configurations in the WSO2 ESB server by including them in a C-App.

1. Open the WSO2 Developer Studio ESB Tool.
2. Expand the C-APP project that was created when you created the ESB Solution project (i.e. **ESBDataMappingProjectCompositeApplication**), and double-click on the POM file. You view the following screen to select project files into the C-APP.

You need to refresh the screen to view the registry resource files . Once you refresh the screen, you view all the artifacts in the workspace.



3. Click the refresh button in the top right-hand corner to load newly added registry files, as shown below.



4. Select the REST API file and the three registry resource files containing the mapping configuration, input schema, and output schema as shown below.

- **Configuration:** Script file that is used to execute the mapping.
- **Input schema:** JSON schema which represents the input message format.
- **Output schema:** JSON schema which represents the output message format.

The screenshot shows the WSO2 Developer Studio interface. The top bar has tabs for 'ESBDataMappingProjectCompositeApplication/pom.xml', 'FoodMapping.datamapper\_diagram', and 'Developer Studio Dashboard'. The main area displays the project details for 'ESBDataMappingProjectCompositeApplication' with fields for Group Id (com.example.ESBDataMappingProj), Artifact Id (ESBDataMappingProjectComposit), Version (1.0.0), and Description (ESBDataMappingProjectCompositeApplication). Below this is a 'Dependencies' section containing a table:

Artifact	Server Role	Version
<input checked="" type="checkbox"/> ESBDataMappingProject	--	1.0.0
<input checked="" type="checkbox"/> com.example.ESBDataMappingProject.api_._FoodMenuConversion	EnterpriseServiceBus	1.0.0
<input checked="" type="checkbox"/> ESBDataMappingProjectRegistry	--	1.0.0
<input checked="" type="checkbox"/> com.example.ESBDataMappingProjectRegistry.resource_._Foo...	EnterpriseServiceBus	1.0.0
<input checked="" type="checkbox"/> com.example.ESBDataMappingProjectRegistry.resource_._Foo...	EnterpriseServiceBus	1.0.0
<input checked="" type="checkbox"/> com.example.ESBDataMappingProjectRegistry.resource_._Foo...	EnterpriseServiceBus	1.0.0

At the bottom of the dependencies panel are 'Select All' and 'Deselect All' buttons.

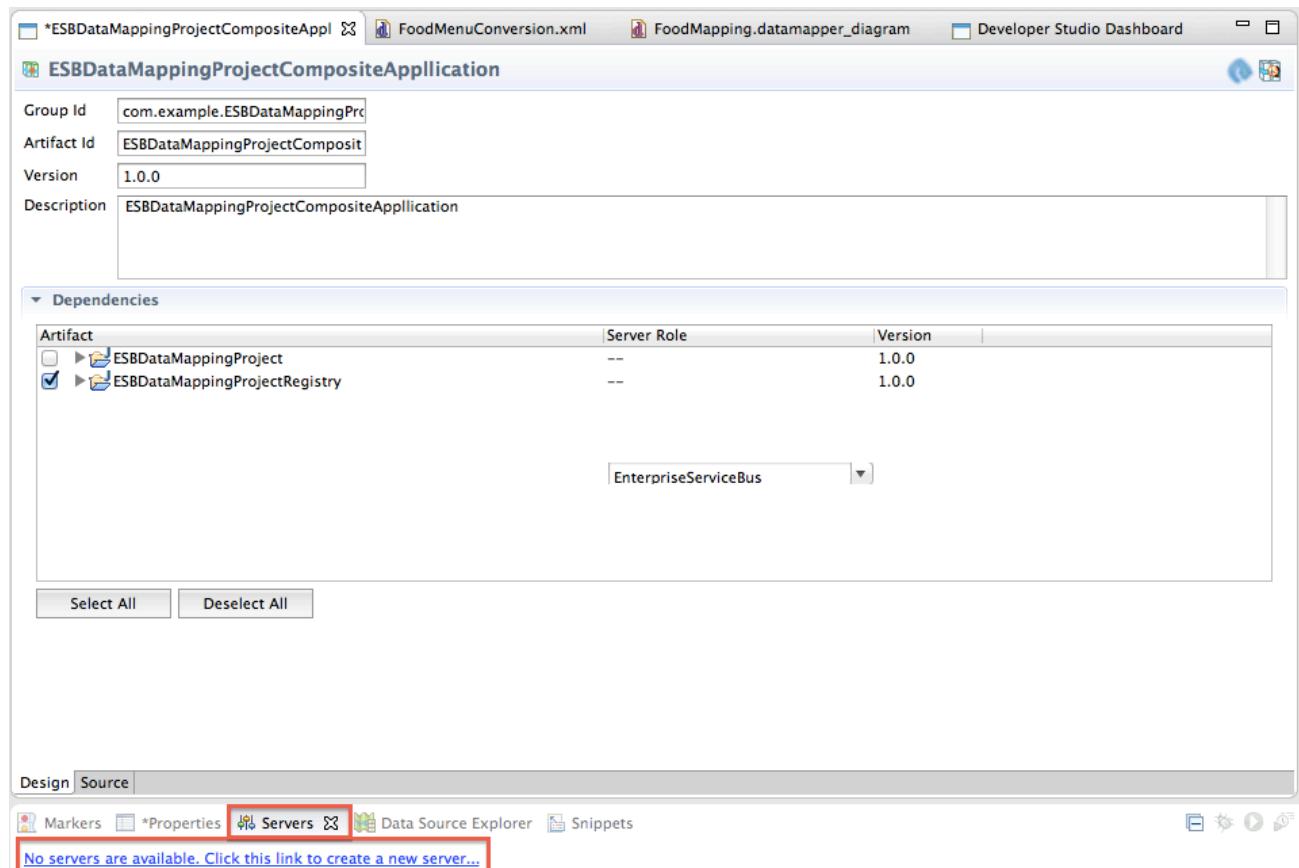
- Click on the Data Mapper mediator. You view the following in the **Properties** tab of the Data Mapper mediator configuration as shown below.
  - Configuration:** Script file that is used to execute the mapping.
  - Input Schema:** JSON schema, which represents the input message format.
  - Output Schema:** JSON schema, which represents the output message format.
  - Input Type:** Expected input message type (xml/json/csv).
  - Output Type:** Target output message type (xml/json/csv).

The screenshot shows the 'Properties' tab for a 'DataMapperMediator' component. The left sidebar shows tabs for 'Core' and 'Appearance', with 'Core' selected. The main table lists properties and their values:

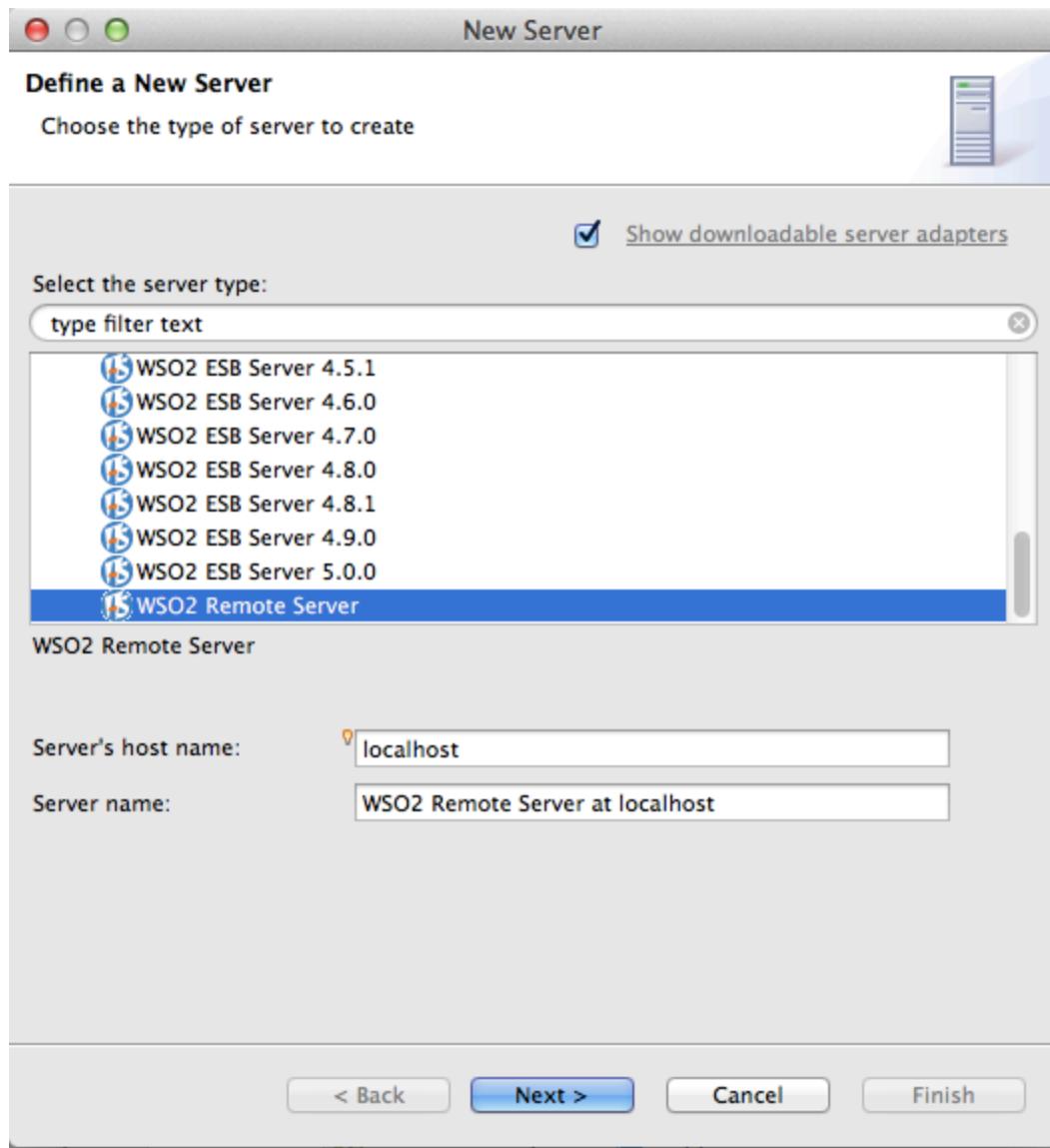
Property	Value
Configuration	gov:datamapper/FoodMapping.dmc
Input Schema	gov:datamapper/FoodMapping_inputSchema.json
Output Schema	gov:datamapper/FoodMapping_outputSchema.json
Input Type	XML
Output Type	XML

If your mapping failed during runtime, check if the input type and output type are set correctly in the mediator configuration.

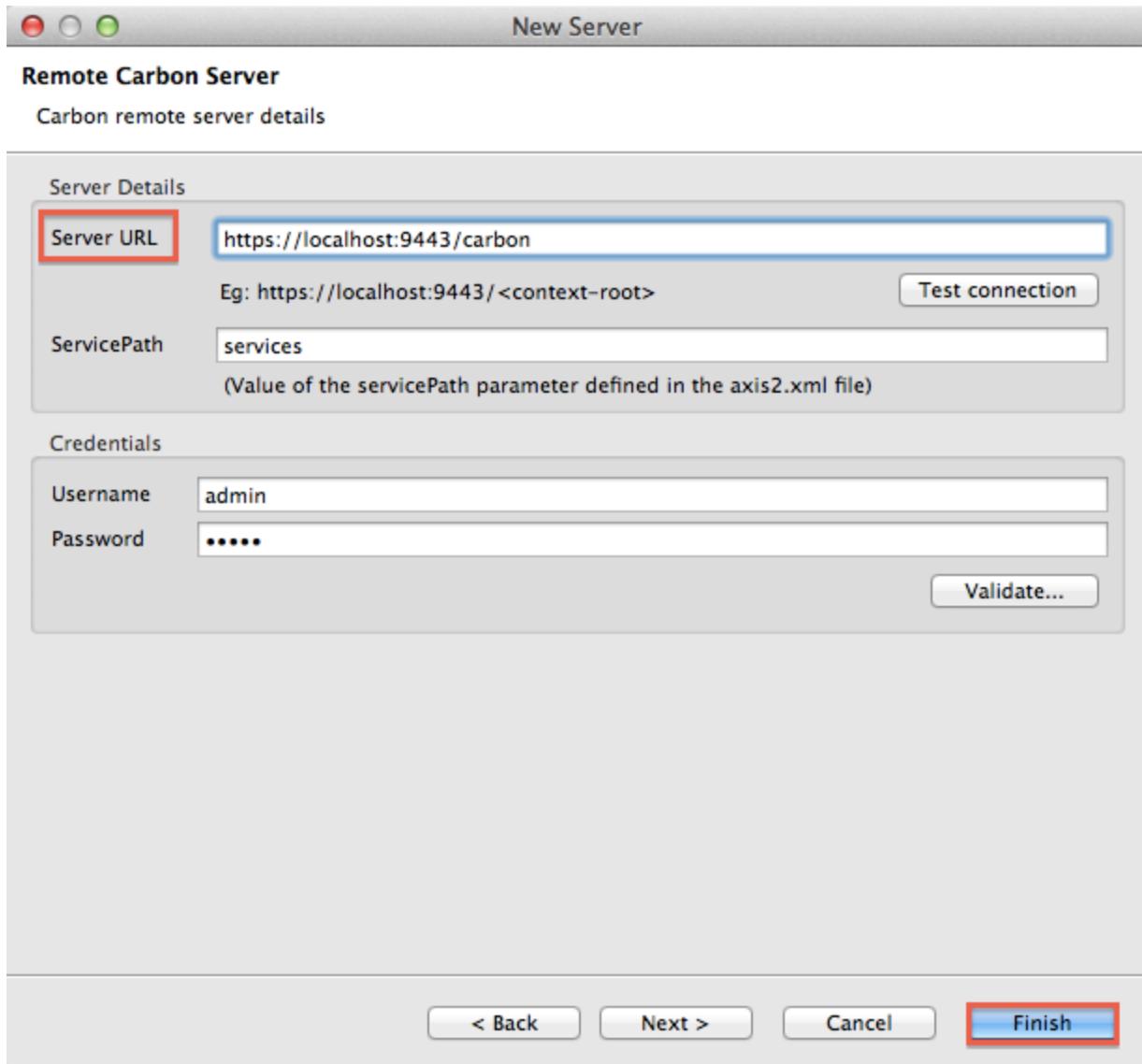
- Start WSO2 ESB server. For instructions, see [Running the Product](#).
- Click the **Servers** Tab in the WSO2 Developer Studio ESB Tool, and click the **No servers are available. Click this link to create a new server...** link as shown below.



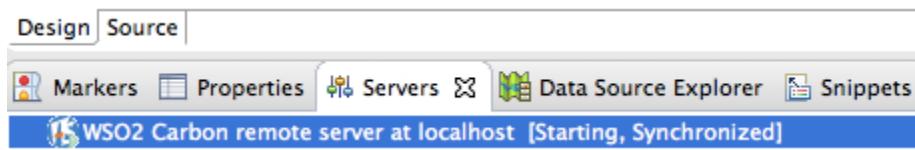
8. Click **WSO2**, click **WSO2 Carbon remote server**, and then click **Next** as shown below.



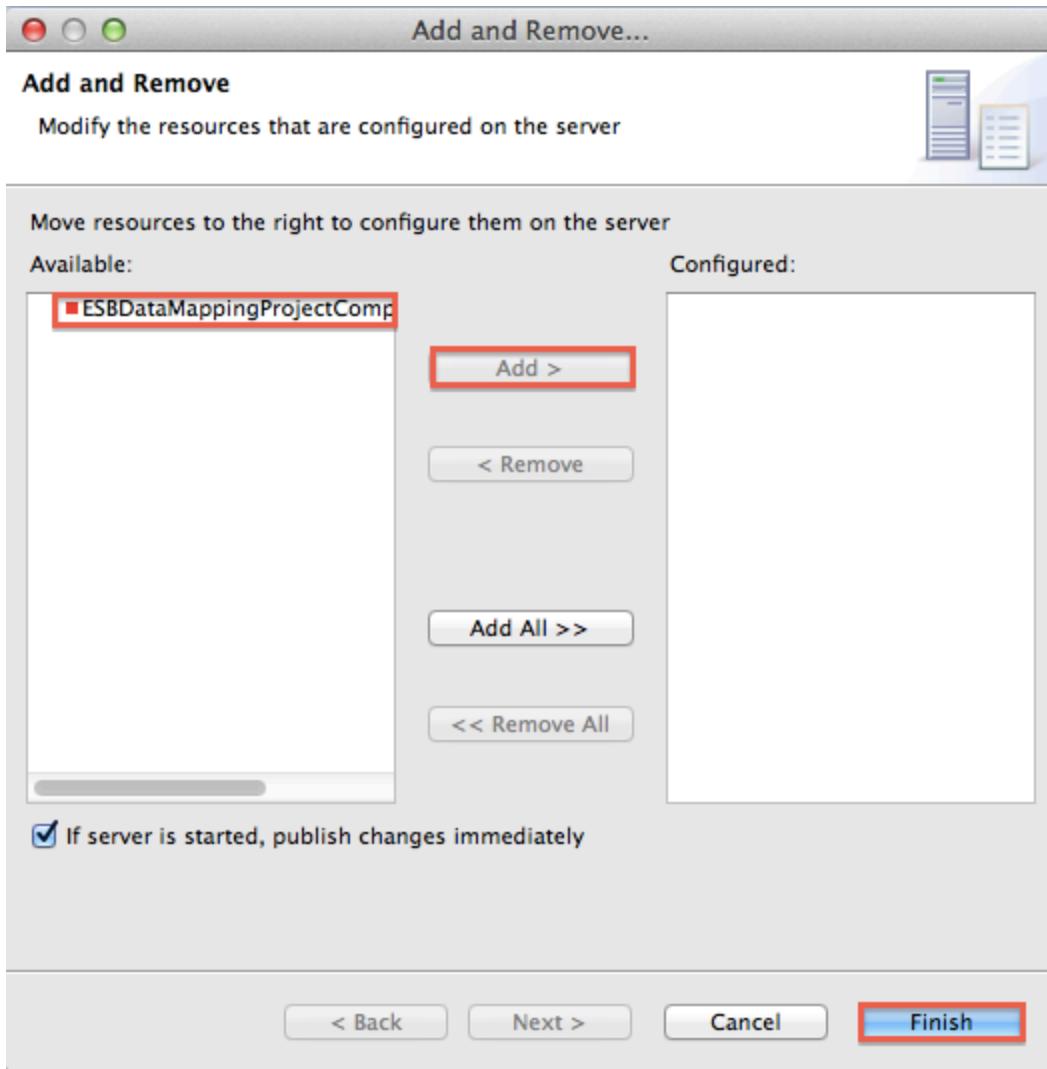
9. Enter the URL of WSO2 ESB for **Server URL**, and click **Finish** as shown below.



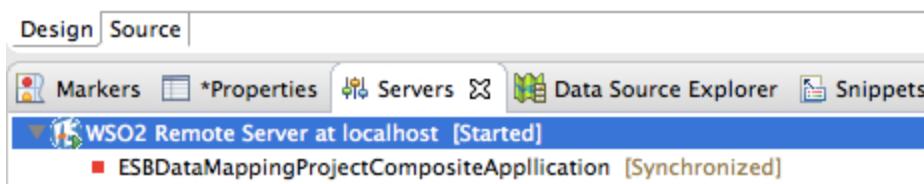
You view the WSO2 ESB server added in the **Servers** tab as shown below.



10. Right-click on **WSO2 Carbon remote server at localhost**, and then click **Add & Remove**.
11. Select the C-App in the **Available:** box, click **Add** to move it to the **Configured:** box, and then click **Finish** as shown below.



You view the C-App added to the WSO2 ESB server as shown below.



12. Log in to the WSO2 ESB Management Console using the following URL and admin/admin credentials: [https://<ESB\\_HOST>:<ESB\\_PORT>carbon/](https://<ESB_HOST>:<ESB_PORT>carbon/)
13. Click **Main**, and then click **APIs** in the **Service Bus** menu. You view the deployed REST API invocation URL as shown below.

Select	API Name	API Invocation URL	Action
<input type="checkbox"/>	FoodMenuConversion	http://10.100.5.72:8280/convertMenu	<a href="#">Enable Statistics</a> <a href="#">Edit</a> <a href="#">Delete</a>

#### Invoking the created REST API

Follow the steps below to test invoking the created REST API.

1. Open Postman REST client.
2. Enter the following details to create the client message, enter the content of the XML file you created in step 12 as the payload in the text area provided, and click **Send** as shown below.
  - **URL:** `http://<ESB_HOST>:<ESB_PORT>8280/convertMenu`
  - **Method:** POST
  - **Body:** raw xml/application
  - **Message:** Enter the input

The screenshot shows the Postman interface with a red box highlighting the XML payload in the 'Body' tab. The XML content is as follows:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <breakfast_menu>
3 <food>
4 <name>Belgian Waffles</name>
5 <price>$5.95</price>
6 <description>Two of our famous Belgian Waffles with plenty of real maple syrup</description>
7 <calories>650</calories>
8 <origin>Belgian</origin>
9 <veg>true</veg>
10 </food>
11 <food>
12 <name>Strawberry Belgian Waffles</name>
13 <price>$7.95</price>
14 <description>Light Belgian waffles covered with strawberries and whipped cream</description>
15 <calories>900</calories>
16 <origin>Belgian</origin>
17 <veg>true</veg>
18 </food>
19 </breakfast_menu>

```

You view the expected JSON message received as shown below.

The screenshot shows the Postman interface with a red box highlighting the JSON payload in the 'Body' tab. The JSON content is as follows:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <menus>
3 <item>
4 <name>FRENCH TOAST</name>
5 <price>$4.50</price>
6 <calories>685.0</calories>
7 <origin>French</origin>
8 <veg>true</veg>
9 <description>Thick slices made from our homemade sourdough bread</description>
10 </item>
11 <item>
12 <name>BERRY-BERRY BELGIAN WAFFLES</name>
13 <price>$8.95</price>
14 <calories>1024.0</calories>
15 <origin>Belgian</origin>
16 <veg>true</veg>
17 <description>Light Belgian waffles covered with an assortment of fresh berries and whipped cream</description>
18 </item>
19 <item>
20 <name>STRAWBERRY BELGIAN WAFFLES</name>
21 <price>$7.95</price>
22 <calories>1024.0</calories>
23 <origin>Belgian</origin>
24 <veg>true</veg>
25 <description>Light Belgian waffles covered with strawberries and whipped cream</description>
26 </item>
27 <item>
28 <name>BELGIAN WAFFLES</name>
29 <price>$5.95</price>
30 <calories>741.0</calories>
31 <origin>Belgian</origin>
32 <veg>true</veg>
33 <description>Two of our famous Belgian Waffles with plenty of real maple syrup</description>
34 </item>
35 <item>
36 <name>HOMESTYLE BREAKFAST</name>
37 <price>$6.95</price>
38 <calories>1080.0</calories>
39 <origin>French</origin>
40 </item>
41 </menus>

```

Similarly, you can use the above instructions to check the following message conversions:

- The input employee message in XML format, and the output engineer message in XML/JSON/CSV formats, which is sent to the client as the response. (i.e. XML->XML/JSON/CSV)
- The input employee message in JSON format, and the output engineer message in XML/JSON/CSV formats, which is sent to the client as the response. (i.e. JSON->XML/JSON/CSV)
- The input employee message in CSV format, and the output engineer message in XML/JSON/CSV formats, which is sent to the client as the response. (i.e. CSV->XML/JSON/CSV)

In the above sample, the output message format is fully compatible to represent as JSON and CSV.

However, this is not guaranteed in every occasion. For example, if you have defined a complex XML output message with namespaces and attributes, JSON message or CSV will not be built as expected.

## DBLookup Mediator

The **DBLookup Mediator** can execute an arbitrary SQL select statement and then set a resulting values as a local message property in the message context. The DB connection used may be looked up from an external data source or specified inline.

The DBLookup mediator can set a property from one row in a result set. It cannot return multiple rows. If you need to get multiple records, or if you have a table with multiple parameters (such as URLs), you can use the [WSO2 Data Services Server](#) to create a data service and invoke that service from the ESB using the [Callout mediator](#) instead.

The DBLookup mediator is a [content-aware](#) mediator.

[Syntax](#) | [UI Configuration](#) | [Examples](#) | [Samples](#)

### Syntax

The syntax of the DBLookup mediator changes depending on whether you connect to the database using a connection pool, or using a data source. Click on the relevant tab to view the required syntax.

[Connection Pool](#)[Data source](#)

```
<dblookup>
 <connection>
 <pool>
 <driver/>
 <url/>
 <user/>
 <password/>
 <property name="name" value="value" />*
 </pool>
 </connection>
 <statement>
 <sql>select something from table where something_else = ?</sql>
 <parameter [value="" | expression=""]
 type="CHAR|VARCHAR|LONGVARCHAR|NUMERIC|DECIMAL|BIT|TINYINT|SMALLINT|INTEGER|BIGINT|REAL|FLOAT|DOUBLE|DATE|TIME|TIMESTAMP"/>*
 <result name="string" column="int|string"/>*
 </statement>+
</dblookup>
```

The syntax of the DBLookup mediator further differs based on whether the connection to the database is made using an external datasource or a Carbon datasource. Click on the relevant tab to view the required syntax.

[External Datasource](#) [Carbon Datasource](#)

```
<dblookup>
<connection>
 <pool>
 <dsName/>
 <icClass/>
 <url/>
 <user/>
 <password/>
 <property name="name" value="value"/>*
 </pool>
</connection>
<statement>
 <sql>select something from table where something_else = ?</sql>
 <parameter [value="" | expression=""]
type="CHAR|VARCHAR|LONGVARCHAR|NUMERIC|DECIMAL|BIT|TINYINT|SMALLINT|INTEGER|BIGINT|REAL|FLOAT|DOUBLE|DATE|TIME|TIMESTAMP"/>*
 <result name="string" column="int|string"/>*
</statement>+
</dblookup>
```

```
<dblookup>
<connection>
 <pool>
 <dsName/>
 </pool>
</connection>
<statement>
 <sql>select something from table where something_else = ?</sql>
 <parameter [value="" | expression=""]
type="CHAR|VARCHAR|LONGVARCHAR|NUMERIC|DECIMAL|BIT|TINYINT|SMALLINT|INTEGER|BIGINT|REAL|FLOAT|DOUBLE|DATE|TIME|TIMESTAMP"/>*
 <result name="string" column="int|string"/>*
</statement>+
</dblookup>
```

## UI Configuration

The UI configuration of the DBLookup mediator changes depending on whether you connect to the database using a connection pool, or using a data source. Click on the relevant tab to view the required UI configuration.

[Connection Pool](#) [Data source](#)

The screenshot shows the configuration interface for the DB Lookup Mediator. At the top, there are tabs for 'Mediator' and 'switch to source view'. A 'Help' button is in the top right. The main section is titled 'DB Lookup Mediator'. Under 'Connection Information', there are four fields: 'Driver' (with a dropdown menu), 'Url', 'User', and 'Password'. There are two radio buttons: 'Pool' (selected) and 'Data Source'. Below this is a 'Properties' section with a 'Add Property' button. Further down are 'SQL Statements' and 'Add Statement' sections.

The parameters available to configure the DBLookup mediator are as follows:

Parameter Name	Description
<b>Connection Information</b>	This parameter is used to specify whether the connection should be taken from a connection pool or a datasource.
<b>Driver</b>	The class name of the database driver.
<b>URL</b>	JDBC URL of the database where the data will be looked up.
<b>User</b>	Username used to connect to the database.
<b>Password</b>	Password used to connect to the database.

#### Adding properties to the DBLookup mediator

If you click **Add Property**, the page will expand to display the following parameters.

The screenshot shows the 'Properties' configuration page. It features a table with columns for 'Name', 'Value', and 'Action'. The 'Name' column has a dropdown menu with 'Select A Property'. The 'Value' column contains an input field. The 'Action' column includes a delete icon. Below the table is a 'Add Property' button.

The parameters available to manage properties are as follows:

Parameter Name	Description

<b>Name</b>	The name of the property.
<b>Value</b>	The value of the property.
<b>Action</b>	This parameter enables a property to be deleted.

The available properties are as follows.

<b>Name</b>	<b>Value</b>	<b>Description</b>
autocommit	true / false	The auto-commit state of the connection created by the pool.
isolation	Connection.TRANSACTION_NONE / Connection.TRANSACTION_READ_COMMITTED / Connection.TRANSACTION_READ_UNCOMMITTED / Connection.TRANSACTION_REPEATABLE_READ / Connection.TRANSACTION_SERIALIZABLE	The isolation state of the connection created by the pool.
initialsize	int	The initial number of connections created when the pool is started.
maxactive	int	The maximum number of active connections that can be allocated from the pool at a given time. When this maximum limit is reached, no more active connections will be created by the connection pool. Specify 0 or a negative value if you do not want to set a limit.

maxidle	int	<p>The maximum number of idle connections allowed in the connection pool at a given time. The value should be less than maxActive value. For high performance tune maxIdle to match the number of average, concurrent requests to the pool. If this value is set to a large value, the pool will contain unnecessary idle connections.</p> <p>The enabled idle connections are checked periodically whenever a new connection is requested, and connections that are being idle for longer than minEvictableIdleTimeMillis are released, since it takes time to create a new connection.</p>
maxopenstatements	int	<p>The maximum number of open statements that can be allocated from the statement pool at a given time. When this maximum limit is reached, no more new statements will be created by the statement pool. Specify 0 or a negative value if you do not want to set a limit.</p>
maxwait	long	<p>The maximum number of milliseconds the connection pool will wait for a connection to return before throwing an exception when there are no connections available in the pool. Specify 0 or a negative value if you want the pool to wait indefinitely.</p>

minidle	int	<p>The minimum number of idle connections allowed in the connection pool at a given time, without extra ones being created. The default value is 0, and is derived from <code>maxsize</code>. The connection pool can shrink below this number if validation queries are received by the server.</p> <p>This value should be similar or near to the average number of requests that will be received by the server at the same time. By this setting, you can avoid having to open and close new connections every time a request is received by the server.</p>
poolstatements	true/ false	If the value is <code>true</code> , statement pooling is enabled for the pool.
testonborrow	true/ false	If the value is <code>true</code> , objects are validated before they are borrowed from the pool. If an object which fails the validation test is dropped from the pool and another one in the pool will be picked instead.
testwhileidle	true/ false	If the value is <code>true</code> , the objects in the pool will be validated using an idle object (if any exists). Any object which fails validation test would be dropped from the pool.
validationquery	String	The SQL query that will be used to validate connections from this pool before returning them to the caller.

The UI configuration of the DBLookup mediator further differs based on whether the connection to the database is made using an external datasource or a Carbon datasource. Click on the relevant tab to view the required UI configuration.

[External Datasource](#)[Carbon Datasource](#)

The screenshot shows the configuration interface for the DB Lookup Mediator. At the top, there are two radio buttons: 'Pool' (unchecked) and 'Data Source' (checked). Below that, 'Datasource Type' has 'External' (checked) and 'Carbon Datasource' (unchecked) options. The 'Initial Context' field is empty. The 'Data Source Name' field contains 'DBLookup'. The 'Url' field contains 'jdbc:mysql://localhost:3306/test'. The 'User' field contains 'root'. The 'Password' field contains 'root'. A 'Properties' section with an 'Add Property' button is present. An 'SQL Statements' section with an 'Add Statement' button is also present. A large 'Update' button is at the bottom.

The parameters available to configure the DBLookup mediator are as follows.

Parameter Name	Description
<b>Connection Information</b>	This parameter is used to specify whether the connection should be taken from a connection pool or a datasource.
<b>Datasource Type</b>	This parameter is used to specify whether the connection to the database should be made using an external datasource or a Carbon datasource.
<b>Initial Context</b>	The initial context factory class. The corresponding Java environment property is <code>java.naming.factory.initial</code> .
<b>Datasource Name</b>	The naming service provider URL . The corresponding Java environment property is <code>java.naming.provider.url</code> .
<b>URL</b>	JDBC URL of the database that data will be looked up from.
<b>User</b>	The user name used to connect to the database.
<b>Password</b>	The password used to connect to the database.

Adding properties to the DBLookup mediator

If you click **Add Property**, the page will expand to display the following parameters.

Properties		
Name	Value	Action
Select A Property		Delete
<a href="#"> Add Property</a>		

The parameters available to manage properties are as follows.

Parameter Name	Description
<b>Name</b>	The name of the property.
<b>Value</b>	The value of the property.
<b>Action</b>	This parameter enables a property to be deleted.

The available properties are as follows.

Name	Description	
autocommit	true / false	The auto-commit state of the connections created by the pool.
isolation	Connection.TRANSACTION_NONE / Connection.TRANSACTION_READ_COMMITTED / Connection.TRANSACTION_READ_UNCOMMITTED / Connection.TRANSACTION_REPEATABLE_READ / Connection.TRANSACTION_SERIALIZABLE	The isolation state of the connections created by the pool.
initialsize	int	The initial number of connections created when the pool is started.
maxactive	int	The maximum number of active connections that can be allocated from this pool at a given time. When this maximum limit is reached, no more active connections will be created by the connection pool. Specify 0 or a negative value if you do not want to set a limit.

maxidle	int	The maximum number of idle connections to be allowed in the connection pool at a given time. Specify 0 or a negative value if you want the pool to wait indefinitely.
maxopenstatements	int	The maximum number of open statements that can be allocated from the statement pool at a given time. When this maximum limit is reached, no more new statements will be created by the statement pool. Specify 0 or a negative value if you do not want to set a limit.
maxwait	long	The maximum number of milliseconds that the connection pool will wait for a connection to return before throwing an exception when there are no connections available in the pool. Specify 0 or a negative value if you want the pool to wait indefinitely.
minidle	int	The minimum number of idle connections to be allowed in the connection pool at a given time. Specify 0 or a negative value if you want the pool to wait indefinitely.
poolstatements	true/ false	If the value is true, statement pooling is enabled for the pool.
testonborrow	true/ false	If the value is true, objects are validated before they are borrowed from the pool. An object which fails the validation test will be dropped from the pool and another object in the pool will be picked instead.

testwhileidle	true/ false	If the value is true, the objects in the pool will be validated using an idle object evictor (if any exists). Any object which fails this validation test would be dropped from the pool.
validationquery	String	The SQL query that will be used to validate connections from this pool before returning them to the caller.

Mediator [switch to source view](#)

[Help](#)

### DB Lookup Mediator

Connection Information  Pool  Data Source

Datasource Type  External  Carbon Datasource

JNDI name

**SQL Statements**

[+ Add Statement](#)

The parameters available to configure the DBLookup mediator are as follows.

Parameter Name	Description
<b>Connection Information</b>	This parameter is used to specify whether the connection should be taken from a connection pool or a datasource.
<b>Datasource Type</b>	This parameter is used to specify whether the connection to the database should be made using an external datasource or a Carbon datasource.
<b>JNDI Name</b>	The JNDI used to look up data. See <a href="#">Configuring a JNDI Datasource</a> for more information.

### Adding SQL statements to the DBLookup Mediator

If you click **Add Statement**, the page will be expanded to display the following parameters.

### SQL Statements

[+ Add Statement](#)

SQL*	<input type="text"/>	<a href="#"> Delete</a>
------	----------------------	-------------------------

#### Parameters

[+ Add Parameter](#)

Parameter Type	Property Type	Value/Expression	Namespace	Action
CHAR	Expression	<input type="text"/>	<a href="#"> Namespaces</a>	<a href="#"> Delete</a>

#### Results

[+ Add Result](#)

Result Name	Column	Action
<input type="text"/>	<input type="text"/>	<a href="#"> Delete</a>

[Update](#)

Parameter Name	Description
<b>SQL</b>	This parameter is used to enter one or more SQL statements.
<b>Parameters</b>	This section is used to specify how the values of parameters in the SQL will be determined. A parameter value can be static or calculated at runtime based on a given expression.
<b>Parameter Type</b>	<p>The data type of the parameter. Possible values are as follows.</p> <ul style="list-style-type: none"> <li>• CHAR</li> <li>• VARCHAR</li> <li>• NUMERIC</li> <li>• DECIMAL</li> <li>• BIT</li> <li>• TINYINT</li> <li>• SMALLINT</li> <li>• INTEGER</li> <li>• BIGINT</li> <li>• REAL</li> <li>• DOUBLE</li> <li>• DATE</li> <li>• TIME</li> <li>• TIMESTAMP</li> </ul>
<b>Property Type</b>	<p>This determines whether the parameter value should be a static value or calculated at run time via an expression.</p> <ul style="list-style-type: none"> <li>• <b>Value:</b> If this is selected, a static value would be considered as the property value and this value should be entered in the <b>Value/Expression</b> parameter.</li> <li>• <b>Expression:</b> If this is selected, the property value will be determined during mediation by evaluating an expression. This expression should be entered in the <b>Value/Expression</b> parameter.</li> </ul>

<b>Value/Expression</b>	This parameter is used to enter the static value or the XPath expression used to determine the property value based on the option you selected for the <b>Property Type</b> parameter.
<p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p>	
<b>Action</b>	This allows you to delete a parameter.
<b>Results</b>	<p>This section is used to specify how to deal with the rerun result from a Database query execution.</p> <ul style="list-style-type: none"> <li>• <b>Result Name</b></li> <li>• <b>Column</b></li> <li>• <b>Action</b> - Deletes the result.</li> </ul>

## Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.

Mediator  switch to source view

## Examples

```
<dblookup xmlns="http://ws.apache.org/ns/synapse">
 <connection>
 <pool>
 <driver>org.apache.derby.jdbc.ClientDriver</driver>
 <url>jdbc:derby://localhost:1527/esbdb;create=false</url>
 <user>esb</user>
 <password>esb</password>
 </pool>
 </connection>
 <statement>
 <sql>select * from company where name =?</sql>
 <parameter expression="//m0:getQuote/m0:request/m0:symbol"
 xmlns:m0="http://services.samples/xsd" type="VARCHAR" />
 <result name="company_id" column="id"/>
 </statement>
</dblookup>
```

In this example, when a message is received by a proxy service with a DBLookup mediator configuration, it opens a connection to the database and executes the SQL query. The SQL query uses ? character for attributes that will be filled at runtime. The parameters define how to calculate the value of those attributes at runtime. In this sample, the DBLookup Mediator has been used to extract the id of the company from the company database using the symbol which is evaluated using an XPath against the SOAP envelope.

## Samples

- Sample 360: Introduction to DBLookup Mediator
- Sample 362: DBReport and DBLookup Mediators Together

- Sample 363: Reusable Database Connection Pools

## DB Report Mediator

The **DB Report Mediator** is similar to the [DBLookup Mediator](#). The difference between the two mediators is that the DB Report mediator writes information to a database using the specified insert SQL statement.

The DB Report mediator is a [content-aware](#) mediator.

---

[Syntax](#) | [UI Configuration](#) | [Example](#)

---

### Syntax

The syntax of the DB Report mediator changes depending on whether you connect to the database using a connection pool, or using a data source. Click on the relevant tab to view the required syntax.

Connection PoolData source

```
<dbreport>
 <connection>
 <pool>
 (
 <driver/>
 <url/>
 <user/>
 <password/>

 <dsName/>
 <icClass/>
 <url/>
 <user/>
 <password/>
)
 <property name="name" value="value" />*
 </pool>
 </connection>
 <statement>
 <sql>insert into something values(?, ?, ?, ?, ?)</sql>
 <parameter [value="" | expression=""]
 type="CHAR|VARCHAR|LONGVARCHAR|NUMERIC|DECIMAL|BIT|TINYINT|SMALLINT|INTEGER|BIGINT|REAL|FLOAT|DOUBLE|DATE|TIME|TIMESTAMP" />*
 </statement>+
</dbreport>
```

The syntax of the DBLookup mediator further differs based on whether the connection to the database is made using an external datasource or a Carbon datasource. Click on the relevant tab to view the required syntax.

External DatasourceCarbon Datasource

```

<dbreport>
 <connection>
 <pool>
 <dsName/>
 <icClass/>
 <url/>
 <user/>
 <password/>
 <property name="name" value="value"/>*
 </pool>
 </connection>
 <statement>
 <sql>select something from table where something_else = ?</sql>
 <parameter [value="" | expression=""]
type="CHAR|VARCHAR|LONGVARCHAR|NUMERIC|DECIMAL|BIT|TINYINT|SMALLINT|INTEGER|BIGINT|REA
L|FLOAT|DOUBLE|DATE|TIME|TIMESTAMP"/>*
 </statement>+
</dbreport>

```

```

<dbreport>
 <connection>
 <pool>
 <dsName/>
 </pool>
 </connection>
 <statement>
 <sql>select something from table where something_else = ?</sql>
 <parameter [value="" | expression=""]
type="CHAR|VARCHAR|LONGVARCHAR|NUMERIC|DECIMAL|BIT|TINYINT|SMALLINT|INTEGER|BIGINT|REA
L|FLOAT|DOUBLE|DATE|TIME|TIMESTAMP"/>*
 </statement>+
</dbreport>

```

## UI Configuration

The UI of the DBQuery mediator changes depending on whether you connect to the database using a connection pool, or using a data source. Click on the relevant tab to view the required UI.

[Pool](#)

The following UI is displayed when you select the **Pool** option for the **Connection Information** parameter, indicating that you want the connection to be made via a connection pool.

The screenshot shows the configuration interface for the DB Report Mediator. At the top, there are two radio buttons: 'True' (unchecked) and 'False' (checked). Below this, there are two more radio buttons: 'Pool' (checked) and 'Data Source'. The 'Driver' field is empty. The 'Url' field contains the URL 'jdbc:mysql://localhost:3306/test'. The 'User' field contains 'root'. The 'Password' field contains 'root'. A 'Properties' section has a 'Add Property' button. An 'SQL Statements' section has a 'Add Statement' button. At the bottom, there are three buttons: 'Save & Close', 'Save', and 'Cancel'.

The parameters available to configure the DB Report mediator are as follows.

Parameter Name	Description
Use Transaction	This parameter specifies whether the database operation should be performed within a transaction or not. Click <b>Yes</b> or <b>No</b> as relevant.
Driver	The class name of the database driver.
Url	The JDBC URL of the database that data will be written to.
User	The user name used to connect to the database.
Password	The password used to connect to the database.

#### Adding properties to the DB Report mediator

If you click **Add Property**, the page will expand to display the following parameters.

The screenshot shows a 'Properties' section with a table. The table has columns for 'Name', 'Value', and 'Action'. There is one row with a 'Select A Property' dropdown, an empty 'Value' input field, and a 'Delete' button. Below the table is a 'Add Property' button.

The parameters available to manage properties are as follows.

Parameter Name	Description
Name	The name of the property.
Value	The value of the property.
Action	This parameter enables a property to be deleted.

The available properties are as follows.

Name	Value	Description
autocommit	true / false	The auto-commit state of the connections created by the pool.
isolation	Connection.TRANSACTION_NONE / Connection.TRANSACTION_READ_COMMITTED / Connection.TRANSACTION_READ_UNCOMMITTED / Connection.TRANSACTION_REPEATABLE_READ / Connection.TRANSACTION_SERIALIZABLE	The isolation state of the connections created by the pool.
initialsize	int	The initial number of connections created when the pool is started.
maxactive	int	The maximum number of active connections that can be allocated from this pool at a given time. When this maximum limit is reached, no more active connections will be created by the connection pool. Specify 0 or a negative value if you do not want to set a limit.
maxidle	int	The maximum number of idle connections to be allowed in the connection pool at a given time. Specify 0 or a negative value if you want the pool to wait indefinitely.

maxopenstatements	int	The maximum number of open statements that can be allocated from the statement pool at a given time. When this maximum limit is reached, no more new statements will be created by the statement pool. Specify 0 or a negative value if you do not want to set a limit.
maxwait	long	The maximum number of milliseconds that the connection pool will wait for a connection to return before throwing an exception when there are no connections available in the pool. Specify 0 or a negative value if you want the pool to wait indefinitely.
minidle	int	The minimum number of idle connections to be allowed in the connection pool at a given time. Specify 0 or a negative value if you want the pool to wait indefinitely.
poolstatements	true/ false	If the value is true, statement pooling is enabled for the pool.
testonborrow	true/ false	If the value is true, objects are validated before they are borrowed from the pool. An object which fails the validation test will be dropped from the pool and another object in the pool will be picked instead.
testwhileidle	true/ false	If the value is true, the objects in the pool will be validated using an idle object evictor (if any exists). Any object which fails this validation test would be dropped from the pool.
validationquery	String	The SQL query that will be used to validate connections from this pool before returning them to the caller.

The UI configuration of the DBLookup mediator further differs based on whether the connection to the database is made using an external datasource or a Carbon datasource. Click on the relevant tab to view the required UI configuration.

ExternalCarbon Datasource

The following UI is displayed if you select the **External** option for the **Datasource Type** parameter, indicating that you want the connection to the database to be made using an external datasource.

Mediator [switch to source view](#)

**DB Report Mediator**

UseTransaction  True  False

Connection Information  Pool  Data Source

Datasource Type  External  Carbon Datasource

Initial Context

Data Source Name \*

Url \*

User \*

Password \*

**Properties**

[+ Add Property](#)

**SQL Statements**

[+ Add Statement](#)

**Update**

**Save & Close** **Save** **Cancel**

The parameters available to configure the DB Report mediator are as follows.

Parameter Name	Description
<b>Use Transaction</b>	This parameter specifies whether the database operation should be performed within a transaction or not. Click <b>Yes</b> or <b>No</b> as relevant.
<b>Initial Context</b>	The initial context factory class. The corresponding Java environment property is <code>java.naming.factory.initial</code> .
<b>Datasource Name</b>	The naming service provider URL . The corresponding Java environment property is <code>java.naming.provider.url</code> .

<b>URL</b>	The JDBC URL of the database that data will be written to.
<b>User</b>	The user name used to connect to the database.
<b>Password</b>	The password used to connect to the database.

Adding properties to the DB Report mediator

If you click **Add Property**, the page will expand to display the following parameters.

Properties		
Name	Value	Action
Select A Property		Delete
<a href="#"> Add Property</a>		

The parameters available to manage properties are as follows.

Parameter Name	Description
<b>Name</b>	The name of the property.
<b>Value</b>	The value of the property.
<b>Action</b>	This parameter enables a property to be deleted.

The available properties are as follows.

Name	Value	Description
autocommit	true / false	The auto-commit state of the connections created by the pool.
isolation	Connection.TRANSACTION_NONE / Connection.TRANSACTION_READ_COMMITTED / Connection.TRANSACTION_READ_UNCOMMITTED / Connection.TRANSACTION_REPEATABLE_READ / Connection.TRANSACTION_SERIALIZABLE	The isolation state of the connections created by the pool.
initialsize	int	The initial number of connections created when the pool is started.

maxactive	int	The maximum number of active connections that can be allocated from this pool at a given time. When this maximum limit is reached, no more active connections will be created by the connection pool. Specify 0 or a negative value if you do not want to set a limit.
maxidle	int	The maximum number of idle connections to be allowed in the connection pool at a given time. Specify 0 or a negative value if you want the pool to wait indefinitely.
maxopenstatements	int	The maximum number of open statements that can be allocated from the statement pool at a given time. When this maximum limit is reached, no more new statements will be created by the statement pool. Specify 0 or a negative value if you do not want to set a limit.
maxwait	long	The maximum number of milliseconds that the connection pool will wait for a connection to return before throwing an exception when there are no connections available in the pool. Specify 0 or a negative value if you want the pool to wait indefinitely.
minidle	int	The minimum number of idle connections to be allowed in the connection pool at a given time. Specify 0 or a negative value if you want the pool to wait indefinitely.
poolstatements	true/ false	If the value is true, statement pooling is enabled for the pool.

testonborrow	true/ false	If the value is true, objects are validated before they are borrowed from the pool. An object which fails the validation test will be dropped from the pool and another object in the pool will be picked instead.
testwhileidle	true/ false	If the value is true, the objects in the pool will be validated using an idle object evictor (if any exists). Any object which fails this validation test would be dropped from the pool.
validationquery	String	The SQL query that will be used to validate connections from this pool before returning them to the caller.

The following UI is displayed if you select the **Carbon Datasource** option for the **Datasource Type** parameter, indicating that you want the connection to the database to be made using an Carbon datasource.

Mediator [switch to source view](#)

DB Report Mediator

UseTransaction  True  False

Connection Information  Pool  Data Source

Datasource Type  External  Carbon Datasource

Data Source Name --SELECT--

**SQL Statements**

[+ Add Statement](#)

**Update**

**Save & Close** **Save** **Cancel**

Parameter Name	Description
Use Transaction	This parameter specifies whether the database operation should be performed within a transaction or not. Click <b>Yes</b> or <b>No</b> as relevant.

<b>Datasource</b>	This parameter is used to select a specific Carbon datasource you want to use to make the connection. All the Carbon datasources which are currently available are included in the list.
-------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Adding SQL statements to the DB Report Mediator

If you click **Add Statement**, the page will be expanded to display the following parameters.

**SQL Statements**

[+ Add Statement](#)

SQL*	<input type="text"/>	<a href="#"> Delete</a>
------	----------------------	-------------------------

**Parameters**

[+ Add Parameter](#)

Parameter Type	Property Type	Value/Expression	Namespace	Action
CHAR	Expression	<input type="text"/>	<a href="#"> Namespaces</a>	<a href="#"> Delete</a>

[Update](#)

Parameter Name	Description
<b>SQL</b>	This parameter is used to enter one or more SQL statements.
<b>Parameters</b>	This section is used to specify how the values of parameters in the SQL will be determined. A parameter value can be static or calculated at runtime based on a given expression.
<b>Parameter Type</b>	<p>The data type of the parameter. Possible values are as follows.</p> <ul style="list-style-type: none"> <li>• CHAR</li> <li>• VARCHAR</li> <li>• NUMERIC</li> <li>• DECIMAL</li> <li>• BIT</li> <li>• TINYINT</li> <li>• SMALLINT</li> <li>• INTEGER</li> <li>• BIGINT</li> <li>• REAL</li> <li>• DOUBLE</li> <li>• DATE</li> <li>• TIME</li> <li>• TIMESTAMP</li> </ul>
<b>Property Type</b>	<p>This determines whether the parameter value should be a static value or calculated at runtime via an expression.</p> <ul style="list-style-type: none"> <li>• <b>Value:</b> If this is selected, a static value would be considered as the property value and this value should be entered in the <b>Value/Expression</b> parameter.</li> <li>• <b>Expression:</b> If this is selected, the property value will be determined during mediation by evaluating an expression. This expression should be entered in the <b>Value/Expression</b> parameter.</li> </ul>

<b>Value/Expression</b>	This parameter is used to enter the static value or the XPath expression used to determine the property value based on the option you selected for the <b>Property Type</b> parameter.
You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.	
<b>Action</b>	This allows you to delete a parameter.

## Note

You can configure the mediator using XML. Click **switch to source** view in the **Mediator** window.



## Example

This example demonstrates simple database write operations. The DB Report mediator writes to a table using the details of the message. It updates the stock price of the company using the last quote value, which is calculated by evaluating an XPath expression against the response message.

```
<dbreport xmlns="http://ws.apache.org/ns/synapse">
 <connection>
 <pool>
 <driver>org.apache.derby.jdbc.ClientDriver</driver>
 <url>jdbc:derby://localhost:1527/esbdb;create=false</url>
 <user>esb</user>
 <password>esb</password>
 </pool>
 </connection>
 <statement>
 <sql>update company set price=? where name =?</sql>
 <parameter expression="//m0:return/m1:last/child::text()">
 <!-- Expression to calculate the last quote value -->
 <!-- Using m0 and m1 as aliases for the response message -->
 <!-- Using child::text() to get the text value of the last child node -->
 </parameter>
 <!-- XML namespaces for the response message -->
 <!-- m0: http://services.samples -->
 <!-- m1: http://services.samples/xsd -->
 </statement>
</dbreport>
```

## Samples

For more examples of the DB Report mediator, see:

- [Sample 361: Introduction to DB Report Mediator](#)
- [Sample 362: DB Report and DBLookup Mediators Together](#)
- [Sample 363: Reusable Database Connection Pools](#)
- [Sample 271: File Processing \(moves files into a database using the VFS transport and the DB Report mediator\)](#)

## Drop Mediator

The **Drop Mediator** stops the processing of the current message. This mediator is useful for ensuring that the message is sent only once and then dropped by the ESB. If you have any mediators defined after the <drop/> element, they will not be executed, because <drop/> is considered to be the end of the message flow.

When the Drop mediator is within the `In` sequence, it sends an HTTP 202 Accepted response to the client when it stops the message flow. When the Drop mediator is within the `Out` sequence before the Send mediator, no response is sent to the client.

The Drop mediator is a [content-unaware mediator](#).

---

[Syntax](#) | [UI Configuration](#) | [Example](#)

---

### Syntax

The drop token refers to a <drop> element, which is used to stop further processing of a message:

```
<drop/>
```

### UI Configuration

As with other mediators, after adding the drop mediator to a sequence, you can click its up and down arrows to move its location in the sequence.

#### Note

You can configure the mediator using XML. Click **switch to source** view in the **Mediator** window.



### Example

You can use the drop mediator for messages that do not meet the filter criteria in case the client is waiting for a response to ensure the message was received by the ESB. For example:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <!-- filtering of messages with XPath and regex matches -->
 <filter source="get-property('To')" regex=".*/StockQuote.*">
 <then>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </then>
 <else>
 <drop/>
 </else>
 </filter>
 ...

```

In this scenario, if the message doesn't meet the filter condition, it is dropped, and the HTTP 202 Accepted response is sent to the client. If you did not include the drop mediator, the client would not receive any response.

## EJB Mediator

The **EJB mediator** calls an external Enterprise JavaBean(EJB) and stores the result in the message payload or in a message context property. Currently, this mediator supports EJB3 Stateless Session Beans and Stateful Session Beans.

The EJB mediator is a [content-aware](#) mediator.

---

## Syntax | UI Configuration | Example

---

### Syntax

```

<ejb beanstalk="string" class="string" [sessionId="string"] [remove="true | false"]
[method="string"] [target="string | {xpath}"] [jndiName="string"] />
 <args>
 <arg (value="string | {xpath}") />*
 </args>
</ejb>

```

---

### UI Configuration

Mediator [switch to source view](#)

[Help](#)

### EJB Mediator

Beanstalk\*

Class \*

Method \*

Session Id

Remove

Target

JNDI Name\*

**Method Arguments**

[+ Add argument](#)

Parameter Name	Description
Beanstalk ID	Reference to the application server specific connection source information, which is defined at the synapse.properties.
Class	This required the remote interface definition provided in the EJB 3.0 (EJB service invocation remote/home interface).
Session ID	When the EJB context is invoked in the form state-full bean then the related ejb session status specified will be stored in here. Possible values are as follows. <ul style="list-style-type: none"> <li>• <b>Value:</b> If this is selected, the session ID can be entered as a static value.</li> <li>• <b>Expression:</b> If this is selected, an XPath expression can be entered to evaluate the session ID.</li> </ul>
Remove	This parameter specifies whether the Enterprise Entity Manager should remove the EJB context related parameters once the state full/stateless session is invoked.
Target	If a particular EJB method returns, then the return object can be saved against the the name provided in the target at the synapse property context.
JNDI Name	The Java Naming and Directory Interface (JNDI) is an application programming interface (API) for accessing different kinds of naming and directory services. JNDI is not specific to a particular naming or directory service. It can be used to access many different kinds of systems including file systems; distributed objects systems such as CORBA, Java RMI, and EJB; and directory services such as LDAP, Novell NetWare, and NIS+.
Add Argument	Can be used to define the arguments which is required for the particular ejb method to be invoked Expression/Value.

**Tip**

You can click the "Namespaces" link to add namespaces if you are providing an expression. You will be provided another panel named "Namespace Editor" where you can provide any number of namespace prefixes and the URL used in the XPath expression.

**Note**

You can configure the Mediator using XML. Click on "switch to source view" in the "Mediator" window.

Mediator  switch to source view

**Example**

```
<ejb beanstalk="jack" class="org.ejb.wso2.test.StoreRegister" method="getStoreById"
target="store"
jndiName="ejb:/EJBDemo/StoreRegsiterBean!org.ejb.wso2.test.StoreRegister">
<args>
<arg xmlns:ns="http://org.apache.synapse/xsd"
xmlns:ns3="http://org.apache.synapse/xsd" value="{get-property('loc_id')}"/>
</args>
</ejb>
```

In this example, the EJB Mediator does the EJB service invocation by calling **getStoreById** published at the application server and exposed via `ejb:/EJBDemo/StoreRegsiterBean!org.ejb.wso2.test.StoreRegister`,

then response will be assigned to the **target** specified (variable/expression).

**Enqueue Mediator**

The **Enqueue mediator** uses a **priority executor** to handle messages. Priority executors are used in high-load scenarios when you want to execute different sequences for messages with different priorities. This approach allows you to control the resources allocated to executing sequences and to prevent high-priority messages from getting delayed and dropped. For example, if there are two priorities with value 10 and 1, messages with priority 10 will get 10 times more resources than messages with priority 1.

The Enqueue mediator is a **content-unaware** mediator.

**Syntax | UI Configuration | Example****Syntax**

```
<enqueue xmlns="http://ws.apache.org/ns/synapse"
executor="[Priority Executor Name]" priority="[Priority]" sequence="[Sequence Name]">
</enqueue>
```

## UI Configuration

**Enqueue Mediator**

Executor *	<input type="text"/>
Priority *	<input type="text" value="0"/>
Sequence *	<input type="text"/>
<a href="#">Configuration Registry</a> <a href="#">Governance Registry</a>	
<input type="button" value="Update"/>	

The parameters available to configure the Enqueue mediator are as follows.

Parameter Name	Description
<b>Executor</b>	The <b>priority executor</b> that should be used to process the messages.
<b>Priority</b>	The priority of messages that this mediator will handle. This priority level should also be defined in the priority executor.
<b>Sequence</b>	The <b>sequence</b> that should be used to process messages with the specified priority. This <b>sequence</b> should be saved in the <b>registry</b> before it can be selected here. Click either <b>Configuration Registry</b> or the <b>Governance Registry</b> to select the required <b>sequence</b> from the resource tree.

### Note

You can configure the mediator using XML. Click **switch to source** view in the **Mediator** window.

Mediator  switch to source view

## Example

In this example, two Enqueue mediator configurations use the **priority executor** One based on which requests are prioritized. A **sequence** named **Send** applies to requests with priority 2, and a **sequence** named **LogSend** applies to priority 1.

```
<inSequence xmlns="http://ws.apache.org/ns/synapse">
 <enqueue executor="One" priority="2" sequence="conf:/Send"></enqueue>
 <enqueue executor="One" priority="1" sequence="conf:/LogSend"></enqueue>
</inSequence>
```

## Samples

For another example, see [Sample 652: Priority Based Message Mediation](#).

### Enrich Mediator

The **Enrich Mediator** can process a message based on a given source configuration and then perform the specified action on the message by using the target configuration. It gets an OMElement using the configuration specified in the source and then modifies the message by putting it on the current message using the configuration in the target.

The Enrich mediator is a **content-aware mediator**.

## Syntax | UI Configuration | Examples

### Syntax

```
<enrich>
 <source [clone=true|false] [type=custom|envelope|body|property|inline] xpath="" property="" />
 <target [action=replace|child|sibling]
 [type=custom|envelope|body|property|inline] xpath="" property="" />
</enrich>
```

### UI Configuration

The main properties of the Enrich Mediator available are:

- **Source**
- **Target**

**Enrich Mediator** ? Help

**Source**

Clone \*

Type \*

XPath Expression \*   Namespaces

**Target**

Action \*

Type \*

XPath Expression \*   Namespaces

**Update**

Source Configuration

The following properties are available:

- **Clone**- By setting the clone configuration, the message can be cloned or used as a reference during enriching. The default value is false.
  - **True**
  - **False**
- **Type**- The type that the mediator uses from the original message to enrich the modified message that passes through the mediator.
  - **Custom** - Custom XPath value.
  - **Envelope** - Envelope of the original message used for enriching.
  - **Body** - Body of the original message used for enriching.

- **Property** - Specifies a property.
- **Inline** - Specifies an inline XML value.
- **XPath Expression** - This field is used to specify the custom XPath value if you selected **custom** for the **Type** field.

**Tip**

You can click the Namespaces link to add [namespaces](#) if you are providing an expression. You will be provided another panel named "Namespace Editor," where you can provide any number of namespace prefixes and URL that you have used in the XPath expression.

**Target Configuration**

The following properties are available:

- **Action**- By specifying the action type, the relevant action can be applied to outgoing messages.
  - **Replace** - Replace is the default value of *Action*. It will be used if a specific value for *Action* is not given. This replaces the XML message based on the target type specified on the target configuration.
  - **Child** - Adding as a child of the specified target type.
  - **Sibling** - Adding as a sibling of the specified target type.

Type and XPath Expression - Refer to "Source Configuration" above.

**Note**

You can configure the Mediator using XML. Click on "switch to source view" in the "Mediator" window.



Mediator  switch to source view

**Examples*****Example 1: Setting the property symbol***

In this example, you are setting the property symbol. Later, you can log it using the [Log Mediator](#).

```
<enrich xmlns="http://ws.apache.org/ns/synapse">
 <source clone="false" type="envelope"/>
 <target type="property" property="payload" />
</enrich>
```

***Example 2: Adding a child object to a property***

In this example, you add a child property named Lamborghini to a property named Cars. The configuration for this is as follows:

```

<proxy xmlns="http://ws.apache.org/ns/synapse" name="_TestEnrich"
transports="https,http" statistics="disable" trace="enable" startOnLoad="true">
 <target>
 <inSequence>
 <enrich>
 <source type="inline" clone="true">
 <Cars/>
 </source>
 <target type="property" property="Cars" />
 </enrich>
 <log level="custom">
 <property name="PokeCarListBeforeEnrich"
expression="get-property('Cars')"/>
 </log>
 <enrich>
 <source type="inline" clone="true">
 <Car>Lamborghini</Car>
 </source>
 <target action="child" xpath="$ctx:Cars" />
 </enrich>
 <log level="custom">
 <property name="PokeCarListAfterEnrich"
expression="get-property('Cars')"/>
 </log>
 </inSequence>
 <outSequence/>
 </target>
 <description></description>
</proxy>

```

### **Example 3 - Adding a SOAPEnvelope type object as a property to a message**

In this example, you add the SOAP envelope in a SOAP request as a property to a message. The Enrich mediator is useful in this scenario since adding the property directly using the [Property mediator](#) results in the SOAPEnvelope object being created as an OM type object. The OM type object created cannot be converted back to a SOAPEnvelope object.

```

<enrich>
 <source type="envelope" clone="true" />
 <target type="property" property="ExtractedEnvelope" />
</enrich>

```

### **Example 4 - Preserving the original payload**

In this example, you copy the original payload to a property using the Enrich mediator.

```

<enrich>
 <source clone="false" type="body" />
 <target action="replace" type="property" property="ORIGINAL_PAYLOAD" />
</enrich>

```

Then whenever you need the original payload, you replace the message body with this property value using the Enrich mediator as follows:

```
<enrich>
 <source clone="false" type="property" property="ORIGINAL_PAYLOAD" />
 <target action="replace" type="body" />
</enrich>
```

For other example using the Enrich mediator, see [Sample 15: Using the Enrich Mediator for Message Copying and Content Enrichment](#) and [Sample 440: Converting JSON to XML Using XSLT](#).

## Entitlement Mediator

The **Entitlement Mediator** intercepts requests and evaluates the actions performed by a user against an eXtensible Access Control Markup Language (XACML) policy. WSO2 Identity Server can be used as the XACML Policy Decision Point (PDP) where the policy is set, and WSO2 ESB serves as the XACML Policy Enforcement Point (PEP) where the policy is enforced.

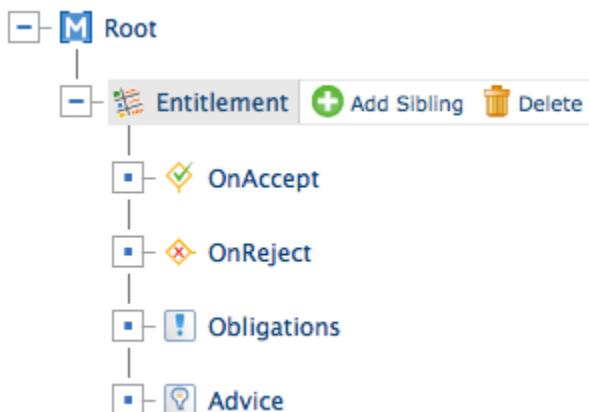
[Syntax](#) | [UI configuration](#) | [Example](#)

## Syntax

```
<entitlementService remoteServiceUrl="" remoteServiceUserName=""
remoteServicePassword=""
callbackClass="org.wso2.carbon.identity.entitlement.mediator.callback.[UTEntitlementCallbackHandler|X509EntitlementCallbackHandler|SAMLEntitlementCallbackHandler|KerberosEntitlementCallbackHandler]"
client="soap|basicAuth|thrift|wsXacml">
 <onReject/>
 <onAccept/>
 <advice/>
 <obligations/>
</entitlementService>
```

## UI configuration

When you add the Entitlement mediator to a sequence, the Entitlement mediator node appears as follows with four sub elements. These sub elements are used to define a mediation sequence to be applied based on the entitlement result.



The following are descriptions for the four sub elements of the Entitlement mediator.

Parameter Name	Description
<b>OnAccept</b>	The sequence to execute when the result returned by the Entitlement mediator is <code>Permit</code> . For example, you can configure the sequence to direct the request to the back end server as requested.
<b>OnReject</b>	The sequence to execute when the result returned by the Entitlement mediator is <code>Deny</code> , <code>Not Applicable</code> or <code>Indeterminate</code> . For example, you can configure the sequence to respond to the client with the message <code>Unauthorized Request</code> .
<b>Obligations</b>	The sequence to execute when the XACML response contains an obligation statement. When this response is received, the Entitlement mediator clones the current message context, creates a new message context, adds the obligation statement to the SOAP body and then executes the sequence. Since the <b>Obligations</b> sequence is executed synchronously, the Entitlement mediator waits for a response. If the sequence returns a <code>true</code> value, the sequence defined for the <b>OnAccept</b> sub element is applied. If the sequence returns a <code>false</code> value, the sequence defined for the <b>OnReject</b> sub element is applied.
<b>Advice</b>	The sequence to execute when the XACML response contains an advice statement. When this response is received, the Entitlement mediator clones the current message context, creates a new message context, adds the advice statement to the SOAP body and then executes the sequence. Since the <b>Advice</b> sequence is executed asynchronously, the Entitlement mediator does not wait for a response.

The Entitlement mediator configuration screen appears below the tree as shown below.

Mediator [switch to source view](#)

### Entitlement Mediator

[Help](#)

Entitlement Server	<input type="text"/>
User Name	<input type="text"/>
Password	<input type="password"/>
Entitlement Callback Handler	<input checked="" type="radio"/> UT <input type="radio"/> X509 <input type="radio"/> SAML <input type="radio"/> Kerberos <input type="radio"/> Custom <input type="text"/> Custom Entitlement Callback Handler Class
Entitlement Service Client Type	<input checked="" type="radio"/> SOAP – Basic Auth (WSO2 IS 4.0.0 or later) <input type="radio"/> Thrift <input type="radio"/> SOAP – Authentication Admin (WSO2 IS 3.2.3 or earlier) <input type="radio"/> WS-XACML
Thrift Host	<input type="text"/>
Thrift Port	<input type="text"/>

---

**On Acceptance**

Specify as  In-Lined Sequence  Referring Sequence

---

**On Rejection**

Specify as  In-Lined Sequence  Referring Sequence

---

**Obligations**

Specify as  In-Lined Sequence  Referring Sequence

---

**Advice**

Specify as  In-Lined Sequence  Referring Sequence

---

**Update**

The parameters available for configuring the Entitlement mediator are as follows.

Parameter Name	Description
<b>Entitlement Server</b>	Server URL of the WSO2 Identity Server that acts as the PDP (e.g., <a href="https://localhost:9443/services">https://localhost:9443/services</a> ).
<b>User Name</b>	This user should have permissions to log in and manage configurations in the WSO2 Identity Server.
<b>Password</b>	The password of the username entered in the <b>User Name</b> parameter.

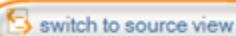
<b>Entitlement Callback Handler</b>	<p>The handler that should be used to get the subject (user name) for the XACML request.</p> <ul style="list-style-type: none"> <li><b>UT:</b> This class looks for the subject name in the Axis2 message context under the username property. This is useful when the UsernameToken security is enabled in WSO2 ESB for a proxy service, because when the user is authenticated for such a proxy service, the username would be set in the Axis2 message context. As a result, the Entitlement mediator would automatically get the subject value for the XACML request from there. This is the default callback class.</li> <li><b>X509:</b> Specify this class if the proxy is secured with X509 certificates.</li> <li><b>SAML:</b> Specify this class if the proxy is secured with WS-Trust.</li> <li><b>Kerberos:</b> Specify this class if the proxy is secured with Kerberos.</li> <li><b>Custom:</b> This allows you to specify a custom entitlement callback handler class.</li> </ul> <p>You can also set properties that control how the subject is retrieved; see <a href="#">Advanced Callback Properties</a>.</p>
<b>Entitlement Service Client</b>	<p>The method of communication to use between the PEP and the PDP. For SOAP, choose whether to use Basic Authentication (available with WSO2 Identity Server 4.0.0 and later) OR the AuthenticationAdmin service, which authenticates with the Entitlement service in Identity Server 3.2.3 and earlier. Thrift uses its own authentication service over TCP. WS-XACML uses Basic Authentication.</p> <p>The XACML standard refrains from specifying which method should be used to communicate from the PEP to the PDP, and many vendors have implemented a proprietary approach. There is a standard called "Web Services Profile of XACML (WS-XACML) Version 1.0, but it has not been widely adopted because of its bias toward SOAP and the performance implications from XML signatures. However, the benefit of adopting a standard is the elimination of vendor locking, because it will allow your current PEP to work even if you move to a PDP from another vendor (as long as the new PDP also supports this standard). Otherwise you may need to modify your existing PEP to adopt to the new PDP. WSO2 Identity Server has its proprietary SOAP API, Thrift API, and basic support for WS-XACML.</p>
<b>Thrift Host</b>	The host used to establish a Thrift connection with the Entitlement service when the Entitlement Service Client is set to Thrift.
<b>Thrift Port</b>	The port used to establish a Thrift connection with the Entitlement service when the Entitlement Service Client is set to Thrift. The default port is 10500.

You will now define the sequences you want to run for the entitlement results.

- If you want to specify an existing sequence for a result, click **Referring Sequence** for that result and select the sequence from the registry.
- If you want to define the sequence in the tree, leave **In-Lined Sequence** selected.
- Click **Update**.
- In the tree, click the first result node for which you want to define the sequence, and then add the appropriate mediators to create the sequence. Repeat for each result node.

## Note

You can also configure the Mediator using XML. Click **switch to source view** in the **Mediator** window.

Mediator  switch to source view

## Example

In the following example, the WSO2 Identity Server (with log in URL `https://localhost:9443/services`) is seen to authenticate the user invoking the secured backend service.

If the authorization test performed on a request sent to this URL fails, the [Fault mediator](#) converts the request into a fault message giving `Unauthorized` as the reason for the request to be rejected and `XACML Authorization Failed` as the detail. Then the [Respond mediator](#) sends the converted message back to the client.

If the user is successfully authenticated, the request is sent using the [Send Mediator](#) to the endpoint with the `http://localhost:8281/services/echo` URL.

```

<entitlementService remoteServiceUrl="https://localhost:9443/services"
 remoteServiceUserName="admin"
 remoteServicePassword=
"enc:kuv2MubUUveMyv6GeHrXr9i159ajJIqUI4eoYHcgGKf/BBFOWn96NTjJQI+wYbWjKW6r79S7L7ZzgYeWx
7D1Gbfff5X3pBN2Gh9yV0BHP1E93QtFqR7uTwI141Tr7V7ZwScwNqJbiNoV+vyLbsqKJE7T3nP8Ih9Y6omygbcL
cHzg="

callbackClass="org.wso2.carbon.identity.entitlement.mediator.callback.UTEntitlementCal
lbackHandler"
 client="basicAuth">

 <onReject>
 <makefault version="soap12">
 <code xmlns:soap12Env="http://www.w3.org/2003/05/soap-envelope"
 value="soap12Env:Receiver"/>
 <reason value="UNAUTHORIZED"/>
 <node/>
 <role/>
 <detail>XACML Authorization Failed</detail>
 </makefault>
 <respond/>
 </onReject>
 <onAccept>
 <send>
 <endpoint>
 <address uri="http://localhost:8281/services/echo"/>
 </endpoint>
 </send>
 </onAccept>
 <obligations/>
 <advice/>
</entitlementService>
```

## Advanced Callback Properties

The abstract `EntitlementCallbackHandler` class supports the following properties for getting the XACML subject (user name), specifying the action, and setting the service name. The various implementations of this class (`UTEntitlementCallbackHandler`, `X509EntitlementCallbackHandler`, etc.) can use some or all of these properties. You implement these properties by adding [Property mediators](#) before the Entitlement mediator in the sequence.

The default `UTEntitlementCallbackHandler` looks for a property called `username` in the Axis2 message context, which it uses as the XACML request `subject-id` value. Likewise, the other handlers look at various properties for values for the attributes and construct the XACML request. The following attribute IDs are used by the default

handlers.

- `urn:oasis:names:tc:xacml:1.0:subject:subject-id` of category `urn:oasis:names:tc:xacml:1.0:subject-category:access-subject`
- `urn:oasis:names:tc:xacml:1.0:action:action-id` of category `urn:oasis:names:tc:xacml:3.0:attribute-category:action`
- `urn:oasis:names:tc:xacml:1.0:resource:resource-id` of category `urn:oasis:names:tc:xacml:3.0:attribute-category:resource`
- `IssuerDN` of category `urn:oasis:names:tc:xacml:3.0:attribute-category:environment` (used only by X509 handler)
- `SignatureAlgorithm` of category `urn:oasis:names:tc:xacml:3.0:attribute-category:environment` (used only by X509 handler)

In most scenarios, you do not need to configure any of these properties.

Property name	Acceptable values	Scope	Description
<code>xacml_subject_identifier</code>	string	axis2	By default, the Entitlement mediator expects to find the XACML subject (user name) in a property called <code>username</code> in the message's Axis2 context. If your authentication mechanism specifies the user name by adding a property of a different name, create a property called <code>xacml_subject_identifier</code> and set it to the name of the property in the message context that contains the subject.
<code>xacml_action</code>	string	axis2	If you are using REST and want to specify a different HTTP verb to use with the service, specify it with the <code>xacml_action</code> property and specify the <code>xacml_use_rest</code> property to true.
<code>xacml_use_rest</code>	true/false	axis2	If you are using REST, and you want to override the HTTP verb to send with the request, you can set this property to true to set to true.
<code>xacml_resource_prefix</code>	string	axis2	If you want to change the service name, use this property to specify the new service name or the text you want to prepend to the service name.
<code>xacml_resource_prefix_only</code>	true/false	axis2	If set to true, the <code>xacml_resource_prefix</code> value is used as the whole service name. If set to false (default), the <code>xacml_resource_prefix</code> is prepended to the service name.

## Event Mediator

The **Event Mediator** redirects incoming events to the specified event topic. For more information, see [Working with Topics and Events](#).

The Event mediator is a content-aware mediator.

---

## Syntax | UI Configuration | Examples

---

### Syntax

```
<event xmlns="http://ws.apache.org/ns/synapse" topic="" [expression=""] />
```

## UI Configuration

The screenshot shows the 'Event Mediator' configuration screen. At the top, there's a 'Topic type' section with 'Static' selected. Below it are fields for 'Topic' and 'Expression'. To the right of the 'Expression' field is a 'Namespace' button. At the bottom are 'Update' and 'Help' buttons.

The parameters available for configuring the Event mediator are as follows:

Parameter Name	Description
<b>Topic Type</b>	<p>The type of topic. The available options are as follows.</p> <ul style="list-style-type: none"> <li><b>Static:</b> If this is selected, the topic to which the events are published is a static value.</li> <li><b>Dynamic:</b> If this is selected, the topic to which the events are published is a dynamic value that should be derived via an XPath expression.</li> </ul>
<b>Topic</b>	<p>The topic to which the events should be published. You can enter a static value or an XPath expression based on the option you selected for the <b>Topic Type</b> parameter.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <b>Tip</b>            You can click <b>NameSpaces</b> to add namespaces when you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.         </div>
<b>Expression</b>	<p>The XPath expression used to build the message to be published to the specified topic.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <b>Tip</b>            You can click <b>NameSpaces</b> to add namespaces when you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.         </div>

## Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.

Mediator  switch to source view

## Examples

In this example, when an event notification comes to the EventingProxy proxy service, they are processed by the PublicEventSource sequence, which logs the messages and publishes them to the topic SampleEventSource. Services that subscribe to the topic SampleEventSource will then receive these messages.

```
<!-- Simple Eventing configuration -->
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <sequence name="PublicEventSource" >
 <log level="full" />
 <event topic="SampleEventSource" />
 </sequence>

 <proxy name="EventingProxy">
 <target inSequence="PublicEventSource" />
 </proxy>
</definitions>
```

## Sample

See the following sample for another example.

[Sample 460: Introduction to Eventing and Event Mediator.](#)

## FastXSLT Mediator

The **FastXSLT Mediator** is similar to the [XSLT mediator](#), but it uses the [Streaming XPath Parser](#) and applies the XSLT transformation to the message stream instead of to the XML message payload. The result is a faster transformation, but you cannot specify the source, properties, features, or resources as you can with the XSLT mediator. Therefore, the FastXSLT mediator is intended to be used to gain performance in cases where the original message remains unmodified. Any pre-processing performed on the message payload will not be visible to the FastXSLT mediator, because the transformation logic is applied on the original message stream instead of the message payload. In cases where the message payload needs to be pre-processed, use the XSLT mediator instead of the FastXSLT mediator.

For example, if you are using the VFS transport to handle files, you might want to read the content of the file as a stream and directly send the content for XSLT transformation. If you need to pre-process the message payload, such as adding or removing properties, use the XSLT mediator instead.

In summary, following are the key differences between the XSLT and FastXSLT mediators:

XSLT Mediator	FastXSLT Mediator
Performs XSLT transformations on the message <b>payload</b> .	Performs XSLT transformations on the message <b>stream</b> .
The message is built before processing. Therefore, you can pre-process the message payload before the XSLT transformation.	The message is not built before processing. Therefore, any pre-processing on the message will not be reflected in the XSLT transformation.
The performance is slower than the FastXSLT mediator.	The performance is faster than the XSLT mediator.

To enable the FastXSLT mediator, your XSLT script must include the following parameter in the XSL output.

`omit-xml-declaration="yes"`

For example:

```
<xsl:output method="xml" omit-xml-declaration="yes" encoding="UTF-8" indent="yes" />
```

If you do not include this parameter in your XSLT when using the FastXSLT mediator, you will get the following error.

```
ERROR XSLTMediator Error creating XSLT transformer
```

The FastXSLT mediator is a [conditionally content-aware mediator](#).

## Syntax | UI Configuration | Example

### Syntax

```
<fastXSLT key="string" />
```

### UI Configuration

Mediator switch to source view

## FastXSLT Mediator

Key Type :  Static Key  Dynamic Key

Key\*  Configuration Registry Governance Registry

**Update**

The parameters available to configure the FastXSLT mediator are as follows.

Parameter Name	Description
<b>Key Type</b>	You can select one of the following options. <ul style="list-style-type: none"> <li>• <b>Static Key:</b> If this is selected, an existing key can be selected from the registry for the <b>Key</b> parameter.</li> <li>• <b>Dynamic Key:</b> If this is selected, the key can be entered dynamically in the <b>Key</b> parameter.</li> </ul>

<b>Key</b>	This specifies the registry key to refer the XSLT to. This supports static and dynamic keys.
	<p><b>Tip</b></p> <p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p>

<p>You can also configure the mediator using XML. Click <b>switch to source view</b> in the <b>Mediator</b> window.</p> 
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Example

```
<fastxslt xmlns="http://ws.apache.org/ns/synapse" key="transform/example.xslt"/>
```

In this example, the XSLT is specified by the key `transform/example.xslt`, which is used to transform the message stream.

## Fault Mediator

The **Fault Mediator** (also called the **Makefault Mediator**) transforms the current message into a fault message. However, this mediator does not send the converted message. The **Send Mediator** needs to be invoked to send a fault message created via the Fault mediator. The fault message's `To` header is set to the `Fault-To` of the original message (if such a header exists in the original message). You can create the fault message as a SOAP 1.1, SOAP 1.2, or plain-old XML (POX) fault.

For more information on faults and errors, see [Error Handling](#).

[Syntax](#) | [UI Configuration](#) | [Examples](#)

## Syntax

```
<makefault [version="soap11|soap12|pox"]>
 <code (value="literal" | expression="xpath")/>
 <reason (value="literal" | expression="xpath")>
 <node>?
 <role>?
 <detail>?
</makefault>
```

## UI Configuration

Click on the relevant tab to view the required UI configuration pending on whether you want to create the fault message as a SOAP 1.1 fault, SOAP 1.2 fault or a plain-old XML (POX) fault.

[SOAP 1.1](#)[SOAP 1.2](#)[Plain-old XML \(POX\)](#)

Mediator [switch to source view](#)

## Fault Mediator

Version  SOAP 1.1  SOAP 1.2  POX

Fault Code\*

Fault String\*  value  expression

Fault Actor

Detail

The parameters available to configure the Fault mediator to create a SOAP 1.1 fault are as follows.

Parameter Name	Description
<b>Fault Code</b>	<p>This parameter is used to select the fault code for which the fault string should be defined. Possible values are as follows.</p> <ul style="list-style-type: none"> <li><b>versionMismatch:</b> Select this to specify the fault string for a SOAP version mismatch.</li> <li><b>mustUnderstand:</b> Select this to specify the fault string for the <code>mustUnderstand</code> error in SOAP.</li> <li><b>Client:</b> Select this to specify the fault string for client side errors.</li> <li><b>Server:</b> Select this to specify the fault string for server side errors.</li> </ul>
<b>Fault String</b>	<p>The detailed fault string of the fault code. The following options are available.</p> <ul style="list-style-type: none"> <li><b>value:</b> If this option is selected, the fault string is specified as a string value.</li> <li><b>expression:</b> If this option is selected, the fault string is specified as an expression.</li> </ul> <p><b>Tip</b></p> <p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p>
<b>Fault Actor</b>	The element of the SOAP fault message which is used to capture the party which caused the fault.
<b>Detail</b>	This parameter is used to enter a custom description of the error.

Mediator [switch to source view](#)

## Fault Mediator

Version  SOAP 1.1  SOAP 1.2  POX

Code\*

Reason\*  value  expression

Role

Node

Detail

The parameters available to configure the Fault mediator to create a SOAP 1.2 fault are as follows.

Parameter Name	Description
<b>Code</b>	<p>This parameter is used to select the fault code for which the reason should be defined. Possible values are as follows.</p> <ul style="list-style-type: none"> <li>• <b>versionMismatch</b>: Select this to specify the reason for a SOAP version mismatch.</li> <li>• <b>mustUnderstand</b>: Select this to specify the reason for the <code>mustUnderstand</code> error in SOAP.</li> <li>• <b>dataEncodingUnknown</b>: Select this to specify the reason for a SOAP encoding error.</li> <li>• <b>Sender</b>: Select this to specify the reason for a sender-side error.</li> <li>• <b>Receiver</b>: Select this to specify the reason for a receiver-side error.</li> </ul>
<b>Reason</b>	<p>This parameter is used to specify the reason for the error code selected in the <b>Code</b> parameter. The following options are available.</p> <ul style="list-style-type: none"> <li>• <b>value</b>: If this option is selected, the reason is specified as a string value.</li> <li>• <b>expression</b>: If this option is selected, the reason is specified as an expression.</li> </ul> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p><b>Tip</b></p> <p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p> </div>

<b>Role</b>	The SOAP 1.1 role name.
<b>Node</b>	The SOAP 1.2 node name.
<b>Detail</b>	This parameter is used to enter a custom description of the error.

Mediator switch to source view

## Fault Mediator

Version  SOAP 1.1  SOAP 1.2  POX

Reason\*  value  expression

Detail

The parameters available to configure the Fault mediator to create a plain-old XML (POX) fault are as follows.

Parameter Name	Description
<b>Reason</b>	<p>This parameter is used to enter a custom fault message. The following options are available.</p> <ul style="list-style-type: none"> <li>• <b>value:</b> If this option is selected, the fault message is specified as a string value.</li> <li>• <b>expression:</b> If this option is selected, the fault message is specified as an expression.</li> </ul> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p><b>Tip</b></p> <p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p> </div>

<b>Detail</b>	<p>This parameter is used to enter details for the fault message. The following options are available.</p> <ul style="list-style-type: none"> <li>• <b>value:</b> If this option is selected, the detail is specified as a string value.</li> <li>• <b>expression:</b> If this option is selected, the detail is specified as an expression.</li> </ul>
	<b>Tip</b>

You can click **NameSpaces** to add namespaces if you are providing an expression. Then the **Namespace Editor** panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.

## Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.

## Examples

In the following example, the `test` message string value is given as the reason for the SOAP error `versionMismatch`.

```
<makefault xmlns="http://ws.apache.org/ns/synapse" version="soap11">
 <code xmlns:soap11Env="http://schemas.xmlsoap.org/soap/envelope/" value="soap11Env:VersionMismatch" />
 <reason value="test message " />
 <role></role>
</makefault>
```

## Samples

[Sample 5: Creating SOAP Fault Messages and Changing the Direction of a Message.](#)

## Filter Mediator

The **Filter Mediator** can be used for filtering messages based on an XPath, JSONPath or a regular expression. If the test succeeds, the Filter mediator executes the other mediators enclosed in the sequence.

The Filter Mediator closely resembles the "If-else" control structure.

The Filter mediator is a [conditionally content aware mediator](#).

[Syntax](#) | [UI Configuration](#) | [Example](#)

## Syntax

```
<filter (source="[XPath|json-eval(JSONPath)]" regex="string") |
xpath="[XPath|json-eval(JSONPath)]">
 mediator+
</filter>
```

This mediator could also be used to handle a scenario where two different sequences are applied to messages that meet the filter criteria and messages that do not meet the filter criteria.

```
<filter (source="[XPath|json-eval(JSONPath)]" regex="string") |
xpath="[XPath|json-eval(JSONPath)]">
 <then [sequence="string"]>
 mediator+
 </then>
 <else [sequence="string"]>
 mediator+
 </else>
</filter>
```

In this case, the Filter condition remains the same. The messages that match the filter criteria will be mediated using the set of mediators enclosed in the `then` element. The messages that do not match the filter criteria will be mediated using the set of mediators enclosed in the `else` element.

## UI Configuration

The parameters available for configuring the Filter mediator are as follows:

Parameter Name	Description
<b>Specify As</b>	<p>This is used to specify whether you want to specify the filter criteria via an XPath expression or a regular expression.</p> <ul style="list-style-type: none"> <li><b>XPath:</b> If this option is selected, the Filter mediator tests the given XPath/JSONPath expression as a Boolean expression. When specifying a JSONPath, use the format <code>json-eval(&lt;JSON_PA TH&gt;)</code>, such as <code>json-eval(getQuote.request.symbol)</code>. For more information on using JSON with the ESB, see <a href="#">JSON Support</a>.</li> <li><b>Source and Regular Expression:</b> If this option is selected, the Filter mediator matches the evaluation result of a source XPath/JSONPath expression as a string against the given regular expression.</li> </ul>

<b>Source</b>	The expression to locate the value that matches the regular expression that you can define in the <b>Regex</b> parameter.
<b>Regex</b>	The regular expression to match the source value.

**Tip**

You can click **NameSpaces** to add namespaces if you are providing an expression. Then the **Namespace Editor** panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.

**Note**

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.


**Example****Example 1: Sending only messages matching the filter criteria**

In this example, the Filter will get the **To** header value and match it against the given regular expression. If this evaluation returns **true**, it will send the message. If the evaluation returns **false**, it will drop the message.

```
<filter source="get-property('To')" regex=".*/StockQuote.*">
 <then>
 <send/>
 </then>
 <else>
 <drop/>
 </else>
</filter>
```

**Example 2: Applying separate sequences**

In this example, the **Log mediator** is used to log information from a service named Bus Services via a property when the request matches the filter criteria. When the request does not match the filter criteria, another log mediator configuration is used log information from a service named Train Service in a similar way.

```

<filter source="get-property('Action')" regex=". *getBusNo">
 <then>
 <log level="custom">
 <property name="service" value="Bus Services is called"/>
 </log>
 </then>
 <else>
 <log level="custom">
 <property name="service" value="Train Service is called"/>
 </log>
 </else>
</filter>

```

## ForEach Mediator

The ForEach mediator requires an XPath expression and a sequence (inline or referred). It splits the message into a number of different messages derived from the original message by finding matching elements for the XPath expression specified. Based on the matching elements, new messages are created for each iteration and processed sequentially. The processing is carried out based on a specified sequence. The behaviour of ForEach mediator is similar to a generic loop. After mediation, the sub-messages are merged back to their original parent element in the original message sequentially.

The ForEach mediator creates the following properties during mediation.

Property	Description
FOREACH_ORIGINAL_MESSAGE	This contains the original envelop of the messages split by the ForEach mediator.
FOREACH_COUNTER	This contains the count of the messages processed. The message count increases during each iteration.

[Iterate Mediator](#) is quite similar to the ForEach mediator. You can use complex XPath expressions to conditionally select elements to iterate over in both mediators. Following are the main difference between ForEach and Iterate mediators:

- ForEach supports modifying the original payload. You can use Iterate for situations where you send the split messages to a target and collect them by an Aggregate in a different flow
- You need to always accompany an Iterate with an Aggregate mediator. ForEach loops over the sub-messages and merges them back to the same parent element of the message.
- In Iterate you need to send the split messages to an endpoint to continue the message flow. However, ForEach does not allow using [Call](#), [Send](#) and [Callout](#) mediators in the sequence.
- ForEach does not split the message flow, unlike Iterate Mediator. It guarantees to execute in the same thread until all iterations are complete.

When you use ForEach mediator, you can only loop through segments of the message and do changes to a particular segment. For example, you can change the payload using payload factory mediator. But you cannot send the split message out to a service. Once you exit from the for-each loop, it automatically aggregates the split segments. This replaces the for-each function of the complex XSLT mediators using a ForEach mediator and a Payload Factory mediator. However, to implement the split-aggregate pattern, you still need to use Iterate mediator.

# Syntax

```
<for-each expression="xpath" [sequence="sequence_ref"] [id="foreach_id"] >
 <sequence>
 (mediator) +
 </sequence>?
</for-each>
```

## UI Configuration

Mediator switch to source view

### ForEach Mediator

Help

ForEach ID	<input type="text"/>
Expression*	<input type="text"/> Namespaces

**Sequence**

Anonymous

Pick From Registry

**Update**

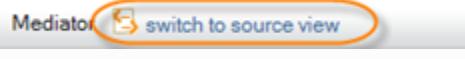
The parameters available to configure the ForEach mediator are as follows.

Parameter Name	Description
<b>ForEach ID</b>	If a value is entered for this parameter, it will be used as the prefix for the FOREACH_ORIGINAL_MESSAGE and FOREACH_COUNTER properties created during mediation. This is an optional parameter. However, it is recommended to define a ForEach ID in nested ForEach scenarios to avoid the properties mentioned from being overwritten.
<b>Expression</b>	<p>The XPath expression with which different messages are derived by splitting the parent message. This expression should have matching elements based on which the splitting is carried out.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <p>You can click <b>NameSpaces</b> to add namespaces when you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p> </div>

<b>Sequence</b>	<p>The <b>mediation sequence</b> that should be applied to the messages derived from the parent message. <b>ForEach</b> mediator is used only for transformations, thereby, you should not include <b>Call</b>, <b>Send</b> and <b>Callout</b> mediators, which are used to invoke endpoints, in this sequence.</p> <p>You can select one of the following options.</p> <ul style="list-style-type: none"> <li>• <b>Anonymous</b>: This allows you to define an anonymous sequence to be applied to the split messages by adding the required mediators as children of the <b>ForEach</b> mediator in the mediator tree.</li> <li>• <b>Pick from Registry</b>: This allows you to pick an existing mediation sequence that is saved in the <b>Registry</b>. Click either <b>Configuration Registry</b> or <b>Governance Registry</b> as relevant to select the required mediation sequence from the Resource Tree.</li> </ul>
-----------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.



## Example

In this configuration, the `//m0:getQuote/m0:request` XPath expression evaluates the split messages to be derived from the parent message. Then the split messages pass through a sequence which includes a **Log** mediator with the log level set to `full`.

```
<foreach id="foreach_1" expression="//m0:getQuote/m0:request"
xmlns:m0="http://services.samples">
 <sequence>
 <log level="full">
 </sequence>
</foreach>
```

## Sample

See [Sample 18: Transforming a Message Using ForEach Mediator](#).

## Header Mediator

The **Header Mediator** allows you to manipulate SOAP and HTTP headers.

The Header mediator is a **conditionally content aware** mediator.

---

[Syntax](#) | [UI Configuration](#) | [Examples](#)

---

## Syntax

```
<header name="string" (value="string|{property}" | expression="xpath")
[scope=default|transport] [action=set|remove]/>
```

The optional `action` attribute specifies whether the mediator should set or remove the header. If no value is specified, the header is set by default.

## UI Configuration

The screenshot shows the WSO2 ESB UI for configuring a Header Mediator. The top navigation bar includes 'Mediator' and 'switch to source view'. The main panel is titled 'Header Mediator' and contains the following fields:

- Name \***: CustomHeader
- Action :** Set (radio button selected)
- Value** (radio button selected) or **Expression \***: CustomHeaderValue
- Inline XML Header \***: (checkbox)
- Scope**: Transport

A 'Namespaces' link is located next to the 'Namespaces' field. A 'Help' link is in the top right corner. At the bottom left is a 'Update' button.

The parameters available to configure the Header mediator are as follows.

Parameter Name	Description
<b>Name</b>	The name of the header element. You can specify the namespace used in the header element by clicking the <b>Namespaces</b> link next to the text field.
<b>Action</b>	Select <b>Set</b> if you want to set the header as a new header. Select <b>Remove</b> if you want to remove the header from the incoming message.
<b>Value/Expression</b>	A static value or an XPath expression that will be executed on the message to set the header value.
<b>Inline XML Header</b>	This parameter allows you to directly input any XML syntax related to the Header mediator (specifically for SOAP headers). For example, to achieve the following configuration, you should enter the <code>lastTradeTimestamp</code> element in the <b>Inline XML Header</b> parameter.
	<pre>&lt;header&gt;     &lt;urn:lastTradeTimestamp xmlns:urn=" http://synapse.apache.org/     "&gt;Mon May 13 13:52:17 IST 2013&lt;/urn:lastTradeTimestamp&gt; &lt;/header&gt;</pre>
<b>Scope</b>	Select <b>Synapse</b> if you want to manipulate SOAP headers. Select <b>Transport</b> if you want to manipulate HTTP headers.
<b>Namespaces</b>	You can click this link to add <b>namespaces</b> if you are providing an expression. The <b>Namespace Editor</b> panel would appear. You can enter any number of namespace prefixes and URL that you have used in the XPath expression in this panel.

## Note

You can also configure the Mediator using XML. Click **switch to source view** in the **Mediator** window.

Mediator

## Examples

This section covers the following scenarios in which the Header mediator can be used.

- Example 1 - SOAP headers
- Example 2 - HTTP headers
- Example 3 - Handling headers with complex XML
- Example 4 - Adding a dynamic SOAP header
- Example 5 - Setting the endpoint URL dynamically

### **Example 1 - SOAP headers**

In the following example, the value for `P1` code should be included in the SOAP header of the message sent from the client to the ESB. To do this, the header mediator is added to the in sequence of the proxy configuration as shown below.

To get a response with `Hello World` in the SOAP header, the header mediator is also added to the out sequence.

```
<inSequence>
 <header>
 <p1:Code xmlns:p1="http://www.XYZ.com/XSD">XYZ</p1:Code>
 </header>
 <send>
 <endpoint>
 <address
uri="http://localhost:8899/services/SimpleStockQuoteService?wsdl" />
 </endpoint>
 </send>
</inSequence>
<outSequence>
 <header>
 <p2:Header xmlns:p2="http://www.ABC.com/XSD">
 <p2:Hello>World</p2:Hello>
 </p2:Header>
 </header>
 <send/>
</outSequence>
```

### **Example 2 - HTTP headers**

The following example makes the ESB add the HTTP header `Accept` with the value `image/jpeg` to the HTTP request made to the endpoint.

```

<in>
 <header name="Accept" value="image/jpeg" scope="transport"/>
 <send>
 <endpoint name="people">
 <address uri="http://localhost:9763/people/eric+cooke" format="get"/>
 </endpoint>
 </send>
</in>
<out>
 <send/>
</out>

```

### **Example 3 - Handling headers with complex XML**

A header can contain XML structured values by embedding XML content within the `<header>` element as shown below.

```

<header>
 <m:complexHeader xmlns:m="http://org.synapse.example">
 <property key="k1" value="v1" />
 <property key="k2" value="v2" />
 </m:complexHeader>
</header>

```

### **Example 4 - Adding a dynamic SOAP header**

The following configuration takes the value of an element named `symbol` in the message body (the namespace `http://services.samples/xsd`), and adds it as a SOAP header named `header1`.

```

<header xmlns:m="http://org.synapse.example" xmlns:sym="http://services.samples/xsd"
name="m:header1" scope="default" expression="//sym:symbol"/>

```

### **Example 5 - Setting the endpoint URL dynamically**

In this example, the Header mediator allows the endpoint URL to which the message is sent to be set dynamically. It specifies the default address to which the message is sent dynamically by deriving the To header of the message via an XPath expression. Then the [Send mediator](#) sends the message to a [Default Endpoint](#). A [Default Endpoint](#) sends the message to the default address of the message (i.e. address specified in the To header). Therefore, in this scenario, selecting the [Default Endpoint](#) results in the message being sent to relevant URL calculated via the `fn:concat('http://localhost:9764/services/Axis2SampleService_', get-property('epr'))` expression.

```

<header name="To"
expression="fn:concat('http://localhost:9764/services/Axis2SampleService_',get-property
y('epr'))"/>
<send>
<endpoint>
<default/>
</endpoint>
</send>

```

## In and Out Mediators

The **In** and **Out** **Mediators** act as predefined filters. Messages that are in the In path of the ESB will traverse through the child mediators of the In Mediator. Messages that are in the Out path of ESB will traverse through the child mediators of the Out Mediator.

### Note

Do not use these mediators in proxy service sequences, as proxy services have the predefined sequences <inSequence> and <outSequence> for this purpose.

---

## Syntax | UI Configuration | Example

### Syntax

#### In

```

<in>
 mediator+
</in>

```

#### Out

```

<out>
 mediator+
</out>

```

---

### UI Configuration

After adding an In or Out mediator, you add and configure its child mediators.

---

### Example

In the following example, the In mediator has a [Log mediator](#) and a [Filter mediator](#) as child mediators. The [Log mediator](#) logs the messages in the In path. Then the [Filter mediator](#) filters these messages and sends the messages which match the filter criteria to <http://localhost:9000>. The messages are sent via the [Send mediator](#) which is added as a child to the [Filter mediator](#).

The messages in the Out path are sent via the [Send mediator](#) which is added as a child to the Out mediator.

```

<sequence name="main" xmlns="http://ws.apache.org/ns/synapse">
 <in>
 <log level="full"/>
 <filter source="get-property('To')" regex="http://localhost:9000.*">
 <send/>
 </filter>
 </in>
 <out>
 <send/>
 </out>
</sequence>

```

## Iterate Mediator

The **Iterate Mediator** implements the [Splitter enterprise integration pattern](#) and splits the message into a number of different messages derived from the parent message. The Iterate mediator is similar to the [Clone mediator](#). The difference between the two mediators is, the Iterate mediator splits a message into different parts, whereas the Clone mediator makes multiple identical copies of the message.

The Iterate mediator is a [content aware mediator](#).

Iterate Mediator is quite similar to the [ForEach mediator](#). You can use complex XPath expressions to conditionally select elements to iterate over in both mediators. Following are the main difference between ForEach and Iterate mediators:

- ForEach supports modifying the original payload. You can use Iterate for situations where you send the split messages to a target and collect them by an Aggregate in a different flow
- You need to always accompany an Iterate with an Aggregate mediator. ForEach loops over the sub-messages and merges them back to the same parent element of the message.
- In Iterate you need to send the split messages to an endpoint to continue the message flow. However, ForEach does not allow using [Call](#), [Send](#) and [Callout](#) mediators in the sequence.
- ForEach does not split the message flow, unlike Iterate Mediator. It guarantees to execute in the same thread until all iterations are complete.

When you use ForEach mediator, you can only loop through segments of the message and do changes to a particular segment. For example, you can change the payload using payload factory mediator. But you cannot send the split message out to a service. Once you exit from the for-each loop, it automatically aggregates the split segments. This replaces the for-each function of the complex XSLT mediators using a ForEach mediator and a Payload Factory mediator. However, to implement the split-aggregate pattern, you still need to use Iterate mediator.

[Syntax](#) | [UI Configuration](#) | [Examples](#)

---

## Syntax

```

<iterate [continueParent=(true | false)] [preservePayload=(true | false)]
[attachPath="xpath"]? expression="xpath">
 <target [to="uri"] [soapAction="qname"] [sequence="sequence_ref"]
[endpoint="endpoint_ref"]>
 <sequence>
 (mediator) +
 </sequence>?
 <endpoint>
 endpoint
 </endpoint>?
 </target>+
</iterate>

```

## UI Configuration

Mediator [switch to source view](#)

### Iterate Mediator

[Help](#)

Iterate ID	<input type="text"/>
Sequential Mediation	<input type="checkbox"/> False
Continue Parent	<input type="checkbox"/> False
Preserve Payload	<input type="checkbox"/> False
Iterate Expression*	<input type="text"/> <a href="#">Namespaces</a>
Attach Path	<input type="text"/> <a href="#">Namespaces</a>

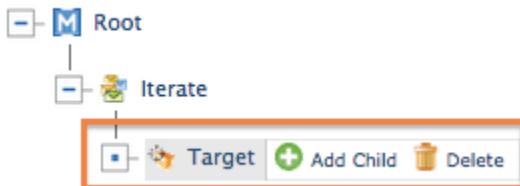
Clicking on Update will add an iteration target if a target is not already present

The parameters available to configure the Iterate mediator are as follows.

Parameter Name	Description
<b>Iterate ID</b>	The iterate ID can be used to identify messages created by the iterate mediator. This is particularly useful when aggregating responses of messages that are created using nested iterate mediators.
<b>Sequential Mediation</b>	This parameter is used to specify whether the split messages should be processed sequentially or not. The processing is carried based on the information relating to the sequence and endpoint specified in the <a href="#">target configuration</a> . The possible values are as follows. <ul style="list-style-type: none"> <li><b>True:</b> If this is selected, the split messages will be processed sequentially. Note that selecting <b>True</b> might cause delays due to high resource consumption.</li> <li><b>False:</b> If this is selected, the split messages will not be processed sequentially. This is the default value and it results in better performance.</li> </ul> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <p>The responses will not necessarily be aggregated in the same order that the requests were sent, even if the <b>sequential Mediation</b> parameter is set to true.</p> </div>

<b>Continue Parent</b>	This parameter is used to specify whether the original message should be preserved or not. Possible values are as follows. <ul style="list-style-type: none"> <li>• <b>True:</b> If this is selected, the original message will be preserved.</li> <li>• <b>False:</b> If this is selected, the original message will be discarded. This is the default value.</li> </ul>
<b>Preserve Payload</b>	This parameter is used to specify whether the original message payload should be used as a template when creating split messages. Possible values are as follows. <ul style="list-style-type: none"> <li>• <b>True:</b> If this is selected, the original message payload will be used as a template.</li> <li>• <b>False:</b> If this is selected, the original message payload will not be used as a template. This is the default value.</li> </ul>
<b>Iterate Expression</b>	The XPath expression used to split the message.. This expression selects the set of XML elements from the request payload that are applied to the mediation defined within the iterate target. Each iteration of the iterate mediator will get one element from that set. New messages are created for each and every matching element and processed in parallel or in sequence based on the value specified for the <b>Sequential Mediation</b> parameter. <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;">         You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.       </div>
<b>Attach Path</b>	To form new messages, you can specify an XPath expression for elements that the split elements are attached to (as expressed in Iterate expression). <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;">         You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.       </div>

Each Iterate mediator has its own target by default. It appears in the mediation tree as shown below once you configure the above parameters and save them.



The following section would appear below the mediation tree.

The screenshot shows the 'Target Configuration' panel within the WSO2 ESB Mediator configuration interface. It includes fields for 'SOAP Action' and 'To Address', and sections for 'Sequence' and 'Endpoint' selection. The 'Sequence' section has three options: 'None' (selected), 'Anonymous', and 'Pick From Registry'. The 'Endpoint' section also has three options: 'None' (selected), 'Anonymous', and 'Pick From Registry'. A 'Help' button is at the top right, and a 'Update' button is at the bottom left.

The parameters available to configure the target configuration are as follows.

Parameter Name	Description
<b>SOAP Action</b>	The SOAP action of the message.
<b>To Address</b>	The target endpoint address.
<b>Sequence</b>	This parameter is used to specify whether split messages should be mediated via a <a href="#">sequence</a> or not, and to specify the sequence if they are to be further mediated. Possible options are as follows. <ul style="list-style-type: none"> <li><b>None:</b> If this is selected, no further mediation will be performed for the split messages.</li> <li><b>Anonymous:</b> If this is selected, you can define an anonymous <a href="#">sequence</a> for the split messages by adding the required mediators as children to <b>Target</b> in the mediator tree.</li> <li><b>Pick From Registry:</b> If this is selected, you can refer to a pre-defined <a href="#">sequence</a> that is currently saved as a resource in the <a href="#">registry</a>. Click either <b>Configuration Registry</b> or <b>Governance Registry</b> as relevant to select the required <a href="#">sequence</a> from the resource tree.</li> </ul>
<b>Endpoint</b>	The <a href="#">endpoint</a> to which the split messages should be sent. Possible options are as follows. <ul style="list-style-type: none"> <li><b>None:</b> If this is selected, the split messages are not sent to an <a href="#">endpoint</a>.</li> <li><b>Anonymous:</b> If this is selected, you can define an anonymous <a href="#">endpoint</a> within the iterate target configuration to which the split messages should be sent. Click the <b>Add</b> link which appears after selecting this option to add the anonymous <a href="#">endpoint</a>. See <a href="#">Adding an Endpoint</a> for further information.</li> <li><b>Pick from Registry:</b> If this is selected, you can refer to a pre-defined <a href="#">endpoint</a> that is currently saves as a resource in the registry. Click either <b>Configuration Registry</b> or <b>Governance Registry</b> as relevant to select the required <a href="#">endpoint</a> from the resource tree.</li> </ul>

## Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.



## Examples

In this example, the Iterate Mediator splits the messages into parts and processes them asynchronously. See also [Splitting Messages into Parts and Processing in Parallel \(Iterate/Aggregate\)](#).

```
<iterate expression="//m0:getQuote/m0:request" preservePayload="true"
 attachPath="//m0:getQuote"
 xmlns:m0="http://services.samples">
 <target>
 <sequence>
 <send>
 <endpoint>
 <address
 uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </sequence>
 </target>
</iterate>
```

## Sample

See [Sample 400: Message Splitting and Aggregating the Responses](#) for another example.

### Log Mediator

The **Log mediator** is used to log mediated messages. For more information on logging, see [Monitoring Logs in WSO2 Administration Guide](#).

The Log mediator is a conditionally content aware mediator.

---

[Syntax](#) | [UI Configuration](#) | [Examples](#)

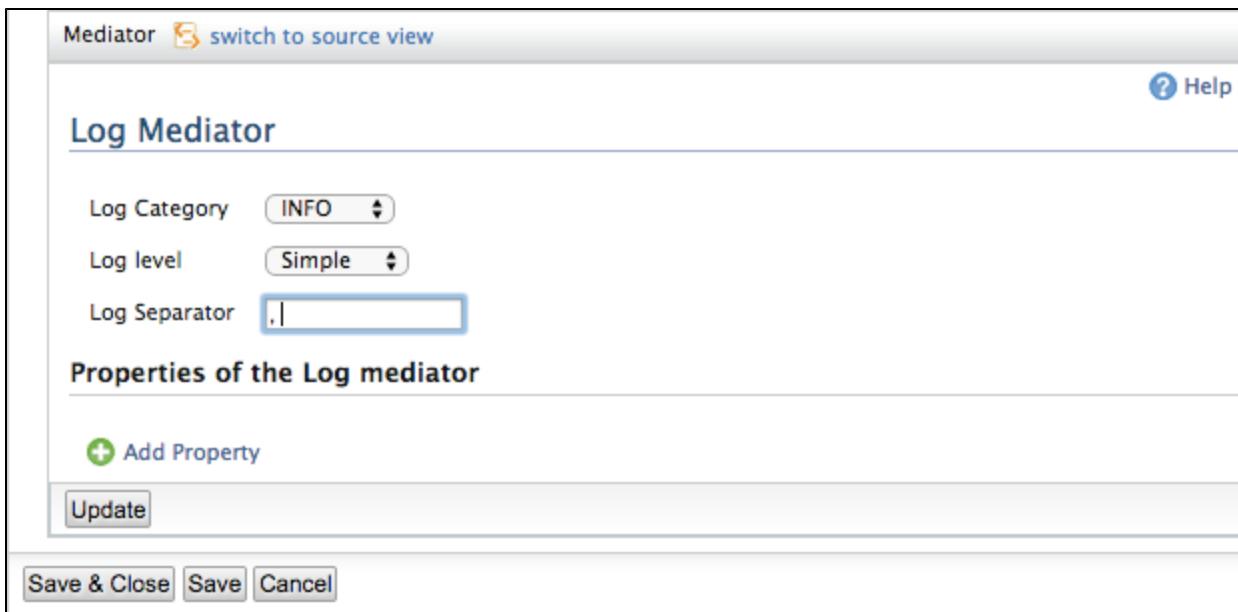
## Syntax

The log token refers to a `<log>` element, which may be used to log messages being mediated.

```
<log [level="string"] [separator="string"]>
 <property name="string" (value="literal" | expression="[XPath|json-eval(JSON
Path)]")/*>
</log>
```

---

[UI Configuration](#)



The general parameters available to configure the Log mediator are as follows.

Parameter Name	Description
<b>Log Category</b>	<p>This parameter is used to specify the log category. Possible values are as follows. Following log levels correspond to ESB service level logs.</p> <ul style="list-style-type: none"> <li>• <b>TRACE</b> - This designates fine-grained informational events than the DEBUG.</li> <li>• <b>DEBUG</b> - This designates fine-grained informational events that are most useful to debug an application.</li> <li>• <b>INFO</b> - This designates informational messages that highlight the progress of the application at coarse-grained level.</li> <li>• <b>WARN</b> - This designates potentially harmful situations.</li> <li>• <b>ERROR</b> - This designates error events that might still allow the application to continue running.</li> <li>• <b>FATAL</b> - This designates very severe error events that will presumably lead the application to abort.</li> </ul>
<b>Log Level</b>	<p>This parameter is used to specify the log level. The possible values are as follows.</p> <ul style="list-style-type: none"> <li>• <b>Full</b>: If this is selected, all the standard headers logged at the <b>Simple</b> level as well as the full payload of the message will be logged. This log level causes the message content to be parsed and hence incurs a performance overhead.</li> <li>• <b>Simple</b>: If this is selected, the standard headers (i.e. To, From, WSAction, SOAPAction, ReplyTo, and MessageID) will be logged.</li> <li>• <b>Headers</b>: If this is selected, all the SOAP header blocks are logged.</li> <li>• <b>Custom</b>: If this is selected, only the properties added to the Log mediator configuration will be logged.</li> </ul> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <p>The <b>properties</b> included in the Log mediator configuration will be logged regardless of the log level selected.</p> </div>
<b>Log Separator</b>	This parameter is used to specify a value to be used in the log to separate attributes. The , comma is default.

Properties to be logged by the Log mediator can be added by clicking **Add Property**. The parameters available to configure a property are as follows.

## Properties of the Log mediator

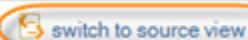
Property Name	Property Value	Value/Expression	Action
<input type="text"/>	Value 	<input type="text"/>	 Delete

 [Add Property](#)

Parameter Name	Description
<b>Property Name</b>	The name of the property to be logged.
<b>Property Value</b>	<p>The possible values for this parameter are as follows:</p> <ul style="list-style-type: none"> <li><b>Value:</b> If this is selected, a static value would be considered as the property value and this value should be entered in the <b>Value/Expression</b> parameter.</li> <li><b>Expression:</b> If this is selected, the property value will be determined during mediation by evaluating an expression. This expression should be entered in the <b>Value/ Expression</b> parameter.</li> </ul>
<b>Value/Expression</b>	<p>This parameter is used to enter a status value as the property value, or to enter an expression to evaluate the property value based on what you entered for the <b>Property Value</b> parameter. When specifying a JSONPath, use the format <code>json-eval(&lt;JSON_PATH&gt;)</code>, such as <code>json-eval(getQuote.request.symbol)</code>. For more information on using JSON with the ESB, see <a href="#">JSON Support</a>.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p> </div>
<b>Action</b>	This parameter allows the property to be deleted.

## Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.

Mediator 

## Examples

### Example 1 - Full log

In this example, everything is logged including the complete SOAP message.

```
<log level="full" xmlns="http://ws.apache.org/ns/synapse"/>
```

### Example 2 - Custom logs

In this example, the log level is `custom`. A property with an XPath expression which is used to get a stock price from a message is included. This results in logging the stock price which is a dynamic value.

```
<log level="custom" xmlns="http://ws.apache.org/ns/synapse">
<property name="text"
 expression="fn:concat('Stock price - ',get-property('stock_price'))"/>
</log>
```

## Loopback Mediator

The **Loopback Mediator** moves messages from the in flow (request path) to the out flow (response path). All the configuration included in the in sequence that appears after the Loopback mediator is skipped.

The messages that have already been passed from the In sequence to the Out sequence cannot be moved to the Out sequence again via the Loopback mediator.

The Loopback mediator is a [content-unaware](#) mediator.

---

## Syntax | UI Configuration | Example

---

### Syntax

The loopback token refers to a `<loopback>` element, which is used to skip the rest of the in flow and move the message to the out flow.

```
<loopback/>
```

### UI Configuration

As with other mediators, after adding the Loopback mediator to a sequence, you can click its up and down arrows to move its location in the sequence.

### Example

This example is a main sequence configuration with two [PayloadFactory](#) mediators. Assume you only want to use the first factory but need to keep the second factory in the configuration for future reference. The Loopback mediator is added after the first [PayloadFactory](#) mediator configuration to skip the second [PayloadFactory](#) mediator configuration. This configuration will cause the message to be processed with the first payload factory and then immediately move to the out flow, skipping the second payload factory in the in flow.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <payloadFactory>
 <format>
 <m:messageBeforeLoopBack xmlns:m="http://services.samples">
 <m:messageBeforeLoopBackSymbol>
 <m:symbolBeforeLoopBack>$1</m:symbolBeforeLoopBack>
 </m:messageBeforeLoopBackSymbol>
 </m:messageBeforeLoopBack>
 </format>
 <args>
 <arg xmlns:m0="http://services.samples"
 evaluator="xml"
 expression="//m0:symbol/text()"/>
 </args>
 </payloadFactory>

 <loopback/>

 <payloadFactory>
 <format>
 <m:messageAfterLoopBack xmlns:m="http://services.samples">
 <m:messageAfterLoopBackSymbol>
 <m:symbolAfterLoopBack>$1</m:symbolAfterLoopBack>
 </m:messageAfterLoopBackSymbol>
 </m:messageAfterLoopBack>
 </format>
 <args>
 <arg xmlns:m0="http://services.samples"
 evaluator="xml"
 expression="//m0:symbolBeforeLoopBack/text()"/>
 </args>
 </payloadFactory>
 </in>
 <out>
 <send/>
 </out>
 </sequence>
</definitions>

```

## OAuth Mediator

The **OAuth Mediator** supports 2 forms of OAuth. It bypasses the RESTful requests and authenticates users against WSO2 Identity Server.

When a client tries to invoke a RESTful service, it may be required to verify the credentials of the client. This can be achieved by registering an OAuth application in the WSO2 Identity Server. When the client sends a REST call with the Authorization header to the ESB, the OAuth mediator validates it with the Identity server and proceeds.

See [2-legged OAuth for Securing a RESTful Service](#) for detailed instructions to carry out this process.

If you are using OAuth 1 a, you will get the `org.apache.synapse.SynapseException: Unable to find SCOPE value in Synapse Message Context` error when the `SCOPE` property is not set in the synapse message context. To avoid this error, add a property with the name `scope` and a value in the

synapse message context as shown in the [Example](#) section.

## Syntax | UI Configuration | Example

### Syntax

```
<oauthService remoteServiceUrl="" username="" password="" />
```

### UI Configuration

The screenshot shows the 'OAuth Mediator' configuration screen. It has three input fields: 'OAuth Server', 'Username', and 'Password', each with a corresponding text input box. Below these fields is a 'Update' button. At the bottom of the screen are three buttons: 'Save & Close', 'Save', and 'Cancel'.

The parameters available to configure the OAuth mediator are as follows.

Parameter Name	Description
<b>OAuth Server</b>	The server URL of the WSO2 Identity Server.
<b>Username</b>	The user name to be used to log into the WSO2 Identity Server.
<b>Password</b>	The password used to log into the WSO2 Identity Server.

### Example

In the following OAuth mediator configuration accesses a remote service via the `https://localhost:9443/service` URL. The user accessing this service is authenticated via the OAuth application registered in the WSO2 Identity Server and accessed via the `http://ws.apache.org/ns/synapse` URL. The username used to log into the WSO2 Identity Server is `foo` and the password is `bar`. Both the user name and the password should be registered in the Identity Server. The [Property mediator](#) adds a property named `scope` to the synapse message context. The value of this property will be used by the OAuth mediator to send the OAuth request.

The following example is applicable for OAuth 2.0 as well.

```
<property name="scope" scope="default" type="STRING" value="123"/>
<oAuthService xmlns="http://ws.apache.org/ns/synapse"
remoteServiceUrl="https://localhost:9443/services" username="foo" password="bar" />
```

## PayloadFactory Mediator

The **PayloadFactory Mediator** transforms or replaces the contents of a message. Each argument in the mediator configuration can be a static value, or you can specify an XPath or JSON expression to get the value at runtime by evaluating the provided expression against the existing SOAP message. You can configure the format of the request or response and map it to the arguments provided.

The PayloadFactory mediator is a [content aware mediator](#).

---

### Syntax | UI Configuration | Examples

#### Syntax

```
<payloadFactory media-type="xml | json">
 <format .../>
 <args>
 <arg (value="string" | expression=" {xpath} | {json} ")/*>
 </args>
</payloadFactory>
```

You can also provide the expression inline to get the value of the Synapse message context property, instead of specifying the expression as arguments.

The `media-type` attribute specifies whether to format the message in XML or JSON. If no media type is specified, the message is formatted in XML. If you want to change the payload type of the outgoing message, such as to change it to JSON, add the `messageType` property after the `</payloadFactory>` tag. For example:

```
...
</payloadFactory>
<property name="messageType" value="application/json" scope="axis2" />
```

---

#### UI Configuration

The screenshot shows the configuration interface for the PayloadFactory Mediator. At the top, it says "Payload Media-Type" set to "xml". Below that, "Payload Format" is set to "Define inline". There are two options: "Define inline" (selected) and "Pick From Registry". Under "Arguments", there is a "Add Argument" button. At the bottom, there are "Save & Close", "Save", and "Cancel" buttons.

Parameters available to configure the PayloadFactory mediator are as follows:

Parameter Name	Description
<b>Payload Media-Type</b>	This parameter is used to specify whether the message payload should be created in JSON or XML.
<b>Payload Format</b>	<p><b>Define Inline:</b> If this is selected, the payload format can be defined within the PayloadFactory mediator configuration by entering it in the text field which appears. To add content to the payload, enter variables for each value you want to add using the format \$n (starting with 1 and incrementing with each additional variable, i.e., \$1, \$2, etc.). You will then create arguments in the same order as the variables to specify each variable's actual value.</p> <p><b>Pick from Registry:</b> If this is selected, an existing payload format which is saved in the <a href="#">Registry</a> can be selected. Click either <b>Governance Registry</b> or <b>Configuration Registry</b> as relevant to select the payload format from the resource tree.</p>
<b>Arguments</b>	<p>This section is used to add an argument that defines the actual value of each variable in the format definition. The arguments must be entered in the same order as the variables in the format, so that the first argument defines the value for variable \$1, the second argument defines the value for variable \$2, etc. An argument can specify a literal string (e.g., "John") or an XPath or JSON expression that extracts the value from the content in the incoming payload.</p> <div style="border: 1px solid orange; padding: 10px;"> <p>If you already know the argument is XML, to avoid the PayloadFactory mediator throwing an error when the argument value begins with an html tag, add the following attribute:</p> <pre>deepCheck="false"</pre> <p>e.g., <code>&lt;arg deepCheck="false" evaluator="xml" expression="ctx:variable1"&gt;</code></p> </div>

## Note

You can configure the mediator using XML. Click **switch to source** view in the **Mediator** window.

Mediator  switch to source view

## Examples

- Example 1: XML
- Example 2: JSON
- Example 3: Adding arguments
- Example 4: Suppressing the namespace
- Example 5: Including a complete SOAP envelope as the format
- Example 6: Adding expression inline xml
- Example 6: Adding expression inline json
- Samples

This section provides examples of using PayloadFactory mediator to generate XML and JSON messages.

### ***Example 1: XML***

This example is based on [Sample 17: Transforming / Replacing Message Content with PayloadFactory Mediator](#).

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <!-- using payloadFactory mediator to transform the request message -->
 <payloadFactory media-type="xml">
 <format>
 <m:getQuote xmlns:m="http://services.samples">
 <m:request>
 <m:symbol>$1</m:symbol>
 </m:request>
 </m:getQuote>
 </format>
 <args>
 <arg xmlns:m0="http://services.samples" expression="//m0:Code"/>
 </args>
 </payloadFactory>
 </in>
 <out>
 <!-- using payloadFactory mediator to transform the response message -->
 <payloadFactory media-type="xml">
 <format>
 <m:CheckPriceResponse xmlns:m="http://services.samples/xsd">
 <m:Code>$1</m:Code>
 <m:Price>$2</m:Price>
 </m:CheckPriceResponse>
 </format>
 <args>
 <arg xmlns:m0="http://services.samples/xsd"
expression="//m0:symbol"/>
 <arg xmlns:m0="http://services.samples/xsd"
expression="//m0:last"/>
 </args>
 </payloadFactory>
 </out>
 <send/>
 </sequence>
</definitions>

```

### **Example 2: JSON**

This example sends a JSON message to the back end. For more information on using JSON with the ESB, see [JSON Support](#).

```

<payloadFactory media-type="json">
 <format>
 {
 "coordinates": null,
 "created_at": "Fri Jun 24 17:43:26 +0000 2011",
 "truncated": false,
 "favorited": false,
 "id_str": "$1",
 "entities": {
 "urls": [
],
 "hashtags": [
 {
 "text": "$2",
 "indices": [
 35,
 45
]
 }
],
 "user_mentions": [
]
 },
 "in_reply_to_user_id_str": null,
 "contributors": null,
 "text": "$3",
 "retweet_count": 0,
 "id": "#",
 "in_reply_to_status_id_str": null,
 "geo": null,
 "retweeted": false,
 "in_reply_to_user_id": null,
 "source": "YoruFukurou",
 "in_reply_to_screen_name": null,
 "user": {
 "id_str": "#",
 "id": "#"
 },
 "place": null,
 "in_reply_to_status_id": null
 }
 </format>
 <args>
 <arg expression=".entities.hashtags[0].text" evaluator="json"/>
 <arg expression="//entities/hashtags/text"/>
 <arg expression="//user/id"/>
 <arg expression="//user/id_str"/>
 <arg expression=".user.id" evaluator="json"/>
 <arg expression=".user.id_str" evaluator="json"/>
 </args>
 </payloadFactory>
 <property name="messageType" value="application/json" scope="axis2"/>

```

#### Note

By default, JSON messages are converted to XML when they are received by the PayloadFactor mediator. However, if you enable the JSON stream formatter and builder, incoming JSON messages are left in JSON format, which improves performance. To enable them, uncomment the following lines in <PRODUCT\_HOME>/repository/conf/axis2/axis2.xml:

```
<!--messageFormatter contentType="application/json"
 class="org.apache.axis2.json.JSONStreamFormatter"-->

<!--messageBuilder contentType="application/json"
 class="org.apache.axis2.json.JSONStreamBuilder"-->
```

When the JSON stream formatter and builder are enabled, if you specify a JSON expression in the PayloadFactory mediator, you must use the `evaluator` attribute to specify that it is JSON. You can also use the evaluator to specify that an XPath expression is XML, or if you omit the evaluator attribute, XML is assumed by default. For example:

XML	<code>&lt;arg xmlns:m0=" http://sample" expression="//m0:symbol" evaluator="xml" /&gt;</code>
	or
	<code>&lt;arg xmlns:m0=" http://sample " expression="//m0:symbol" /&gt;</code>
JSON	<code>&lt;arg expression=".user.id" evaluator="json" /&gt;</code>

#### **Example 3: Adding arguments**

In the following configuration, the values for format parameters `code` and `price` will be assigned with values that are evaluated from arguments given in the specified order.

```
<payloadFactory media-type="xml">
 <format>
 <m:checkpriceresponse xmlns:m="http://services.samples/xsd">
 <m:code>$1</m:code>
 <m:price>$2</m:price>
 </m:checkpriceresponse>
 </format>
 <args>
 <arg xmlns:m0="http://services.samples/xsd" expression="//m0:symbol"/>
 <arg xmlns:m0="http://services.samples/xsd" expression="//m0:last"/>
 </args>
</payloadFactory>
```

#### **Example 4: Suppressing the namespace**

To prevent the ESB from adding the default Synapse namespace in an element in the payload format, use `xmlns=""` as shown in the following example.

```
<ser:getPersonByUmid xmlns:ser="http://service.directory.com">
 <umid xmlns="">sagara</umid>
</ser:getPersonByUmid>
```

#### **Example 5: Including a complete SOAP envelope as the format**

In the following configuration, an entire SOAP envelope is added as the format defined inline. This is useful when you want to generate the result of the PayloadFactory mediator as a complete SOAP message with SOAP headers.

```
<payloadFactory media-type="xml">
<format>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body>
<error>
<mes>$1</mes>
</error>
</soapenv:Body>
</soapenv:Envelope>
</format>
<args>
<arg value=" Your request did not return any results. Please enter a valid EIN and try again"/>
</args>
</payloadFactory>
```

#### **Example 6: Adding expression inline xml**

In the following configuration, the values for format parameters `code` and `price` will be assigned with values from Synapse message context properties `symbol` and `last`.

```
<payloadFactory media-type="xml">
<format>
<m:checkpriceresponse xmlns:m="http://services.samples/xsd">
<m:code>$ctx:symbol</m:code>
<m:price>$ctx:last</m:price>
</m:checkpriceresponse>
</format>
</payloadFactory>
```

#### **Example 6: Adding expression inline json**

In the following configuration, the values for format parameters `code` and `price` will be assigned with values from Synapse message context properties `symbol` and `last`.

```
<payloadFactory media-type="json">
<format>
"tags": {
 "name": "$ctx:symbol",
 "value": "$ctx:last"
}
</format>
</payloadFactory>
```

#### **Samples**

The following samples demonstrate the use of the PayloadFactory mediator.

- [Sample 17: Transforming / Replacing Message Content with PayloadFactory Mediator](#)

#### **POJOCommand Mediator**

**Tip**

This mediator implements the popular [command pattern](#).

The **POJOCommand** (or **Command**) **Mediator** creates an instance of the specified command class, which may implement the `org.apache.synapse.Command` interface or should have a public void method `public void execute()`. If any properties are specified, the corresponding `setter` methods are invoked on the class before each message is executed. It should be noted that a new instance of the **POJOCommand** class is created to process each message. After execution of the **POJOCommand** Mediator, the new value returned by a call to the corresponding `getter` method is stored back in the message or in the context depending on the `action` attribute of the property.

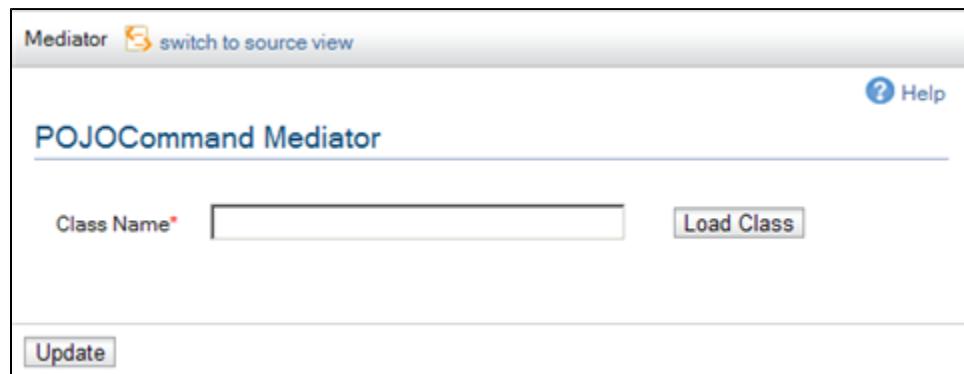
The `action` attribute may specify whether this behavior is expected or not via the `Read`, `Update` and `ReadAndUpdate` properties.

## Syntax | UI Configuration

### Syntax

```
<pojoCommand name="class-name">
(
<property name="string" value="string"/> |
<property name="string" context-name="literal" [action=(ReadContext | UpdateContext
| ReadAndUpdateContext)]>
(either literal or XML child)
</property> |
<property name="string" expression="xpath" [action=(ReadMessage | UpdateMessage |
ReadAndUpdateMessage)]/>
) *
</pojoCommand>
```

### UI Configuration



To load the **POJOCommand** class, enter the class name in the **Class Name** parameter and click **Load Class**. Then the mediator page will expand as shown below.

**POJOCommand Mediator**

Class Name\*  [Load Class](#)

**Properties of the POJOCommand mediator**

Property Name	Read Info	Update Info	Action
Z	From: <input type="button" value="Value"/> Value: <input type="text" value="value3"/>	To: <input type="button" value="- NONE -"/> <a href="#">Delete</a>	
Y	From: <input type="button" value="Value"/> Value: <input type="text" value="value2"/>	To: <input type="button" value="- NONE -"/> <a href="#">Delete</a>	
X	From: <input type="button" value="Value"/> Value: <input type="text" value="value1"/>	To: <input type="button" value="- NONE -"/> <a href="#">Delete</a>	

Parameters available to configure properties for the POJOCommand mediator are as follows.

Parameter	Description
<b>Property Name</b>	The name of the property. This will be automatically loaded from the class.
<b>Read Info</b>	The value to set for the property. You can select one of the following sources in the <b>From</b> field. <ul style="list-style-type: none"> <li><b>Value:</b> Select this if you want the property value to be a static value. This static value should be entered in the <b>Value</b> field.</li> <li><b>Message:</b> Select this if you want to read the property value from an incoming message. The XPath expression to execute on the relevant message should be entered in the <b>Value</b> field.</li> <li><b>Context:</b> Select this if you want to read a value from message context properties. The relevant property key should be entered in the <b>Value</b> field.</li> </ul>
<b>Update Info</b>	This parameter specifies the action to be executed on the property value. You can select one for the following actions in the <b>To</b> field. <ul style="list-style-type: none"> <li><b>None:</b> Select this if no activity should be performed on the property value.</li> <li><b>Message:</b> Select this if you want to update the message. The XPath expression of the element you want to update should be entered in the <b>Value</b> field.</li> <li><b>Context:</b> Select this if you want to update properties (message context). The relevant property key should be entered in the <b>Value</b> field.</li> </ul>
<b>Action</b>	Click <b>Delete</b> to delete a property.

### Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.

Mediator [switch to source view](#)

## Property Mediator

The **Property Mediator** has no direct impact on the message, but rather on the message context flowing through Synapse. You can retrieve the properties set on a message later through the [Synapse XPath Variables](#) or the [get-property\(\)](#) extension function. A property can have a defined scope for which it is valid. If a property has no defined scope, it defaults to the Synapse message context scope. Using the property element with the **action** specified as `remove`, you can remove any existing message context properties.

See [Properties Reference](#) for a list of various types of properties supported by WSO2 ESB with descriptions and use cases.

The Property mediator is a conditionally content aware mediator.

---

[Syntax](#) | [UI configuration](#) | [Examples](#)

---

## Syntax

```
<property name="string" [action=set|remove] [type="string"] (value="literal" |
expression="xpath") [scope=default|transport|axis2|axis2-client] [pattern="regex"
[group="integer"]]>
 <xml-element/>?
</property>
```

## UI configuration

Mediator [switch to source view](#) [Help](#)

**Property Mediator**

Name *	<input type="text"/>
Action	<input checked="" type="radio"/> Set <input type="radio"/> Remove
Set Action as	<input checked="" type="radio"/> Value <input type="radio"/> Expression
Type	<input type="text" value="STRING"/>
Value *	<input type="text"/>
Pattern	<input type="text"/>
Group	<input type="text"/>
Scope	<input type="text" value="Synapse"/>
<input type="button" value="Update"/>	

The parameters available for configuring the Property mediator are as follows:

Parameter Name	Description
Name	A name for the property.

Action	<p>The action to be performed for the property.</p> <ul style="list-style-type: none"> <li><b>Set:</b> If this is selected, the property will be set in the message context.</li> <li><b>Remove:</b> If this is selected, the property will be removed from the message context.</li> </ul>
Set Action As	<p>The possible values for this parameter are as follows:</p> <ul style="list-style-type: none"> <li><b>Value:</b> If this is selected, a static value would be considered as the property value and this value should be entered in the <b>Value</b> parameter.</li> <li><b>Expression:</b> If this is selected, the property value will be determined during mediation by evaluating an expression. This expression should be entered in the <b>Expression</b> parameter.</li> </ul>
Type	<p>The data type for the property. Property mediator will handle the property as a property of selected type. Available values are as follows.</p> <ul style="list-style-type: none"> <li><b>STRING</b></li> <li><b>INTEGER</b></li> <li><b>BOOLEAN</b></li> <li><b>DOUBLE</b></li> <li><b>FLOAT</b></li> <li><b>LONG</b></li> <li><b>SHORT</b></li> <li><b>OM</b></li> </ul> <p><b>String</b> is the default type.</p> <div style="border: 1px solid #ccc; padding: 10px;"> <p>The <b>OM</b> type is used to set xml property values on the message context. This is useful when the expression associated with the property mediator evaluates to an XML node during mediation. When the <b>OM</b> type is used, the XML is converted to an AXIOM OMElement before it is assigned to a property.</p> </div>
Value	If the <b>Value</b> option is selected for the <b>Set Action As</b> parameter, the property value should be entered as a constant in this parameter.
Expression	<p>If the <b>Expression</b> option is selected for the <b>Set Action As</b> parameter, the expression which determines the property value should be entered in this parameter. This expression can be an XPath expression or a JSONPath expression.</p> <p>When specifying a JSONPath, use the format <code>json-eval(&lt;JSON_PATH&gt;)</code>, such as <code>json-eval(getQuote.request.symbol)</code>. In both XPath and JSONPath expressions, you can return the value of another property by calling <code>get-property(property-name)</code>. For example, you might create a property called <code>JSON_PATH</code> of which the value is <code>json-eval(pizza.toppings)</code>, and then you could create another property called <code>JSON_PRINT</code> of which the value is <code>get-property('JSON_PATH')</code>, allowing you to use the value of the <code>JSON_PATH</code> property in the <code>JSON_PRINT</code> property. For more information on using JSON with the ESB, see <a href="#">JSON Support</a>.</p> <div style="border: 1px solid #80C080; padding: 10px;"> <p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p> </div>
Pattern	This parameter is used to enter a regular expression that will be evaluated against the value of the property or result of the XPath/JSON Path expression.

<b>Group</b>	The number (index) of the matching item evaluated using the regular expression entered in the <b>Pattern</b> parameter.
<b>Scope</b>	<p>The scope at which the property will be set or removed from. Possible values are as follows.</p> <ul style="list-style-type: none"> <li>• <b>Synapse</b>: This is the default scope. The properties set in this scope last as long as the transaction (request-response) exists.</li> <li>• <b>Transport</b>: The properties set in this scope will be considered transport headers. For example, if it is required to send an HTTP header named 'CustomHeader' with an outgoing request, you can use the property mediator configuration with this scope.</li> <li>• <b>Axis2</b>: Properties set in this scope have a shorter life span than those set in the <b>Synapse</b> scope. They are mainly used for passing parameters to the underlying Axis2 engine</li> <li>• <b>axis2-client</b>: This is similar to the <b>Synapse</b> scope. The difference between the two scopes is that the <b>axis2-client</b> scope can be accessed inside the <code>mediate()</code> method of a mediator via a custom mediator created using the <a href="#">Class mediator</a>. See <a href="#">axis2-client</a> for further information.</li> <li>• <b>Operation</b>: This scope is used to retrieve a property in the operation context level.</li> <li>• <b>Registry</b>: This scope is used to retrieve properties within the registry.</li> <li>• <b>System</b>: This scope is used to retrieve Java system properties.</li> </ul> <p>See <a href="#">XPath Extension Functions</a> for a detailed explanation of each scope.</p>

## Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.



## Examples

### Example 1: Setting and logging and property

In this example, we are setting the property symbol and later we can log it using the Log Mediator.

```
<property name="symbol"
 expression="fn:concat("Normal Stock - ", //m0:getQuote/m0:request/m0:symbol)"
 xmlns:m0="http://services.samples/xsd"/>

<log level="custom">
 <property name="symbol" expression="get-property('symbol')"/>
</log>
```

### Example 2: Sending a fault message based on the Accept http header

In this configuration, a response is sent to the client based on the Accept header. The [PayloadFactory](#) mediator transforms the message contents. Then a [Property mediator](#) sets the message type based on the Accept header using the `$ctx:accept` expression. The message is then sent back to the client via the [Respond](#) mediator.

## Note

There are predefined XPath variables (such as `$ctx`) that you can directly use in the Synapse configuration, instead of using the `synapse:get-property()` function. These XPath variables get properties of various scopes and have better performance than the `get-property()` function, which can have much lower

performance because it does a registry lookup. These XPath variables get properties of various scopes. For more information on these XPath variables, see [Synapse XPath Variables](#).

```
<payloadFactory media-type="xml">
 <format>
 <m:getQuote xmlns:m="http://services.samples">
 <m:request>
 <m:symbol>Error</m:symbol>
 </m:request>
 </m:getQuote>
 </format>
</payloadFactory>
<property name="messageType" expression="$ctx:accept" scope="axis2" />
<respond/>
```

## Publish Event Mediator

The **Publish Event mediator** constructs events and publishes them to different systems such as BAM and CEP. This is done via [event sinks](#).

The Publish Event mediator is a [content-aware mediator](#).

---

### Syntax | UI Configuration | Example

#### Syntax

```
<publishEvent>
 <eventSink>String</eventSink>
 <streamName>String</streamName>
 <streamVersion>String</streamVersion>
 <attributes>
 <meta>
 <attribute name="string" type="dataType" default="" (value="literal" |
expression="[XPath]") />
 </meta>
 <correlation>
 <attribute name="string" type="dataType" default="" (value="literal" |
expression="[XPath]") />
 </correlation>
 <payload>
 <attribute name="string" type="dataType" default="" (value="literal" |
expression="[XPath]") />
 </payload>
 <arbitrary>
 <attribute name="string" type="dataType" default="" value="literal" />
 </arbitrary>
 </attributes>
</publishEvent>
```

## UI Configuration

[Help](#)

### Publish Event Mediator

Stream Name *	<input type="text"/>			
Stream Version *	<input type="text"/>			
EventSink *	<input type="button" value="▼"/>			

#### Meta Attributes

Attribute Name	Attribute Value	Value/Expression	Type	Action
<input type="text"/>	<input type="button" value="Value ▾"/>	<input type="text"/>	<input type="button" value="STRING ▾"/>	Delete

[Add Attribute](#)

#### Correlated Attributes

Attribute Name	Attribute Value	Value/Expression	Type	Action
<input type="text"/>	<input type="button" value="Value ▾"/>	<input type="text"/>	<input type="button" value="STRING ▾"/>	Delete

[Add Attribute](#)

#### Payload Attributes

Attribute Name	Attribute Value	Value/Expression	Type	Action
<input type="text"/>	<input type="button" value="Value ▾"/>	<input type="text"/>	<input type="button" value="STRING ▾"/>	Delete

[Add Attribute](#)

#### Arbitrary Attributes

Attribute Name	Attribute Value	Value/Expression	Type	Action
<input type="text"/>	<input type="button" value="Value ▾"/>	<input type="text"/>	<input type="button" value="STRING ▾"/>	Delete

[Add Attribute](#)

Parameters that can be configured for the Publish Event mediator are as follows.

Parameter Name	Description
<b>Stream Name</b>	The name of the stream to be used for sending data.
<b>Stream Version</b>	The version of the stream to be used for sending data.
<b>EventSink</b>	The name of the event sink to which the events should be published.

You can add the following four types of attributes to a Publish Event mediator configuration.

**Meta Attributes:** The list of attributes which are included in the Meta section of the event.

**Correlated Attributes:** The list of attributes that are included in the Correlated section of the event.

**Payload Attributes:** The list of attributes that are included in the Payload section of the event.

**Arbitrary Attributes:** The list of attributes that are included in the Arbitrary section of the event.

The parameters that are available to configure an individual attribute are as follows.

Parameter Name	Description
<b>Attribute Name</b>	The name of the attribute.
<b>Attribute Value</b>	<p>This parameter specifies whether the value of the attribute should be a static value or an expression.</p> <ul style="list-style-type: none"> <li>• <b>Value:</b> Select this if the attribute value should be a static value, and enter the relevant value in the <b>Value/Expression</b> parameter.</li> <li>• <b>Expression:</b> Select this if the attribute value should be evaluated via an XPath expression, and enter the relevant expression in the <b>Value/Expression</b> parameter.</li> </ul>
<b>Value/Expression</b>	The value of the attribute. You can enter a static value, or an expression to evaluate the value depending on the selection made in the <b>Attribute Value</b> parameter.
<b>Type</b>	The data type of the attribute.
<b>Action</b>	Click <b>Delete</b> in the relevant row to delete an attribute.

### Example

In this configuration, the Publish Event mediator uses four attributes to extract information from the ESB. This information is published in an event sink in the ESB named `sample_event_sink`.

```

<publishEvent>
 <eventSink>sample_event_sink</eventSink>
 <streamName>stream_88</streamName>
 <streamVersion>1.1.2</streamVersion>
 <attributes>
 <meta>
 <attribute name="http_method"
 type="STRING"
 defaultValue=""
 expression="get-property('axis2', 'HTTP_METHOD')"/>
 <attribute name="destination"
 type="STRING"
 defaultValue=""
 expression="get-property('To')"/>
 </meta>
 <correlation>
 <attribute name="date"
 type="STRING"
 defaultValue=""
 expression="get-property('SYSTEM_DATE')"/>
 </correlation>
 <payload>
 <attribute xmlns:m0="http://services.samples"
 name="symbol"
 type="STRING"
 defaultValue=""
 expression="$body/m0:getQuote/m0:request/m0:symbol"/>
 </payload>
 </attributes>
</publishEvent>

```

## Respond Mediator

The **Respond Mediator** stops the processing on the current message and sends the message back to the client as a response.

---

### Syntax | UI Configuration | Example

---

#### Syntax

The respond token refers to a `<respond>` element, which is used to stop further processing of a message and send the message back to the client.

```
<respond/>
```

#### UI Configuration

As with other mediators, after adding the Respond mediator to a sequence, you can click its up and down arrows to move its location in the sequence.

#### Example

Assume that you have a configuration that sends the request to the Stock Quote service and changes the response

value when the symbol is WSO2 or CRF. Also assume that you want to temporarily change the configuration so that if the symbol is CRF, the ESB just sends the message back to the client without sending it to the Stock Quote service or performing any additional processing. To achieve this, you can add the Respond mediator at the beginning of the CRF case as shown below. All the configuration after the Respond mediator is ignored. As a result, the rest of the CRF case configuration is left intact, allowing you to revert to the original behavior in the future by removing the Respond mediator if required.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <switch source="//m0:getQuote/m0:request/m0:symbol"
xmlns:m0="http://services.samples">
 <case regex="WSO2">
 <property name="symbol" value="Great stock - WSO2"/>
 <send>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </case>
 <case regex="CRF">
 <respond/>
 <property name="symbol" value="Are you sure? - CRF"/>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </case>
 <default>
 <property name="symbol"
expression="fn:concat('Normal Stock - ',
//m0:getQuote/m0:request/m0:symbol)">
 </default>
 </switch>
 </in>
 <out>
 <send/>
 </out>
 </sequence>
 </definitions>

```

## Router Mediator

The Router Mediator is deprecated. You can use the [Filter Mediator](#) or [Conditional Router Mediator](#) instead.

## Rule Mediator

The **Rule Mediator** integrates the WSO2 Rules component to the WSO2 ESB in order to define dynamic integration decisions in terms of rules.

The Rule mediator uses an XML message as an input and produces a processed XML message after applying a set of rules. The result xml message can be used as the new soap body message. Alternatively, the information in the processed XML message can be used route the message or do any further processing. or the information can be

used to route the message. The use the information to route the message or do any other processing.

The Rule mediator is a **content aware** mediator.

---

## Syntax | UI Configuration | Example

---

### Syntax

```
<rule>

 <ruleset>
 <source [key="xs:string"]>
 [in-Lined]
 </source>
 <creation>
 <property name="xs:string" value="xs:string"/>*
 </creation>
 </ruleset>

 <session type="[stateless|stateful]"/*>

 <facts>
 <fact name="xs:string" type="xs:string" expression="xs:string"
value="xs:string"/>+
 </facts>

 <results>
 <result name="xs:string" type="xs:string" expression="xs:string"
value="xs:string"/>*
 </results>

 [<childMediators>
 <mediator/>*
 </childMediators>]

</rule>
```

---

### UI Configuration

Mediator [switch to source view](#)

[Help](#)

## Rule Mediator

### Source

Value \*

Xpath  [NameSpaces](#)

### Target

Value \*

Result Xpath  [NameSpaces](#)

Xpath  [NameSpaces](#)

Action

### Rule Set

Rule Script As\*  In-Lined  Key  URL

[Editor](#)

Rule Type

### Input Facts

Wrapper Name  NameSpace

[Add a Fact](#)

### Output Facts

Wrapper Name  NameSpace

[Add a Fact](#)

The parameters available to configure the Rule mediator are categorised into the following main elements.

Source

This section is used to enter the source from which the XML input for the mediator should be taken.

The parameters available in this section are as follows.

Parameter Name	Description

<b>Value</b>	This parameter can be used to enter a static value as the source.
<b>XPath</b>	This parameter enables you to use an XPath expression to obtain the input facts of the message from a SOAP body, a SOAP header or a property. This XPath expression can be used to specify the message path for the relevant fact type even when creating fact objects.

#### Target

This section is used to enter details of the destination to which the result of the mediation should be added.

The parameters available in this section are as follows.

Parameter Name	Description
<b>Value</b>	This parameter is used to enter a static value to specify the location go which the resulting message should be added.
<b>Result Xpath</b>	This parameter is used to derive the location to which the resulting message should be added via an Xpath expression.
<b>Xpath</b>	This parameter is used to enter a Xpath expression to specify a part of the generated result XML to be added to the target.
<b>Action</b>	This parameter is used to specify whether the result XML should replace the target, or whether it should be added as a child or a sibling.

#### Rule Set

The rule set contains the rules that apply to the input and output facts based on which WSO2 BRS can create the web service WSDL. input facts are the facts that are sent by the rule service client and the output facts are the facts which are received by the client.

The parameters available in this section are as follows.

Parameter Name	Description

<b>Rule Script As</b>	<p>This parameter is used to specify how you want to enter the rule script. Possible options are as follows.</p> <ul style="list-style-type: none"> <li><b>In-Lined:</b> If this is selected, the rule script can be added within the mediator configuration.</li> <li><b>Key:</b> If this is selected, the rule script can be saved in the <a href="#">Registry</a> and accessed via a key.</li> <li><b>URL:</b> If this is selected, you can refer to a rule script via a URL.</li> </ul> <p>If the rule set is non-XML, you may need to wrap it with a CDATA section inside a XML tag as shown in the example below.</p> <pre>&lt;X&gt;&lt;! [ CDATA[ native code ] ]&gt;&lt;/X&gt;</pre> <p><b>Note</b></p> <p>The key or inline Rule script must be defined. Otherwise, the Rule Mediator configuration will be invalid.</p>
<b>Rule Type</b>	This parameter is used to specify whether the rule type is <b>Regular</b> or <b>Decision Table</b> .

#### Input Facts

Input facts are facts that are sent by the rule service client. This section contains parameters that can be used to configure the input facts. Values should be defined for the following parameters before you add the individual input facts.

Parameter Name	Description
<b>Wrapper Name</b>	The name of the wrapping element for all the facts.
<b>NameSpace</b>	The namespace for the wrapping element.

You can click **Add a Fact** to add an input fact. The page will expand to display the following section.

Input Facts						
Wrapper Name	<input type="text"/>	NameSpace	<input type="text"/>			
Type	Element Name	Namespace	Xpath	NS Editor	Action	
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<a href="#">NameSpaces</a>		Delete
<a href="#">Add a Fact</a>						

The parameters available to configure an input fact are as follows.

Parameter Name	Description
<b>Type</b>	The fact type. You can select any of the registered fact types.
<b>Element Name</b>	The element name of the fact in the XML configuration.
<b>NameSpace</b>	The namespace for the element.

<b>XPath</b>	This parameter can be used to enter an XPath expression to create an input fact using a part of the input XML.
<b>NS Editor</b>	Click this link to edit the namespaces. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.
<b>Action</b>	This parameter can be used to delete and existing fact.

#### Output Facts

Output facts are the facts received by the rule service client after the rules in the rule set are applied.

Parameter Name	Description
<b>Wrapper Name</b>	The name of the wrapping element for all the output facts.
<b>NameSpace</b>	The namespace for the wrapping element.

You can click **Add a Fact** to add an output fact. The page will expand to display the following section.

#### Output Facts

Wrapper Name	<input type="text"/>	NameSpace	<input type="text"/>
Type	Element Name	Namespace	Action
<input type="text"/>	<input type="text"/>	<input type="text"/>	Delete

[Add a Fact](#)

The parameters available to configure an output fact are as follows.

Parameter Name	Description
<b>Type</b>	The fact type. You can select any of the <a href="#">registered fact types</a> .
<b>Element Name</b>	The fact type. You can select any of the <a href="#">registered fact types</a> .
<b>NameSpace</b>	The namespace of the element.
<b>Action</b>	This parameter can be used to delete and existing fact.

#### Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.

Mediator [switch to source view](#)

#### Example

In this example, the rule script is picked from the registry with key `rule/sample.xml`. There is one fact and it is a string variable. Its value is calculated from the current SOAP message using an expression. The Rule engine uses these facts to decide what rules should be applied.

```

<rule>
 <ruleset>
 <source key="rule/sample.xml"/>
 </ruleset>
 <facts>
 <fact name="symbol" type="java.lang.String"
 expression="//m0:getQuote/m0:request/m0:symbol/child::text()"
 xmlns:m0="http://services.samples"/>
 </facts>
 <childMediators>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 <drop/>
 </childMediators>

```

```
</rule>
```

## Script Mediator

The **Script Mediator** is used to invoke the functions of a variety of scripting languages such as JavaScript, Groovy, or Ruby.

WSO2 ESB uses Rhino engine to execute JavaScripts. Rhino engine converts the script to a method inside a Java class. Therefore, when processing large JSON data volumes, the code length must be less than 65536 characters, since the Script mediator converts the payload into a Java object. However, you can use the following alternative options to process large JSON data volumes.

- Achieve the same functionality via a [Class mediator](#).
- If the original message consists of repetitive sections, you can use the [Iterate mediator](#) to generate a relatively small payload using those repetitive sections. This will then allow you to use the Script mediator.

A Script mediator can be created in one of the following methods.

- With the script program statements stored in a separate file, referenced via the [Local or Remote Registry entry](#).
- With the script program statements embedded inline within the Synapse configuration.

Synapse uses the Apache Bean Scripting Framework for scripting language support. Any script language supported by BSF may be used to implement the Synapse Mediator. With the Script Mediator, you can invoke a function in the corresponding script. With these functions, it is possible to access the Synapse predefined in a script variable named `mc`. The `mc` variable represents an implementation of the `MessageContext`, named `ScriptMessageContext.java`, which contains the following methods that can be accessed within the script as `mc.methodName`.

Return?	Method Name	Description
Yes	getPayloadXML()	This gets an XML representation of SOAP Body payload.
No	setPayloadXML(payload)	This sets the SOAP body payload from XML.

No	addHeader(mustUnderstand, content)	This adds a new SOAP header to the message.
Yes	getEnvelopeXML()	This gets the XML representation of the complete SOAP envelope.
No	setTo(reference)	This is used to set the value which specifies the receiver of the message.
Yes	setFaultTo(reference)	This is used to set the value which specifies the receiver of the faults relating to the message.
No	setFrom(reference)	This is used to set the value which specifies the sender of the message.
No	setReplyTo(reference)	This is used to set the value which specifies the receiver of the replies to the message.
Yes	getPayloadJSON()	This gets the JSON representation of a SOAP Body payload.
No	setPayloadJSON(payload)	This sets the JSON representation of a payload obtained via the <code>getPayloadJSON()</code> method and sets it in the current message context.
Yes	getProperty(name)	This gets a property from the current message context.
No	setProperty(key, value)	This is used to set a property in the current message context. The previously set property values are replaced by this method.

Implementing a Mediator with a script language has advantages over using the built-in Synapse Mediator types or implementing a custom Java class Mediator. The Script Mediators have the flexibility of a class Mediator with access to the `SynapseMessageContext` and `SynapseEnvironment` APIs. Also, the ease of use and dynamic nature of scripting languages allow the rapid development and prototyping of custom mediators. An additional benefit of some scripting languages is that they have very simple and elegant XML manipulation capabilities, which make them very usable in a Synapse mediation environment. e.g., JavaScript E4X or Ruby REXML.

For both types of script mediator definitions, the `MessageContext` passed into the script has additional methods over the standard Synapse `MessageContext` to enable working with XML natural to the scripting language. Examples are when using JavaScript `getPayloadXML` and `setPayloadXML`, E4X XML objects and when using Ruby, REXML documents.

The Script mediator is a [content aware mediator](#).

## Syntax | UI Configuration | Examples

### Syntax

Click on the relevant tab to view the syntax for a script mediator using an Inline script, or a script mediator using a script of a registry

[Using an Inline script](#)[Using a script of the registry](#)

The following syntax applies when you create a Script mediator with the script program statements embedded inline within the Synapse configuration.

```
<script language="js"><! [CDATA[...script source code...]]></script>
```

The following syntax applies when you create a Script mediator with the script program statements stored in a separate file, referenced via the [Local or Remote Registry entry](#).

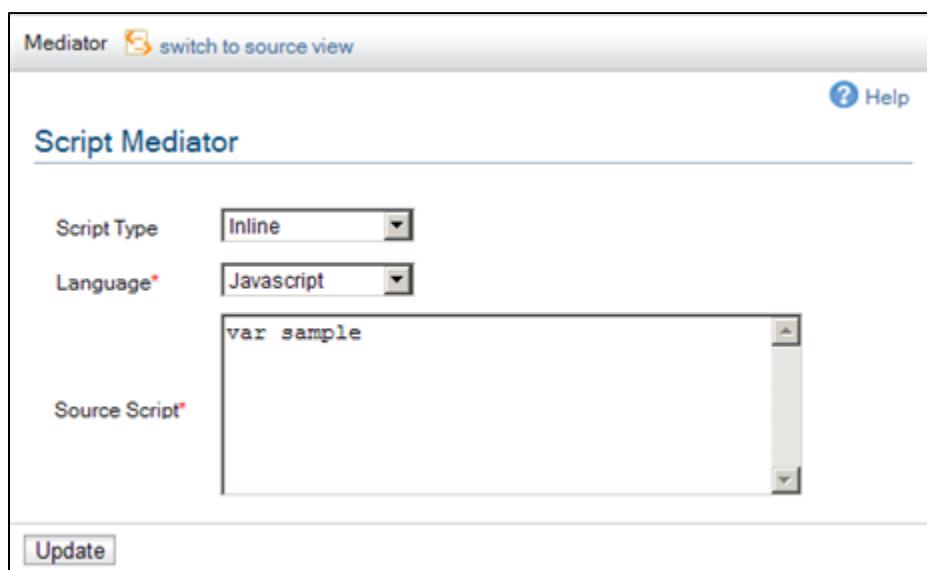
```
<script key="string" language="js" [function="script-function-name"]>
 <include key="string"/>
</script>
```

## UI Configuration

Click on the relevant tab to view the required UI configuration depending on the script type you have selected. The available script types are as follows:

- **Inline:** If this script type is selected, the script is specified inline.
- **Registry:** If this script type is selected, a script which is already saved in the registry will be referred using a key.

[Inline](#)Using a script of the registry



The parameters available to configure a Script mediator using an inline script are as follows.

Parameter Name	Description
<b>Language</b>	The scripting language for the Script mediator. You can select from the following available languages. <ul style="list-style-type: none"> <li>• JavaScript - This is represented as <code>js</code> in the source view.</li> <li>• Groovy - This is represented as <code>groovy</code> in the source view.</li> <li>• Ruby - This is represented as <code>rb</code> in the source view.</li> </ul>
<b>Source</b>	Enter the source in this parameter.

Mediator [switch to source view](#)

## Script Mediator

Help

Script Type	<input type="button" value="Registry Key"/>
Language*	<input type="button" value="Javascript"/>
Function*	<input type="text"/>
Key Type :	<input checked="" type="radio"/> Static Key <input type="radio"/> Dynamic Key
Key*	<input type="text"/> <a href="#">Configuration Registry</a> <a href="#">Governance Registry</a>
<b>Include keys</b>	
<a href="#">+ Add include key</a>	
<input type="button" value="Update"/>	

The parameters available to configure a Script mediator using a script saved in the registry are as follows.

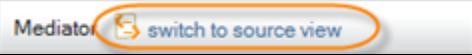
Parameter Name	Description
<b>Language</b>	The scripting language for the Script mediator. You can select from the following available languages. <ul style="list-style-type: none"> <li>• JavaScript - This is represented as <code>js</code> in the source view.</li> <li>• Groovy - This is represented as <code>groovy</code> in the source view.</li> <li>• Ruby - This is represented as <code>rb</code> in the source view.</li> </ul>
<b>Function</b>	The function of the selected script language to be invoked. This is an optional parameter. If no value is specified, a default function named <code>mediate</code> will be applied. This function considers the Synapse MessageContext as a single parameter. The function may return a boolean. If it does not, then the value <code>true</code> is assumed and the Script mediator returns this value.
<b>Key Type</b>	You can select one of the following options. <ul style="list-style-type: none"> <li>• <b>Static Key:</b> If this is selected, an existing key can be selected from the registry for the <b>Key</b> parameter.</li> <li>• <b>Dynamic Key:</b> If this is selected, the key can be entered dynamically in the <b>Key</b> parameter.</li> </ul>
<b>Key</b>	The <b>Registry</b> location of the source. You can click either <b>Configuration Registry</b> or the <b>Governance Registry</b> to select the source from the resource tree.

<b>Include keys</b> <p>This parameter allows you to include functions defined in two or more scripts your Script mediator configuration. After pointing to one script in the <b>Key</b> parameter, you can click <b>Add Include Key</b> to add the function in another script.</p> <p>When you click <b>Add Include Key</b>, the following parameters will be displayed. Enter the script to be included in the <b>Key</b> parameter by clicking either <b>Configuration Registry</b> or the <b>Governance Registry</b> and then selecting the relevant script from the resource tree.</p>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Include keys			
<input type="text" value="Key"/>	<input type="text"/>	<a href="#" style="color: blue;">Configuration Registry</a> <a href="#" style="color: blue;">Governance Registry</a>	Delete

## Note

You can configure the mediator using XML. Click **switch to source** view in the **Mediator** window.



## Examples

- Example 1 - Using an inline script
- Example 2 - Using a script saved in the registry
- Example 3 - Adding an Include key
- Example per method
- Samples

### Example 1 - Using an inline script

The following configuration is an example of an inline mediator using JavaScript/E4X which returns false if the SOAP message body contains an element named `symbol`, which has a value of `IBM`.

```
<script language="js"><![CDATA[mc.getPayloadXML()..symbol != "IBM";]]></script>
```

### Example 2 - Using a script saved in the registry

In the following example, script is loaded from the registry by using the key `repository/conf/sample/resources/script/test.js`.

```
<script language="js"
key="repository/conf/sample/resources/script/test.js"
function="testFunction"/>
```

`script language="js"` indicates that the function invoked should be in the JavaScript language. The function named `testFunction` which is invoked should be saved as a resource in the Registry. The script can be as shown in the example below.

```

function testFunction(mc) {
 var symbol = mc.getPayloadXML()..*::Code.toString();
 mc.setPayloadXML(
 <m:getQuote xmlns:m="http://services.samples/xsd">
 <m:request>
 <m:symbol>{symbol}</m:symbol>
 </m:request>
 </m:getQuote>);
}

```

### **Example 3 - Adding an Include key**

The following configuration has an `include` key.

```

<script language="js" key="stockquoteScript" function="transformRequest">
 <include key="sampleScript"/>
</script>

```

The script is written in JavaScript. The function to be executed is `transformRequest`. This function may be as follows in a script saved in the [Registry](#).

```

// stockquoteTransform.js
function transformRequest(mc) {
 transformRequestFunction(mc);
}

function transformResponse(mc) {
 transformResponseFunction(mc);
}

```

In addition, the function in the script named `sampleScript` which is included in the mediation configuration via the `include` key sub element is also executed in the mediation. Note that in order to do this, `sampleScript` script should also be saved as a resource in the [Registry](#). This script can be as follows.

```
// sample.js
function transformRequestFunction(mc) {
var symbol = mc.getPayloadXML()...*::Code.toString();
mc.setPayloadXML(
<m:getquote m="http://services.samples">
<m:request>
<m:symbol>{symbol}</m:symbol>
</m:request>
</m:getquote>);
}

function transformResponse(mc) {
var symbol = mc.getPayloadXML()...*::symbol.toString();
var price = mc.getPayloadXML()...*::last.toString();
mc.setPayloadXML(
<m:checkpriceresponse m="http://services.samples/xsd">
<m:code>{symbol}</m:code>
<m:price>{price}</m:price>
</m:checkpriceresponse>);
}
```

### **Example per method**

The following table contains examples of how some of the commonly used methods can be included in the script invoked by the following sample Script mediator configuration.

```
<script language="js"
key="conf:/repository/esb/transform.js"
function="transform"/>
```

Return?	Method Name	Example
Yes	getPayloadXML()	<p>The script invoked can be as follows.</p> <div style="border: 1px solid black; padding: 10px;"> <pre>// sample.js02.function transformRequestFunction(mc) { var symbol = mc.getPayloadXML()...*::Code.toString(); mc.setPayloadXML( &lt;m:getquote m="http://services.samples"&gt; &lt;m:request&gt; &lt;m:symbol&gt;{symbol}&lt;/m:symbol&gt; &lt;/m:request&gt; &lt;/m:getquote&gt; ); }</pre> </div> <p>mc.getPayloadXML( ) returns the response received in XML form.</p>
No	setPayloadXML(payload)	<p>See the example above for the getPayloadXML( ) method. mc.setPayl...*::symbol &lt;/m:request&gt; &lt;/m:getquote&gt; ) is used in that script to set the payload in the current message context.</p>

No	addHeader(mustUnderstand, Object content)	<p>The script invoked can be as follows.</p> <pre>&lt;script language="js"&gt; var wsse = new Namespace('http://docs.oasis-open.org/ws- var envelope = mc.getEnvelopeXML(); var username = envelope..wsse::Username.toString(); var password = envelope..wsse::Password.toString(); mc.addHeader(false, &lt;urn:AuthenticationInfo&gt;&lt;urn:userName&gt;{username}&lt;/urn:u- &lt;/script&gt;</pre> <p>The addHeader method configured as</p> <pre>mc.addHeader(false, &lt;urn:AuthenticationInfo&gt;&lt;urn:userName&gt;{username}&lt;/urn:u- above script is used to extract user name and password values included</pre>
No	getEnvelopeXML()	<p>The script invoked can be as follows.</p> <pre>&lt;script language="js"&gt; var wsse = new Namespace('http://docs.oasis-open.org/ws- var envelope = mc.getEnvelopeXML(); var username = envelope..wsse::Username.toString(); var password = envelope..wsse::Password.toString(); mc.addHeader(false, &lt;urn:AuthenticationInfo&gt;&lt;urn:userName&gt;{username}&lt;/urn:u- - See more at: <a href="http://sajithblogs.blogspot.com/2013/08/">http://sajithblogs.blogspot.com/2013/08/</a></pre>
Yes	getPayloadJSON()	<p>The script invoked can be as follows.</p> <pre>function transform(mc) {     payload = mc.getPayloadJSON();     results = payload.results;     var response = new Array();     for (i = 0; i &lt; results.length; ++i) {         location_object = results[i];         l = new Object();         l.name = location_object.name;         l.tags = location_object.types;         l.id = "ID:" + (location_object.id);         response[i] = l;     }     mc.setPayloadJSON(response); }</pre> <p>mc.getPayloadJSON() returns the JSON payload (received as the result of the script as shown in the above JavaScript code. See <a href="#">JSON Support</a> for further information.</p>

No	setPayloadJSON(payload)	See the example script for the <code>getPayloadJSON()</code> method. The <code>mc.setPayloadJSON()</code> method can be used to replace the existing incoming JSON payload and set that array object as the new payload. See
Yes	getProperty (name)	The script invoked can be as follows. <pre>&lt;script language="js"&gt; var time1 = mc.getProperty("TIME_1"); var time2 = mc.getProperty("TIME_2"); var timeTaken = time2 - time1; print("Time Duration : " + timeTaken + " ms "); mc.setProperty("RESPONSE_TIME", timeTaken); &lt;/script&gt;</pre> In this example, the <code>getProperty</code> method is used to get two time durations and then calculate the difference. The result is then set this difference in the message context.
No	setProperty(property)	See the example for the <code>getProperty</code> method. The <code>setProperty</code> method is used to set a property (name, value) in the message context.

## Samples

The following samples demonstrate how to use the Script mediator.

- [Sample 350: Introduction to the Script Mediator Using JavaScript](#)
- [Sample 351: Inline script mediation with JavaScript](#)
- [Sample 352: Accessing Synapse message context API using a scripting language](#)
- [Sample 353: Using Ruby Scripts for Mediation](#)
- [Sample 354: Using Inline Ruby Scripts for Mediation](#)

See also [Sample 441: Converting JSON to XML Using JavaScript](#)

## Send Mediator

The **Send Mediator** is used to send messages out of Synapse to an endpoint. The Send Mediator also copies any message context properties from the current message context to the reply message received on the execution of the send operation, so that the response could be correlated back to the request. Messages may be correlated by WS-A MessageID, or even simple custom text labels.

A send operation can be blocking or non-blocking depending on the actual transport implementation used. The default NIO-based http/s implementation does not block on a send. Therefore, if a message should be sent and further processed (e.g. transformed) afterwards, it is required to clone the message into two copies and then perform the processing to avoid conflicts.

The Send mediator is a [content-unaware](#) mediator.

Do not add any mediator configurations after Send mediator in the same sequence, because WSO2 ESB does not process them. Any mediator configuration after the Send mediator should go to the outSequence or receive sequence.

---

## Syntax | UI Configuration | Examples

---

## Syntax

```
<send/>
```

If the message is to be sent to one or more endpoints, use the following syntax.

```
<send>
 (endpointref | endpoint) +
</send>
```

- The endpointref token refers to the following:

```
<endpoint key="name" />
```

- The endpoint token refers to an anonymous endpoint definition.

## UI Configuration

Parameter Name	Description

<b>Select Endpoint Type</b>	<p>This parameter is used to specify the endpoint type to which the message should be sent. The available options are as follows.</p> <ul style="list-style-type: none"> <li>• <b>None:</b> If this is selected for a Send mediator included in the Out sequence, the message is not sent to any endpoint, but it will be sent back to the client. If this option is selected for a Send mediator included in the In sequence, the message will be sent to the URL specified in its To header.</li> <li>• <b>Define Inline:</b> If this is selected, the endpoint to which the message should be sent can be included within the Send mediator configuration. Click <b>Add</b> to add the required endpoint. See <a href="#">Adding an Endpoint</a> for further details.</li> <li>• <b>Pick from Registry:</b> If this is selected, the message can be sent to a pre-defined endpoint which is currently saved as a resource in the <a href="#">registry</a>. Click either <b>Configuration Registry</b> or <b>Governance Registry</b> as relevant to select the required endpoint from the resource tree.</li> <li>• <b>XPath:</b> If this is selected, the endpoint to which the message should be sent will be derived via an XPath expression. You are required to enter the relevant XPath expression in the text field that appears when this option is selected.</li> </ul> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p> </div>
<b>Receiving Sequence Type</b>	<p>The sequence to use for handling the response from the endpoint. Possible options are as follows.</p> <ul style="list-style-type: none"> <li>• <b>Default:</b> If this is selected, the mediation sequence in the Out sequence will be used.</li> <li>• <b>Static:</b> If this is selected, the sequence will be static. You can select a pre-defined sequence that is currently saved as a resource in the <a href="#">registry</a>. Click either <b>Configuration Registry</b> or <b>Governance Registry</b> as relevant to select the required sequence from the resource tree.</li> <li>• <b>Dynamic:</b> If this is selected, the sequence will be derived via an XPath expression. The XPath expression should be entered in the <b>Receiving Sequence</b> parameter which appears when this option is selected.</li> </ul> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p> </div>
<b>Build Message Before Sending</b>	<p>This parameter is used to specify whether the message should be built before sending or not. The possible values are as follows.</p> <ul style="list-style-type: none"> <li>• <b>Yes:</b> If this is selected, the full message XML is built in the memory before the message is sent. <b>Yes</b> should be selected if your configuration includes a logic that is performed after the Send has initiated.</li> <li>• <b>No:</b> If this is selected, the full message XML is not built in the memory before the message is sent. This improves performance.</li> </ul>

## Examples

### Example 1 - Send mediator used in the In sequence and Out sequence

In this example, the first send operation is included in the In mediator. Both the request and response will go through the main sequence, but only request messages will go through the In mediator. Similarly, only response messages

will go through the Out mediator. The request will be forwarded to the endpoint with the given address. The response will go through the second send operation, which in this example just sends it back to the client because there is no Out endpoint specified.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <in>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 <drop/>
 </in>
 <out>
 <send/>
 </out>
</definitions>
```

### **Example 2 - Specifying a response handling sequence (service chaining)**

```
<send receive="personInfoSeq">
 <endpoint key="PersonInfoEpr" />
</send>
```

In this example, requests are sent to the `PersonInfoEpr` endpoint, and responses from the service at that endpoint are handled by a sequence named `personInfoSeq`. This approach is particularly useful for service chaining. For example, if you want to take the responses from the `PersonInfoEpr` service and send them to the `CreditEpr` service for additional processing before sending the final response back to the client. In this case, you can configure the `personInfoSeq` sequence to send the response to the `CreditEpr` service and also specify another receive sequence named `creditSeq` that sends the response from the `CreditEpr` service back to the client. Following is the configuration of these sequences.

```
<sequence name="personInfoSeq">
 <xslt key="xslt">
 <property name="amount" expression="get-property('ORG_AMOUNT')"/>
 </xslt>
 <send receive="creditSeq">
 <endpoint key="CreditEpr" />
 </send>
</sequence>

<sequence name="creditSeq">
 <log level="full" />
 <send/>
</sequence>
```

### **Example 3 - Configuring a blocking/non-blocking send operation**

In this example, the Send mediator in a proxy service using the [VFS transport](#) is transferring a file to a VFS endpoint. VFS is a non-blocking transport by default, which means a new thread is spawned for each outgoing

message. The [Property mediator](#) added before the Send mediator removes the [ClientAPINonBlocking](#) property from the message to perform the mediation in a single thread. This is required when the file being transferred is large and you want to avoid out-of-memory failures.

```
<inSequence>
 <property name="ClientAPINonBlocking"
 value="true"
 scope="axis2"
 action="remove" />
 <send>
 <endpoint name="FileEpr">
 <address uri="vfs:file:///home/shammi/file-out" />
 </endpoint>
 </send>
</inSequence>
```

## Sequence Mediator

The **Sequence Mediator** refers to an already defined sequence element, which is used to invoke a named sequence of mediators. This is useful when you need to use a particular set on mediators in a given order repeatedly.

You can alternatively select a pre-defined sequence from the [registry](#) as the in/out/fault sequence for a proxy service or a REST service without adding any mediator configurations inline. The difference between these two options are described in the table below.

Attribute	Picking a pre-defined sequence as in/out/fault sequence	Referring to a pre-defined sequence via the Sequence mediator
Adding other mediators	Other mediator configurations that are not already included in the pre-defined sequence cannot be added to the in/out/fault sequence.	Other mediator configurations that are not already included in the pre-defined sequence can be added to the in/out/fault sequence
Applying changes done to the pre-defined sequence	Any changes done to the sequence saved in the <a href="#">registry</a> after it was selected as the in/out/fault sequence will not be considered when carrying out mediation.	Any changes done to the sequence saved in the <a href="#">registry</a> after it was selected as the in/out/fault sequence will be considered when carrying out mediation.

The Sequence mediator is a [content-unaware](#) mediator.

## Syntax | UI Configuration | Example

### Syntax

A sequence ref token refers to a `<sequence>` element, which is used to invoke a named sequence of mediators.

```
<sequence key="name" />
```

### UI Configuration

The screenshot shows the 'Sequence Mediator' configuration page. At the top, there's a 'Key Type' section with 'Static Key' selected. Below it is a 'Referring sequence' input field. To the right of the input field are two buttons: 'Configuration Registry' and 'Governance Registry'. At the bottom left is an 'Update' button.

The parameters available to configure the Sequence mediator are as follows.

Parameter Name	Description
<b>Key Type</b>	This parameter defines whether the key to access the required sequence is a static key or a dynamic key. Possible values are as follows. <ul style="list-style-type: none"> <li><b>Static Key:</b> If this is selected, the key to access the sequence is a static value. You can click either <b>Configuration Registry</b> or <b>Governance Registry</b> as relevant to select the require key from the resource tree for the <b>Referring Sequence</b> parameter.</li> <li><b>Dynamic Key:</b> If this is selected, you can define the key to access the sequence as a dynamic value by entering an XPath expression in the <b>Referring Sequence</b> parameter.</li> </ul>
<b>Referring sequence</b>	The key to access the sequence saved in the registry. You can enter a static value selected from the resource tree, or an XPath expression based on the option you selected for the <b>Key Type</b> parameter.

### Tip

You can click **NameSpaces** to add namespaces if you are providing an expression. Then the **Namespace Editor** panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.

### Example

In this example, the following sequence named `StoreSend` is saved in the Configuration registry. It includes a **Store Mediator** to store the request in a message store named `JMSMS` and a **Send Mediator** to send it to an endpoint afterwards.

```

<sequence xmlns="http://ws.apache.org/ns/synapse" name="conf:/StoreSend">
 <axis2ns4:store xmlns:axis2ns4="http://ws.apache.org/ns/synapse"
messageStore="JMSMS"
sequence="conf:/repository/components/org.wso2.carbon.throttle/templates"></axis2ns4:store>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"></address>
 </endpoint>
 </send>
</sequence>

```

The Sequence mediator configuration can be as follows to invoke the `StoreSend` sequence after using a **PayloadFactory**.

actory mediator to transform the contents of the request.

```
<inSequence xmlns="http://ws.apache.org/ns/synapse">
 <payloadFactory media-type="xml">
 <format>
 <m:checkpriceresponse xmlns:m="http://services.samples/xsd">
 <m:code>$1</m:code>
 <m:price>$2</m:price>
 </m:checkpriceresponse>
 </format>
 <args>
 <arg expression="//m0:symbol" xmlns:m0="http://services.samples/xsd">
 <arg expression="//m0:last" xmlns:m0="http://services.samples/xsd">
 </arg></arg></args>
 </payloadFactory>
 <sequence key="conf:/StoreSend"></sequence>
 </inSequence>
```

## Smooks Mediator

The **Smooks Mediator** can be used to apply lightweight transformations on messages in an efficient manner. Smooks is a powerful framework for processing, manipulating and transforming XML. More information about Smooks can be obtained from the official [Smooks website](#).

[Syntax](#) | [UI configuration](#) | [Examples](#)

### Syntax

```
<smooks [config-key="string"]>
 <input [type="|text|xml"]/>
 <output [type="|text|xml|java"] [property="string"] [action="string"]/>
</smooks>
```

### UI configuration

The screenshot shows the WSO2 ESB configuration interface for the Smooks Mediator. At the top, there's a navigation bar with 'Mediator' and a 'switch to source view' link. Below it, the title 'Smooks Mediator' is displayed. The configuration form includes fields for 'Config-Key\*' (with a red asterisk indicating it's required), 'Input' (set to 'XML'), 'Expression' (empty), 'Output' (set to 'XML'), and a 'Namespaces' section with an 'Add' button. There are also links to 'Configuration Registry' and 'Governance Registry'.

The parameters available for configuring the Smooks mediator are as follows:

Parameter Name	Description
----------------	-------------

<b>Config-Key</b>	The key to access the Smooks configuration. The Smooks configuration should be saved in the <a href="#">Registry</a> as a local entry before it can be used here. Click either <b>Configuration Registry</b> or <b>Governance Registry</b> to select the Smooks configuration from the resource tree.
<b>Input</b>	You can select either <b>XML</b> or <b>Text</b> as the input.
<b>Expression</b>	This parameter is used to enter an XPath expression to select the exact message block to which the transformation should be applied. If no expression is entered, the transformation would apply to the entire message body by default.
<b>Output</b>	<p>The format of the output. The output type can be XML, Text or Java, and the output can be one of the following.</p> <ul style="list-style-type: none"> <li>• <b>Property:</b> If this is selected, the output defined as a property will be saved in the message context for future use.</li> <li>• <b>Expression:</b> If this is selected, the output is defined as an expression and the following additional actions can be performed.           <ul style="list-style-type: none"> <li>• <b>Add:</b> The selected node will be added as a child to the message.</li> <li>• <b>Replace:</b> Selected node will be replaced in the message.</li> <li>• <b>Sibling:</b> Selected node will be added as a sibling.</li> </ul> </li> </ul>

## Examples

### **Example 1: Performance tuning**

Smooks can be used to split a file and send split results to a JMS endpoint. In this case, having a value other than -1 for `jms:highWaterMark` in the Smooks configuration file can result in a low throughput for message publishing, since Smooks will spend resources on message counting while the messages are being published. Therefore, it is recommended to use -1 as the `highWaterMark` value for high throughput values. The following is a sample Smooks configuration file with this setting. For more information on creating the Smooks configuration file, see the documentation in the official [Smooks website](#).

## Sample Smooks configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd"
xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd"
xmlns:jms="http://www.milyn.org/xsd/smooks/jms-routing-1.2.xsd">
 <!--
 Filter the message using the SAX Filter (i.e. not DOM, so no
 intermediate DOM, so we can process huge messages...
 -->
 <core:filterSettings type="SAX" />
 <!-- Capture the message data 2 seperate DOM model, for "order" and "order-item"
fragments... -->
 <resource-config selector="order,order-item">
 <resource>org.milyn.delivery.DomModelCreator</resource>
 </resource-config>
 <!-- At each "order-item", apply a template to the "order" and "order-item" DOM
model... -->
 <ftl:freemarker applyOnElement="order-item">
 <!--
 Note in the template that we need to use the special FreeMarker variable ".vars"
 because of the hyphenated variable names ("order-item"). See
 http://freemarker.org/docs/ref_specvar.html.
 -->
 <ftl:template>/repository/resources/orderitem-split.ftl.txt</ftl:template>
 <ftl:use>
 <!-- Bind the templating result into the bean context, from where
 it can be accessed by the JMSRouter (configured above). -->
 <ftl:bindTo id="orderItem_xml" />
 </ftl:use>
 </ftl:freemarker>
 <!-- At each "order-item", route the "orderItem_xml" to the ActiveMQ JMS Queue...
 -->
 <jms:router routeOnElement="order-item" beanId="orderItem_xml"
destination="smooks.exampleQueue">
 <jms:message>
 <!-- Need to use special FreeMarker variable ".vars" -->

 <jms:correlationIdPattern>${order.@id}-${.vars["order-item"].@id}</jms:correlationIdPa
ttern>
 </jms:message>
 <jms:jndi properties="/repository/conf/jndi.properties" />
 <jms:highWaterMark mark="-1" />
 </jms:router>
 </smooks-resource-list>
```

### Example 2: Referring files from the Smooks configuration

The following Smooks configuration refers a bindings file named `mapping.xml`. This file should be saved in `<ESB_HOME>/TESTDev` directory in order to be accessed via the class path. These bindings will be applied during mediation when the Smooks configuration is included in a Smooks mediator configuration via the [Registry](#).

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.0.xsd">
 <resource-config selector="org.xml.sax.driver">
 <resource>org.milyn.smooks.edi.EDIReader</resource>
 <param name="mapping-model">TESTDev/smooks/mapping.xml</param>
 </resource-config>
</smooks-resource-list>
```

## Samples

Sample 654: Smooks Mediator.

### Spring Mediator

The **Spring Mediator** exposes a spring bean as a mediator. The Spring Mediator creates an instance of a mediator, which is managed by Spring. This Spring bean must implement the `Mediator` interface for it to act as a Mediator.

The Spring mediator is a [content aware mediator](#).

---

[Syntax](#) | [UI Configuration](#) | [Example](#)

---

### Syntax

```
<spring:spring bean="exampleBean" key="string"/>
```

The attributes of the `<spring>` element:

- **key** - References the Spring ApplicationContext/Configuration (i.e. spring configuration XML) used for the bean. This key can be a [registry key](#) or [local entry key](#).
- **bean** - Is used for looking up a Spring bean from the spring Application Context. Therefore, a bean with the same name must be in the given spring configuration. In addition, bean must implement the `Mediator` interface.

---

### UI Configuration

The screenshot shows the 'Spring Mediator' configuration page. At the top, there's a 'switch to source view' link and a 'Help' button. The main section has two input fields: 'Bean' (marked with a red asterisk) and 'Key' (also marked with a red asterisk). Below these fields are two links: 'Configuration Registry' and 'Governance Registry'. At the bottom is a large 'Update' button.

These are the options for the Spring Mediator:

- **Bean** - Is used for looking up a Spring bean from the spring Application Context.
- **Key** - The [registry](#) reference to the spring Application-Context/Configuration used for the bean. You can select it by clicking the "Configuration Registry" or "Governance Registry" links.

## Note

You can configure the Mediator using XML. Click on "switch to source view" in the "Mediator" window.



Mediator  switch to source view

### Example

```
<spring:spring bean="springtest" key="conf/sample/resources/spring/springsample.xml"/>
```

In the above configuration, the spring XML is in the registry and it can be looked up using the registry key `conf/sample/resources/spring/springsample.xml`. This spring XML (i.e `springsample.xml`) must contain a bean with the name `springtest`. The following figure shows an example that can be used as the registry resource - `springsample.xml`.

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
 <bean id="springtest" class="org.apache.synapse.mediators.spring.SpringTestBean"
singleton="false">
 <property name="testProperty" value="100"/>
 </bean>
</beans>
```

Also, you need to build the JAR file of the following Spring Bean class and place it in the `<ESB_HOME>/repository/components/lib/` directory.

```
package org.apache.synapse.mediators.spring;

import org.apache.synapse.MessageContext;
import org.apache.synapse.mediators.AbstractMediator;

public class SpringTestBean extends AbstractMediator{
 private String testProperty;

 public void setTestProperty(String testProperty){
 this.testProperty = testProperty;
 }

 public boolean mediate(MessageContext mc) {
 // Do something useful..
 // Note the access to the Synapse Message context
 return true;
 }
}
```

For more examples, see [Mediating with Spring](#).

### Store Mediator

The **Store mediator** enqueues messages passing through its mediation sequence in a given message store. It can

serve as a [dead letter channel](#) if it is included in a fault sequence and if its message store is connected to a [message processor](#) that forwards all the messages in the store to an endpoint.

## Syntax

The Store mediator is a [content aware mediator](#).

```
<axis2ns1:store xmlns:axis2ns1="http://ws.apache.org/ns/synapse" messageStore="string"
sequence="string"></axis2ns1:store>
```

## UI Configuration

Mediator [switch to source view](#)

**Store Mediator**

Message Store \*

On Store Sequence

[Configuration Registry](#) [Governance Registry](#)

The parameters available to configure the Store mediator is as follows.

Parameter Name	Description
<b>Message Store</b>	Select the message store where messages will be stored. You should <a href="#">add the message store</a> to the ESB before you can select it here.
<b>On Store Sequence</b>	The <a href="#">sequence</a> that will be called before the message gets stored. This <a href="#">sequence</a> should be pre-defined in the registry before it can be entered here. Click either <b>Configuration Registry</b> or <b>Governance Registry</b> to select the required sequence from the resource tree. For more information on configuring the Registry, go to <a href="#">Working with the Registry</a> in the Common Admin Guide.

## Example

A proxy service can be configured with the Store mediator as follows to save messages in a message store named JMSMS.

```
<proxy name="SimpleProxy" transports="http https" startonload="true" trace="disable">
 <target>
 <insequence>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"
type="STRING"></property>
 <property name="OUT_ONLY" value="true" scope="default"
type="STRING"></property>
 <store messagestore="JMSMS"></store>
 </insequence>
 </target>
</proxy>
```

## Switch Mediator

The **Switch Mediator** is an XPath or JSONPath filter. The XPath or JSONPath is evaluated and returns a string. This string is matched against the regular expression in each switch case mediator, in the specified order. If a matching case is found, it will be executed, and the remaining switch case mediators are not processed. If none of the case statements are matching, and a default case is specified, the default will be executed.

The Switch mediator is a [conditionally content aware mediator](#).

---

## Syntax | UI Configuration | Example

### Syntax

```
<switch source="[XPath|json-eval(JSON Path)]">
 <case regex="string">
 mediator+
 </case>+
 <default>
 mediator+
 </default>?
</switch>
```

---

### UI Configuration

The screenshot shows the configuration interface for the Switch Mediator. At the top, there's a navigation bar with 'Mediator' and 'switch to source view'. On the right, there's a 'Help' link. Below the navigation, the title 'Switch Mediator' is displayed. The configuration form includes:

- Source XPath \***: A text input field.
- Number of cases**: A value '0 cases' next to a button labeled '+ Add case'.
- Specify default case**: A button with a plus sign.
- Update**: A button at the bottom left.

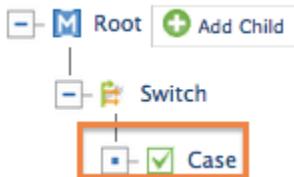
The parameters available to configure the Switch mediator are as follows.

Parameter Name	Description
<b>Source XPath</b>	The source XPath or JSONPath to be evaluated. When specifying a JSONPath, use the format <code>json-eval(&lt;JSON_PATH&gt;)</code> , such as <code>json-eval(getQuote.request.symbol)</code> . For more information on using JSON with the ESB, see <a href="#">JSON Support</a> . If you use <b>Namespaces</b> in the expression, click <b>Namespaces</b> and map the namespace prefix to the correct URI.

<b>Number of cases</b>	This parameter displays the number of cases currently added to the Switch mediator configuration. See <a href="#">Switch-case-mediator</a> for instructions to add a case.
<b>Specify default case</b>	Click this link to add a default switch-case mediator. Adding a default switch case mediator is optional. If it is specified, it will be executed if no matching switch-case is identified.

#### Switch-case mediator

1. To add a case, click **Add case**, which adds an empty switch-case mediator under the Switch mediator. A switch-case mediator would appear as a child of the Switch mediator in the mediator tree as shown below.



2. Click **Case** to configure the switch-case mediator. The page will expand to display the section shown below where a regular expression can be added in the **Case Value (Regular Expression)** parameter.

Mediator	<a href="#">switch to source view</a>	<a href="#">Help</a>
<b>Switch-Case Mediator</b>		
Case Value (Regular Expression) * <input type="text" value="/m0:getQuote/m0:request/m0:sym"/>		
<input type="button" value="Update"/> <input type="button" value="Save &amp; Close"/> <input type="button" value="Save"/> <input type="button" value="Cancel"/>		

3. Click **Case** again and click **Add Child**, and add the mediator(s) you want to execute when this case matches the switching value.

#### Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.



#### Example

In this example the **Property mediator** sets the local property named `symbol` on the current message depending on the evaluation of the string. It will get the text of `symbol` element and match it against the values `MSFT` and `IBM`. If the text does not match either of these symbols, the default case will be executed.

```

<switch source="//m0:getQuote/m0:request/m0:symbol"
xmlns:m0="http://services.samples/xsd">
 <case regex="IBM">
 <!-- the property mediator sets a local property on the *current* message
-->
 <property name="symbol" value="Great stock - IBM"/>
 </case>
 <case regex="MSFT">
 <property name="symbol" value="Are you sure? - MSFT"/>
 </case>
 <default>
 <!-- it is possible to assign the result of an XPath or JSON Path expression
as well -->
 <property name="symbol"
 expression="fn:concat('Normal Stock - ',
//m0:getQuote/m0:request/m0:symbol)"
 xmlns:m0="http://services.samples/xsd"/>
 </default>
</switch>

```

## Throttle Mediator

The **Throttle Mediator** can be used to restrict access to services. This is useful when services used at the enterprise level and it is required to avoid heavy loads that can cause performance issues in the system. It can also be used when you want to avoid certain user groups (i.e. IP addresses and domains) accessing your system. The Throttle mediator defines a throttle group which includes the following.

- A throttle policy which defines the extent to which individual and groups of IP addresses/domains should be allowed to access the service.
- A mediation sequence to handle requests that were accepted based on the throttle policy.
- A mediation sequence to handle requests that were rejected based on the throttle policy.

The Validate mediator is a [content unaware](#) mediator.

---

## Syntax | UI Configuration | Example

### Syntax

```

<throttle [onReject="string"] [onAccept="string"] id="string">
 (<policy key="string"/> | <policy>..</policy>)
 <onReject>..</onReject>?
 <onAccept>..</onAccept>?
</throttle>

```

---

### UI Configuration

The UI of the Throttle mediator are divided into following sections. Before you edit these sections, enter an ID for the Throttle group in the **Throttle Group ID** parameter.

- **Throttle Policy**
- **On Acceptance**
- **On Rejection**

**Throttle Policy**

This section is used to specify the throttle policy that should apply to requests passing through the Throttle mediator. A throttle policy can have a number of entries, each defining the extent to which an individual or a group of IP Addresses/domains should be allowed to access the service to which throttling is applied.

The parameters available to be configured in this section are as follows.

Parameter Name	Description
<b>Throttle Policy</b>	This section is used to specify the policy for throttling. The following options are available. <ul style="list-style-type: none"> <li>• <b>In-Lined Policy:</b> If this is selected, the Throttle policy can be defined within the Throttle mediator configuration. Click <b>Throttle Policy Editor</b> to open the <b>Mediator Throttling Configuration</b> dialog box where the details relating to the Throttle policy can be entered. The parameters in this dialog box are described in the table below.</li> <li>• <b>Referring Policy:</b> If this is selected, you can refer to a pre-defined Throttle policy which is saved in the <b>Registry</b>. You can enter the key to access the policy in the <b>Referring Policy</b> parameter. Click on either <b>Configuration Registry</b> or <b>Governance Registry</b> to select the relevant policy from the Resource Tree.</li> </ul>

The parameters available in the **Mediator Throttling Configuration** dialog box to configure the Throttling policy are as follows.

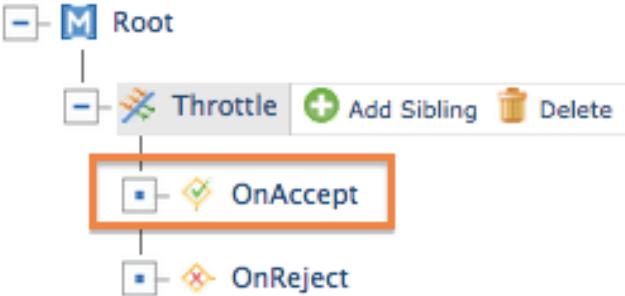
Parameter Name	Description
----------------	-------------

<b>Maximum Concurrent Accesses</b>	The maximum number of messages that are served at a given time. The number of messages between the inflow throttle handler and the outflow throttle handler cannot exceed the value entered for this parameter at any given time. This parameter value is applied to the entire system. It is not restricted to one or more specific IP addresses/domains. <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;">         When this parameter is used, the same Throttle mediator ID should be included in the response flow so that the completed responses are deducted from the available limit.       </div>
<b>Range</b>	This parameter is used to specify the IP addresses/domains to which the entry in the current row should be applied <ul style="list-style-type: none"> <li>If you want to apply the entry to a range of IP addresses, enter the range in this parameter, e.g., 8.100.1.30 – 8.100.1.45. Alternatively, you can enter a single IP address to apply the entry to only one IP address.</li> <li>If you want to apply the entry to a domain, enter the required domain ID in this parameter.</li> <li>If you want to apply the entry to all the IP addresses/domains that are not configured in the other configurations, enter <b>other</b> in this parameter.</li> </ul>
<b>Type</b>	This parameter is used to specify whether the value(s) entered in the <b>Range</b> parameter refers to IP addresses or domains.
<b>Max Request Count</b>	This parameter specifies the maximum number of requests that should be handled within the time interval specified in the <b>Unit Time</b> parameter. <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;">         This parameter is applicable only when the value selected for the <b>Access</b> parameter is <b>Control</b>.       </div>
<b>Unit Time (ms)</b>	The time interval for which the maximum number of requests specified for the Throttle ID in the <b>Max Request Count</b> parameter apply. <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;">         This parameter is applicable only when the value selected for the <b>Access</b> parameter is <b>Control</b>.       </div>
<b>Prohibit Time Period (ms)</b>	If the number of requests entered in the <b>Max Request Count</b> parameter is achieved before the time interval entered in the <b>Unit Time (ms)</b> parameter has elapsed, no more requests are taken by the inflow throttle handler for the time period entered in this parameter. Entering a value in this parameter alters the unit time. <p>For example:</p> <p>Max Request Count = 50          Unit Time = 50000 ms          Prohibit Time Period = 5000 ms</p> <p>If 50 requests are received within 50000 milliseconds, no requests will be taken for the next 5000 milliseconds. Thus, the time slot considered as the unit time is changed to 40000 milliseconds. If no value is entered in the <b>Prohibit Time Period (ms)</b> parameter, no requests will be taken until 15000 more milliseconds (i.e. the remainder of the unit time) have elapsed.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;">         This parameter is applicable only when the value selected for the <b>Access</b> parameter is <b>Control</b>.       </div>

<b>Access</b>	<p>This parameter is used to specify the extent to which the IP addresses/domains specified in the <b>Range</b> parameter are allowed access to the service to which the throttle policy is applied. Possible values are as follows.</p> <ul style="list-style-type: none"> <li><b>Allow:</b> If this is selected, the specified IP addresses/domains are allowed to access the services to which the throttle ID is applied without any restrictions.</li> <li><b>Deny:</b> If this is selected, specified IP addresses/domains are not allowed to access the services to which the throttle ID is applied.</li> <li><b>Control:</b> If this is selected, the specified IP addresses/domains are allowed to access the services to which the throttle ID is applied. However, the number of times they can access the services is controlled by the <b>Max Request Count, Unit Time (ms)</b> and the <b>Prohibit Time Period (ms)</b> parameters.</li> </ul>
<b>Action</b>	This parameter can be used to delete the entry.

#### On Acceptance

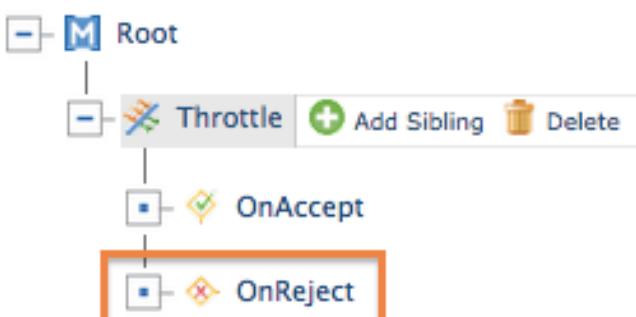
This section is used to specify the mediation sequence that should be applied when a request is accepted based on the **throttle** policy defined for the Throttle mediator. The parameters available to be configured in this section are as follows.

Parameter Name	Description
<b>Specify As</b>	<p>This parameter is used to specify how the On Acceptance sequence is defined. The following options are available.</p> <ul style="list-style-type: none"> <li><b>In-Lined Policy:</b> If this is selected, the mediation sequence to be applied to accepted requests can be defined within the Throttle mediator configuration. Click on the <b>OnAccept</b> node in the mediation tree to define the sequence in-line.</li> </ul>  <ul style="list-style-type: none"> <li><b>Referring Policy:</b> If this is selected, you can refer to a pre-defined mediation sequence in the <b>registry</b>. Click either <b>Configuration Registry</b> or <b>Governance Registry</b> as relevant to select the required sequence from the Resource Tree.</li> </ul>

#### On Rejection

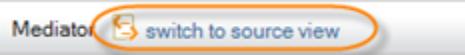
This section is used to specify the mediation sequence that should be applied when a request is rejected based on the **throttle** policy defined for the Throttle mediator. The parameters available to be configured in this section are as follows.

Parameter Name	Description

<p><b>Specify As</b></p> <p>This parameter is used to specify how the On Acceptance sequence is defined. The following options are available.</p> <ul style="list-style-type: none"> <li>• <b>In-Lined Policy:</b> If this is selected, the mediation sequence to be applied to rejected requests can be defined within the Throttle mediator configuration. Click on the <b>OnReject</b> node in the mediation tree to define the sequence in-line.</li> </ul>  <ul style="list-style-type: none"> <li>• <b>Referring Policy:</b> If this is selected, you can refer to a pre-defined mediation sequence in the registry. Click either <b>Configuration Registry</b> or <b>Governance Registry</b> as relevant to select the required sequence from the Resource Tree.</li> </ul>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.



## Example

In this example, the Throttle Mediator inside the [In Mediator](#). Therefore, all request messages directed to the main sequence will be subjected to throttling. The Throttle Mediator has `policy`, `onAccept` and `onReject` tags at top level. The `policy` tag specifies the throttling policy for throttling messages.

The `onAccept` sequence includes a [Log mediator](#) with a custom log to log the accepted requests. Then the [Send mediator](#) sends these requests to `http://localhost:9000/services/SimpleStockQuoteService`.

The `OnReject` sequence too includes a [Log mediator](#) with a custom log to log the rejected requests. Then a [Fault mediator](#) is used to convert the message into a fault message. The fault message is then returned to the client as a response using the [Respond mediator](#), and then dropped using the [Drop mediator](#).

### Example for a concurrency-based policy

This sample policy only contains a component called `MaximumConcurrentAccess`. This indicates the maximum number of concurrent requests that can pass through Synapse on a single unit of time, and this value applies to all the IP addresses and domains.

```

<in>
 <throttle id="A">
 <policy>
 <!-- define throttle policy -->
 <wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
 xmlns:throttle="http://www.wso2.org/products/wso2commons/throttle">
 <throttle:ThrottleAssertion>

 <throttle:MaximumConcurrentAccess>10</throttle:MaximumConcurrentAccess>
 </throttle:ThrottleAssertion>
 </wsp:Policy>
 </policy>
 <onAccept>
 <log level="custom">
 <property name="text" value="***Access Accept***"/>
 </log>
 <send>
 <endpoint>
 <address
 uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </onAccept>
 <onReject>
 <log level="custom">
 <property name="text" value="***Access Denied***"/>
 </log>
 <makefault>
 <code value="tns:Receiver"
 xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason value="***Access Denied***"/>
 </makefault>
 <respond/>
 <drop/>
 </onReject>
 </throttle>
 </in>

```

### **Example for a rates-based policy**

This sample policy only contains a rates-based policy. This indicates the maximum number of concurrent requests that can pass through Synapse on a single unit of time, and this value applies to all the IP addresses and domains.

```

<in>
 <throttle id="A">
 <policy>
 <!-- define throttle policy -->
 <wsp:Policy>
 <throttle:MaximumCount>4</throttle:MaximumCount>
 <throttle:UnitTime>800000</throttle:UnitTime>
 <throttle:ProhibitTimePeriod
wsp:Optional="true">1000</throttle:ProhibitTimePeriod>
 </wsp:Policy>
 </policy>
 <onAccept>
 <log level="custom">
 <property name="text" value="**Access Accept**"/>
 </log>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </onAccept>
 <onReject>
 <log level="custom">
 <property name="text" value="**Access Denied**"/>
 </log>
 <makefault>
 <code value="tns:Receiver"
 xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason value="**Access Denied**"/>
 </makefault>
 <respond/>
 <drop/>
 </onReject>
 </throttle>
</in>

```

## Transaction Mediator

The **Transaction Mediator** is used to provide transaction functionality for its child mediators.

---

[Syntax](#) | [UI configuration](#) | [Example](#)

---

### Syntax

```

<syn:transaction
action="commit|fault-if-no-tx|new|resume|suspend|rollback|use-existing-or-new" />

```

---

### UI configuration

The Action parameter is used to select a transaction action to be performed. Available values are as follows.

Action	Description
<b>Commit Transaction</b> (commit)	This marks the transaction as completed and ends the transaction.
<b>Fault if no Transaction</b> (fault-if-no-tx)	This goes to the error handler if there is no transaction.
<b>Initiate new Transaction</b> (new)	This provides the entry point for a new transaction.
<b>Resume Transaction</b> (resume)	This resumes a paused transaction.
<b>Suspend Transaction</b> (suspend)	This pauses a transaction.
<b>Rollback Transaction</b> (rollback)	This rolls back a transaction.
<b>Use existing or Initiate Transaction</b> (use-existing-or new)	If a transaction already exists, this value continues it. If no transaction already exists, a new transaction will be created.

## Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.

Mediator

## Example

For an example of using the Transaction mediator, see [Transaction Mediator Example](#).

### Transaction Mediator Example

The [Transaction mediator](#) supports [distributed transactions](#) using the Java transaction API (JTA). When it comes to distributed transactions, it involves accessing and updating data on two or more networked computers, often using multiple databases. For example, two databases or a database and a message queue such as JMS. You can use the Synapse configuration language to define when to start, commit, or roll back the transaction. For example, you can mark the start of a transaction at the start of a database commit, mark the end of the transaction at the end of database commit and roll back the transaction if an error occurs.

Let's explore a basic transaction mediator scenario that demonstrates how the transaction mediator can be used to manage complex distributed transactions.

[Transaction mediator scenario](#) | [Prerequisites](#) | [Configuring the example scenario](#) | [Testing the example scenario](#)

Transaction mediator scenario

You have a record in one database and you want to delete that record from the first database and add it to a second database. Assume that these two databases can either be run on the same server or can be in two remote servers. The database tables are defined in a way that the same entry cannot be added twice. Therefore, in the successful scenario, the record will be deleted from the first database and will be added to the second database. In the failure scenario, the record is already in the second database, hence the record will not be deleted from the first table nor will it be added into the second database.

#### Prerequisites

- Windows, Linux or Solaris operating systems with WSO2 ESB installed.
- Apache Derby database server. If you do not have the Apache Derby database set up, download the Apache Derby distribution from <http://db.apache.org>.

#### Configuring the example scenario

1. Copy and paste the following configuration into \$ESB\_HOME/repository/deployment/server/synapse-e-configs/<node>/synapse.xml.

The sample configuration used here is similar to [sample 361](#). Here via the *In* sequence a message is sent to the service, and via the *Out* sequence an entry from the first database is deleted and the second database is updated with that entry. If an entry that already exists is added once again to the second database, the entire transaction will roll back.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="myFaultHandler">
 <log level="custom">
 <property name="text" value="** Rollback Transaction**"/>
 </log>
 <transaction action="rollback" />
 <send/>
 </sequence>
 <sequence name="main" onError="myFaultHandler">
 <in>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </in>
 <out>
 <transaction action="new" />
 <log level="custom">
 <property name="text" value="** Reporting to the Database
esbdb**"/>
 </log>
 <dbreport useTransaction="true"
xmlns="http://ws.apache.org/ns/synapse">
 <connection>
 <pool>
 <dsName>java:jdbc/XADerbyDS</dsName>

 <icClass>org.jnp.interfaces.NamingContextFactory</icClass>
 <url>localhost:1099</url>
 <user>esb</user>
 <password>esb</password>
 </pool>
 </connection>
 <statement>
 <sql>delete from company where name =?</sql>
 </statement>
 </dbreport>
 </out>
</sequence>

```

```

<parameter expression="//m0:return/m1:symbol/child::text()"
 xmlns:m0="http://services.samples"
 xmlns:m1="http://services.samples/xsd"
 type="VARCHAR" />
 </statement>
</dbreport>
<log level="custom">
 <property name="text" value="*** Reporting to the Database
esbdb1***"/>
</log>
<dbreport useTransaction="true"
xmlns="http://ws.apache.org/ns/synapse">
 <connection>
 <pool>
 <dsName>java:jdbc/XADerbyDS1</dsName>

<icClass>org.jnp.interfaces.NamingContextFactory</icClass>
 <url>localhost:1099</url>
 <user>esb</user>
 <password>esb</password>
 </pool>
 </connection>
 <statement>
 <sql>INSERT into company values (?,?,?,?,?)</sql>
 <parameter expression="//m0:return/m1:symbol/child::text()"
 xmlns:m1="http://services.samples/xsd"
 xmlns:m0="http://services.samples"
 type="VARCHAR" />
 <parameter expression="//m0:return/m1:last/child::text()"
 xmlns:m1="http://services.samples/xsd"
 xmlns:m0="http://services.samples"
 type="DOUBLE" />
 </statement>
 </dbreport>
 <transaction action="commit" />
 <send/>
</out>

```

```
</sequence>
</definitions>
```

2. Setup two distributed Derby databases `esbdb` and `esbdb1`. For instructions on setting up the Derby databases, see [Setting up the Derby database server](#).
3. Create a table in `esbdb` by executing the following statement.

```
CREATE table company(name varchar(10) primary key, id varchar(10), price double);
```

4. Create a table in `esbdb1` by executing the following statement.

```
CREATE table company(name varchar(10) primary key, id varchar(10), price double);
```

5. Insert records into the two tables that you created by executing the following statements.

#### To insert records into the table in esbdb

```
INSERT into company values ('IBM','c1',0.0);
INSERT into company values ('SUN','c2',0.0);
```

#### To insert records into the table in esbdb1

```
INSERT into company values ('SUN','c2',0.0);
INSERT into company values ('MSFT','c3',0.0);
```

#### Note

When inserting records into the tables, the order of the record matters.

6. In the `master-datasources.xml` file located in the `<ESB_HOME>/repository/conf/datasources` directory, create data source declarations for the distributed databases, ensuring that the datasource file names are `*-xa-ds.xml`:  
 Datasource1: `esb-derby-xa-ds.xml`

```

<datasources>
 <xa-datasource>
 <jndi-name>jdbc/XADerbyDS</jndi-name>
 <isSameRM-override-value>false</isSameRM-override-value>

 <xa-datasource-class>org.apache.derby.jdbc.ClientXADataSource</xa-datasource-class>
 <xa-datasource-property name="portNumber">1527</xa-datasource-property>
 <xa-datasource-property name="DatabaseName">esbdb</xa-datasource-property>
 <xa-datasource-property name="User">esb</xa-datasource-property>
 <xa-datasource-property name="Password">esb</xa-datasource-property>
 <metadata>
 <type-mapping>Derby</type-mapping>
 </metadata>
 </xa-datasource>
</datasources>

```

Datasource2: esb-derby1-xa-ds.xml

```

<datasources>
 <xa-datasource>
 <jndi-name>jdbc/XADerbyDS1</jndi-name>
 <isSameRM-override-value>false</isSameRM-override-value>

 <xa-datasource-class>org.apache.derby.jdbc.ClientXADataSource</xa-datasource-class>
 <xa-datasource-property name="portNumber">1527</xa-datasource-property>
 <xa-datasource-property name="DatabaseName">esbdb1</xa-datasource-property>
 <xa-datasource-property name="User">esb</xa-datasource-property>
 <xa-datasource-property name="Password">esb</xa-datasource-property>
 <metadata>
 <type-mapping>Derby</type-mapping>
 </metadata>
 </xa-datasource>
</datasources>

```

7. Deploy the back-end service SimpleStockQuoteService. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

8. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).

Testing the example scenario

### To test the successful scenario

- Execute the following command from the <ESB\_HOME>/samples/axis2Client directory.

```

ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=IBM

```

Executing this command removes the IBM record from the first database and adds it to the second database.

When you check both databases you will see that the IBM record is deleted from the first database and added to the second database.

## To test the failure scenario

- Execute the following command from the <ESB\_HOME>/samples/axis2Client directory.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=SUN
```

Executing this command attempts to add an already existing record again to the second database, which results in the fault sequence being executed. You will see an exception raised for duplicate entries and the entire transaction will roll back.

When you check both databases you will see that a record is neither deleted from the first database nor added into the second database.

## URLRewrite Mediator

The **URLRewrite Mediator** is used to modify and transform the URL values available in messages. This can be done by defining a rewrite action for each fragment of a selected property value. Alternatively, you can rewrite the entire URL string at once.

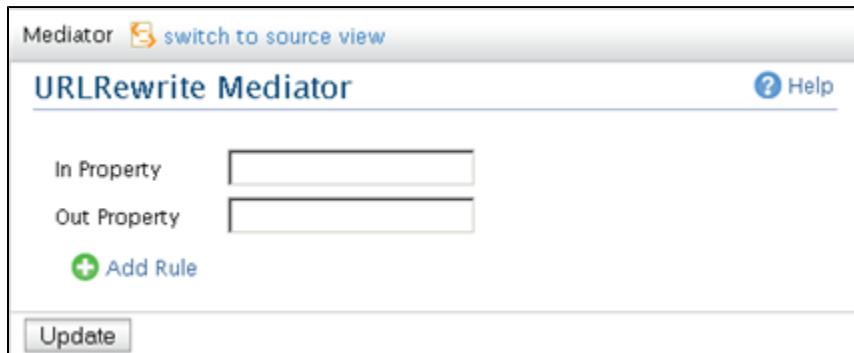
The URLRewrite mediator is a content aware mediator.

### Syntax | UI Configuration | Example

#### Syntax

```
<rewrite [inProperty="string"] [outProperty="string"]>
 <rewriterule>
 <condition>
 ...
 </condition>?
 <action [type="append|prepend|replace|remove|set"] [value="string"]
 [xpath="xpath"] [fragment="protocol|host|port|path|query|ref|user|full"]
 [regex="regex"]>+
 </rewriterule>+
 </rewrite>
```

#### UI Configuration



The parameters available to configure the URL Rewrite mediator are as follows.

Parameter Name	Description
In Property	This parameter is used to enter property of which the value should be considered the input URL. The rewrite rules are applied to the value of the property entered in this parameter to generate the result URL. If no property is entered, the rewrite rules will be applied to the <code>To</code> header of the message.
Out Property	This parameter is used to enter the property to which the transformations done via the rewrite rules should be applied. If no property is entered, the transformations will be applied to the <code>To</code> header of the message.

The Rewrite mediator applies the URL transformations by evaluating a set of rules on the message. To add a rule to the mediator, click **Add Rule**. The rewrite rule will be added to the mediator tree as a child of the URLRewrite mediator (see the image below).



Click `URLRewriteRule` in the mediator tree to configure the rewrite rule you added. The following section will be displayed.

Mediator [switch to source view](#)

### Rewrite Rule

Condition

Rules

[+ Add Action](#)

Update

Save & Close Save Cancel

A rule can consist of one or more rewrite actions and an optional condition. The **Condition** parameter is used to enter the optional condition. If a condition is specified, it will be evaluated before the rewrite actions, and the rewrite actions will be executed only if the condition evaluates to true.

A rewrite action is added by clicking **Add Action** which would display the following row.

**Rules**

Action	Fragment	Option	Value/Expression	Namespace Editor	Regex	Delete
Replace	Protocol	Value				 Delete
<a data-bbox="146 306 285 333" href="#"> Add Action</a> <a data-bbox="146 348 220 375" href="#">Update</a>						

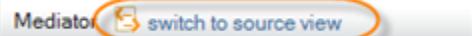
The parameters available to configure a rewrite action are as follows.

Parameter Name	Description
Action	<p>This parameter is used to specify the action to be performed by the rewrite action. Each rewrite action is performed on a fragment entered in the <b>Fragment</b> parameter. Possible values are as follows.</p> <ul style="list-style-type: none"> <li>• <b>Replace</b>: If this is selected, the existing <code>in property</code> value fragment will be replaced by the result value.</li> <li>• <b>Remove</b>: If this is selected, the result value will be removed from the <code>in property</code> value fragment.</li> <li>• <b>Append</b>: If this is selected, the result value will be added to the end of the <code>in property</code> value fragment.</li> <li>• <b>Prepend</b>: If this is selected, the result value will be added to the beginning of the <code>in property</code> value fragment.</li> <li>• <b>Set</b>: If this is selected, the result value will be set as the <code>in property</code> value fragment.</li> </ul>
Fragment	<p>The fragment of the <code>in property</code> (i.e. input URL) for which the rewrite action should be performed. The available fragments are as follows.</p> <ul style="list-style-type: none"> <li>• <b>Protocol</b>:</li> <li>• <b>Host</b></li> <li>• <b>Port</b></li> <li>• <b>Path</b></li> <li>• <b>Query</b></li> <li>• <b>Ref</b></li> <li>• <b>User</b></li> <li>• <b>Full</b></li> </ul> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p>Note that this breakdown is inline with the URI specification (RFC2396). URL rewrite mediator enables rewriting each of the above segments separately and finally combining them to get the final URL value. It also supports rewriting the entire URL string at once.</p> </div>
Option	<p>This parameter is used to define the result value of the rewrite action. Select one of the following.</p> <ul style="list-style-type: none"> <li>• <b>Value</b>: If this is selected, the result value would be a static value.</li> <li>• <b>Expression</b>: If this is selected, the result value will be evaluated using an expression.</li> </ul>
Value/Expression	This parameter is used to enter the result value of the URLRewrite mediator as a static value or an expression, depending on what you selected for the <b>Option</b> parameter.
Namespace Editor	You can click this link to add namespaces when you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.

<b>Regex</b>	This parameter is used to specify which part of the <code>in</code> property value fragment should be replaced by the result value if you selected <b>Replace</b> for the <b>Action</b> parameter.
<b>Delete</b>	Click <b>Delete</b> in the relevant row to remove a rewrite action.

## Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.



### Example

In this example, the URLRewrite mediator has a rewrite action which replaces the value `soap` with value `services` in the fragment `path` of the input URL. Since no `in` property or an `out` property is specified, the `To` header of the request is both the input to which the rewrite rule is applied and the target where the result URL is set. This configuration is typically used when the address URL of a request contains the context `soap` which needs to be converted since all the services are deployed under a context named `services` in the ESB server. Thus, the URL `http://localhost:8280/soap/StockQuoteProxy1` is rewritten as `http://localhost:8280/services/StockQuoteProxy1` to ensure that the requests are successfully delivered to the server.

```
<rewrite>
 <rewriterule>
 <action type="replace" regex="soap" value="services" fragment="path" />
 </rewriterule>
</rewrite>
```

### Samples

For more examples, see:

- [Sample 450: Introduction to the URLRewrite Mediator](#)
- [Sample 451: Conditional URL Rewriting](#)
- [Sample 452: Conditional URL Rewriting with Multiple Rules](#)

### Validate Mediator

You can use the Validate mediator to validate XML and JSON messages.

[Validating XML messages](#) [Validating JSON messages](#)

The Validate mediator validates XML messages against a specified schema. You can specify an XPath to extract and validate a specific part of the message. Otherwise, the mediator validates the first child of the SOAP body of the current message.

A [Fault mediator](#) should be added as a child to the Validate mediator in order to specify the fault sequence to be followed if the validation fails.

The Validate mediator is a [content aware mediator](#).

## Syntax

```
<validate [source="xpath"]>
 <property name="validation-feature-id" value="true|false"/>*
 <schema key="string"/>+
 <on-fail>
 mediator+
 </on-fail>
</validate>
```

## UI Configuration

The screenshot shows the WSO2 ESB UI configuration for the Validate Mediator. The interface is divided into several sections:

- Schema keys defined for Validate Mediator:** Contains fields for "Static Key" and "Source", with links to "Configuration Registry" and "Governance Registry". Includes a "Delete" button and an "Add Key" link.
- Features defined for Validator Mediator:** Contains an "Add feature" link.
- Resources of the Validate mediator:** Contains fields for "Location" and "Key", with links to "Configuration Registry" and "Governance Registry". Includes an "Add Resource" link and an "Update" button.

At the bottom, there are buttons for "Save & Close", "Save", and "Cancel".

The mediator configuration can be divided into the following sections.

### Schema keys defined for Validate Mediator

This section is used to specify the key to access the main schema based on which validation is carried out, as well as to specify the XML which needs to be validated. The parameters available in this section are as follows.

Parameter Name	Description

<b>Schema keys defined for Validate Mediator table</b>	<p>The key for the schema location. It can be specified using one of the following methods.</p> <ul style="list-style-type: none"> <li>If the key is a static value, select <b>Static Key</b> from the list and enter a static key in the data field. This value should be pre-defined and saved as a resource in the <a href="#">Registry</a>. Click either <b>Configuration Registry</b> or <b>Governance Registry</b> as relevant to select the required key from the resource tree.</li> <li>If the key is a dynamic value, Select <b>Dynamic Key</b> from the list and enter an expression to calculate the value in the data field.</li> </ul> <p>Click <b>Add Key</b> to add a new schema key. Click <b>Delete</b> in the relevant row to delete a schema key.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p><b>Tip</b></p> <p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p> </div>
<b>Source</b>	<p>The XPath expression to extract the XML that needs to be validated. The Validate mediator validates the evaluation of this expression against the schema specified in the <b>Schema keys defined for Validate Mediator</b> table. If this is not specified, the validation is performed against the first child of the SOAP body of the current message.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p><b>Tip</b></p> <p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p> </div>

## Features Defined for Validator Mediator

This section is used to specify which features of the Validate mediator should be enabled and which should be disabled. The parameters available in this section are as follows.

Only the **FEATURE\_SECURE\_PROCESSING** feature is currently supported by the validator.

Parameter Name	Description
<b>Feature Name</b>	The name of the feature.
<b>Value</b>	Click <b>True</b> to enable the feature, or click <b>False</b> to disable the feature.
<b>Action</b>	Click <b>Delete</b> in the relevant row to delete a feature.

## Resources of the Validate Mediator

A resource in the Validate mediator configuration enables you to import a schema referenced within another schema. In order to access such a schema via a resource, the parent schema should be saved as a resource in the Registry. The parameters available in this section are as follows.

Parameter Name	Description
<b>Location</b>	The location of the schema to be imported. The value entered here should be equal to the value of the schema location attribute within the relevant <code>&lt;xsd:import&gt;</code> element in the parent schema.
<b>Key</b>	The key to access the parent schema saved in the Registry. Click either <b>Configuration Registry</b> or <b>Governance Registry</b> as relevant to select the key from the resource tree.

## Note

You can also configure the Mediator using XML. Click **switch to source view** in the **Mediator** window.

Mediator  switch to source view

## Examples

### Example 1 - Basic configuration

In this example, the required schema for validating messages going through the validate mediator is given as a registry key, schema\sample.xsd. No source attribute is specified, and therefore the schema will be used to validate the first child of the SOAP body. The mediation logic to follow if the validation fails is defined within the `on-fail` element. In this example, the **Fault Mediator** creates a SOAP fault to be sent back to the party which sent the message.

```

<validate>
 <schema key="schema\sample.xsd" />
 <on-fail>
 <makefault>
 <code value="tns:Receiver"
 xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason value="Invalid Request!!!!"/>
 </makefault>
 <property name="RESPONSE" value="true" />
 <header name="To" expression="get-property('ReplyTo')"/>
 </on-fail>
</validate>

```

### Example 2 - Validate mediator with resources

In this example, the following schema named 08MockServiceSchema is saved in the Registry. This schema is located in MockDataTypes.xsd. A reference is made within this schema to another schema named 08SOAPFault.xsd.

ts which is located in `SOAPFaults.xsd`.

```
<xsd:import namespace= "http://samples.synapse.com/08MockServiceSchema"
schemalocation= "MockDataTypes.xsd">
<xsd:import namespace= "http://samples.synapse.com/08SOAPFaults" schemalocation=
"../Common/SOAPFaults.xsd">
</xsd:import>
```

The Validate mediator can be configured as follows.

```
<validate>
<schema key="MockDataTypes.xsd"/>
<resource location="..../Common/SOAPFaults.xsd"
key="conf:custom/schema/SOAPFaults.xsd"/>
<on-fail>
<log level="custom">
<property name="validation failed" value="Validation failed ###" />
<property name="error_msg" expression="$ctx:ERROR_MESSAGE" />
</log>
</on-fail>
</validate>
```

The schema used by the validate mediator is `MockDataTypes.xsd`. In addition, a resource is used to import the 08 `SOPAFaults` schema which is referred in the `08MockServiceSchema` schema. Note that the value `..../Common/SOAPFaults.xsd` which is specified as the location for the schema to be imported is the same as the location specified for 08 `SOPAFaults` schema in the `08MockServiceSchema` configuration.

The `on-fail` sequence of this Validate mediator includes a [Log mediator](#) which is added as a child to the Validate mediator. This log mediator uses two properties to generate the error message `Validation failed ###` when the validation of a message against the schemas specified fails.

The Validate mediator validates JSON messages against a specified JSON schema. You can specify a JSONPath to extract and validate a specific part of the message. Otherwise, the mediator validates the complete content of the current message.

A [Fault mediator](#) or [PayloadFactory mediator](#) should be added as a child to the Validate mediator in order to specify the fault sequence to be followed if the validation fails.

The Validate mediator is a [content aware mediator](#).

## Syntax

```
<validate [source="xpath"]>
 <schema key="string"/>+
 <on-fail>
 mediator+
 </on-fail>
</validate>
```

## UI Configuration

The screenshot shows the WSO2 ESB UI configuration page for the Validate Mediator. The page has a header with 'Mediator' and a 'switch to source view' link, and a 'Help' button. The main content is divided into several sections:

- Schema keys defined for Validate Mediator:** This section contains a table with columns for 'Static Key' (dropdown), 'Data' (text input), 'Configuration Registry' (link), 'Governance Registry' (link), and a 'Delete' button. A 'Add Key' button is also present.
- Features defined for Validator Mediator:** This section contains a 'Add feature' button and a 'Namespace' link.
- Resources of the Validate mediator:** This section contains fields for 'Location' and 'Key', and links to 'Configuration Registry' and 'Governance Registry'. It also includes a 'Add Resource' button, an 'Update' button, and buttons for 'Save & Close', 'Save', and 'Cancel'.

The mediator configuration can be divided into the following sections.

### Schema keys defined for Validate Mediator

This section is used to specify the key to access the main schema based on which validation is carried out, as well as to specify the JSON message which needs to be validated. The parameters available in this section are as follows.

Parameter Name	Description
<b>Schema keys defined for Validate Mediator</b>	<p>The key for the schema location. It can be specified using one of the following methods.</p> <ul style="list-style-type: none"> <li>If the key is a static value, select Static Key from the list and enter a static key in the data field. This value should be pre-defined and saved as a resource in the <a href="#">Registry</a>. Click either Configuration Registry or Governance Registry as relevant to select the required key from the resource tree.</li> <li>If the key is a dynamic value, Select Dynamic Key from the list and enter an expression to calculate the value in the data field.</li> </ul> <p>Click <b>Add Key</b> to add a new schema key. Click <b>Delete</b> in the relevant row to delete a schema key.</p>

**Source**

The JSONPath expression to extract the JSON element that needs to be validated. The Validate mediator validates the evaluation of this expression against the schema specified in the **Schema keys defined for Validate Mediator** table. If this is not specified, the validation is performed against the whole body of the current message.

E.g: json-eval(\$.msg)

**Note**

You can also configure the Mediator using XML. Click **switch to source view** in the **Mediator** window.

Mediator  switch to source view

**Examples**

Following examples use the below sample schema `StockQuoteSchema.json` file. Add this sample schema file (i.e. `StockQuoteSchema.json`) to the following Registry path: `conf:/schema/StockQuoteSchema.json`. For instructions on adding the schema file to the Registry path, see [Adding a Resource](#).

When adding this sample schema file to the Registry, specify the **Media Type** as `application/json`.

```
{
 "$schema": "http://json-schema.org/draft-04/schema#",
 "type": "object",
 "properties": {
 "getQuote": {
 "type": "object",
 "properties": {
 "request": {
 "type": "object",
 "properties": {
 "symbol": {
 "type": "string"
 }
 },
 "required": [
 "symbol"
]
 }
 },
 "required": [
 "request"
]
 }
 },
 "required": [
 "getQuote"
]
}
```

### **Example 1 - Basic configuration**

In this example, the required schema for validating messages going through the Validate mediator is given as a registry key (i.e. schema\StockQuoteSchema.json). You do not have any source attributes specified. Therefore, the schema will be used to validate the complete JSON body. The mediation logic to follow if the validation fails is defined within the on-fail element. In this example, the [PayloadFactory mediator](#) creates a fault to be sent back to the party, which sends the message.

```
<validate>
 <schema key="conf:/schema/StockQuoteSchema.json" />
 <on-fail>
 <payloadFactory media-type="json">
 <format>{ "Error":$1 }</format>
 <args>
 <arg evaluator="xml" expression="$ctx:ERROR_MESSAGE" />
 </args>
 </payloadFactory>
 <property name="HTTP_SC" value="500" scope="axis2" />
 <respond/>
 </on-fail>
</validate>
```

An example for a valid JSON payload request is given below.

```
{
 "getQuote": {
 "request": {
 "symbol": "WSO2"
 }
 }
}
```

### **Example 2 - Validate mediator with source (JSONPath)**

In this example, it extracts the message element from the JSON request body and validates only that part of the message against the given schema.

```
<validate source="json-eval($.msg)">
 <schema key="conf:/schema/StockQuoteSchema.json" />
 <on-fail>
 <payloadFactory media-type="json">
 <format>{ "Error":$1 }</format>
 <args>
 <arg evaluator="xml" expression="$ctx:ERROR_MESSAGE" />
 </args>
 </payloadFactory>
 <property name="HTTP_SC" value="500" scope="axis2" />
 <respond/>
 </on-fail>
</validate>
```

An example for a valid JSON request payload is given below.

```
{
 "msg": {
 "getQuote": {
 "request": {
 "symbol": "WSO2"
 }
 }
 }
}
```

## XQuery Mediator

The **XQuery Mediator** performs an XQuery transformation on messages.

The XQuery mediator is a [content aware mediator](#).

---

[Syntax](#) | [UI Configuration](#) | [Examples](#)

---

### Syntax

```
<xquery key="string" [target="xpath"]>
 <variable name="string" type="string" [key="string"] [expression="xpath"]
 [value="string"]/>?
</xquery>
```

---

### UI Configuration

The screenshot shows the XQuery Mediator configuration interface. It includes fields for 'Key Type' (set to 'Static Key'), 'Key' (empty), 'Target' (empty), and links to 'Configuration Registry', 'Governance Registry', and 'NameSpaces'. Below these are sections for 'Variables of the XQuery mediator' with an 'Add Variable' button and an 'Update' button. At the bottom are standard save and cancel buttons.

The parameters available to configure the XQuery mediator are as follows.

Parameter Name	Description
<b>Key Type</b>	This parameter specifies whether the key which represents the XQuery transformation should be a static value or a dynamic value. <ul style="list-style-type: none"> <li>• <b>Static:</b> If this is selected, the key would be a static value. This value should be selected from the <a href="#">Key</a> parameter.</li> <li>• <b>Dynamic:</b> If this is selected, the key would be a dynamic value which has to be evaluated via an XPath expression.</li> </ul>
<b>Key</b>	The key that represents the XQuery transformation. The value you enter depends on the value you select for the <b>Key Type</b> parameter, click <b>Configuration Registry</b> or <b>Governance Registry</b> as relevant to select the key. If you select the <b>Dynamic</b> option for the <b>Key Type</b> parameter, enter the XPatch expression which calculates the dynamic key.
	<p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the NameSpaces dialog will appear to define the number of namespace prefixes and URLs used in the XPath expression.</p>
<b>Target</b>	This parameter specifies the node of the message to which the XQuery transformation should be applied. If no target is specified for this parameter, the XQuery transformation is applied to the first child of the SOAP body.
	<p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the NameSpaces dialog will appear to define the number of namespace prefixes and URLs used in the XPath expression.</p>
<b>Add Variable</b>	This link allows you to define one or more variables that could be bound to the dynamic context of the invocation. Click <b>Add Variable</b> to add a variable to the XQuery mediator configuration. The page will expand to display the variable configuration. The behavior of the configuration page would differ depending on whether you select <b>Value</b> or <b>Expression</b> as the variable value type. Click on the variable to edit it.

## Value Expression

**Variables of the XQuery mediator**

Variable Type	Variable Name	Value Type	Value / Expression	Action
<input type="button" value="Select A Value"/>	<input type="text"/>	<input type="button" value="Value"/>	<input type="text"/>	Delete
<a href="#"> Add Variable</a>				
<input type="button" value="Update"/>				

Parameter Name	Description
<b>Variable Type</b>	<p>The data type of the variable. Supported values are as follows.</p> <ul style="list-style-type: none"> <li>• <b>INT</b></li> <li>• <b>INTEGER</b></li> <li>• <b>BOOLEAN</b></li> <li>• <b>BYTE</b></li> <li>• <b>DOUBLE</b></li> <li>• <b>SHORT</b></li> <li>• <b>LONG</b></li> <li>• <b>FLOAT</b></li> <li>• <b>STRING</b></li> <li>• <b>DOCUMENT</b></li> <li>• <b>ELEMENT</b></li> </ul>
<b>Variable Name</b>	The name of the variable. It should correspond to the name of the variable declared in the XQuery script.
<b>Value Type</b>	<p>This parameter specifies whether the variable value should be a static value or a dynamic value.</p> <ul style="list-style-type: none"> <li>• <b>Value:</b> If this is selected, the variable value is a static value. The static value is specified in the <b>Value/Expression</b> parameter.</li> <li>• <b>Expression:</b> If this is selected the variable value is a dynamic value. The XQuery expression is specified in the <b>Value/Expression</b> parameter.</li> </ul>
<b>Value/Expression</b>	This parameter is used to enter the variable value. This can be a static value or a dynamic value. The <b>Type</b> parameter is used to specify the data type of the variable.
<b>Action</b>	This parameter allows the variable to be deleted.

## Variables of the XQuery mediator

Variable Type	Variable Name	Value Type	Value / Expression	Registry Key
Select A Value		Expression		

 Add Variable

Parameter Name	Description
<b>Variable Type</b>	The data type of the variable. This should be a valid type defined by the JSR <ul style="list-style-type: none"> <li>• <b>INT</b></li> <li>• <b>INTEGER</b></li> <li>• <b>BOOLEAN</b></li> <li>• <b>BYTE</b></li> <li>• <b>DOUBLE</b></li> <li>• <b>SHORT</b></li> <li>• <b>LONG</b></li> <li>• <b>FLOAT</b></li> <li>• <b>STRING</b></li> <li>• <b>DOCUMENT</b></li> <li>• <b>DOCUMENT_ELEMENT</b></li> <li>• <b>ELEMENT</b></li> </ul>
<b>Variable Name</b>	The name of the variable. It should correspond to the name of the variable declared in the XQuery expression.
<b>Value Type</b>	This parameter specifies whether the variable value should be a static value or a dynamic value. <ul style="list-style-type: none"> <li>• <b>Value:</b> If this is selected, the variable value is a static value. The static value can be provided in the <b>Value/Expression</b> parameter.</li> <li>• <b>Expression:</b> If this is selected the variable value is a dynamic value. The XQuery expression is provided in the <b>Value/Expression</b> parameter.</li> </ul>
<b>Value/Expression</b>	This parameter is used to enter the variable value. This can be a static value or a dynamic value. The dynamic value can be provided in the <b>Type</b> parameter.
<b>Registry Key</b>	The key to access the variable value if it is saved in the <a href="#">Registry</a> . Click either <b>Configuration Registry</b> or <b>Registry Browser</b> parameter as relevant to select the required key from the resource tree.
<b>Registry Browser</b>	If the variable value is saved in the <a href="#">Registry</a> , click either <b>Configuration Registry</b> or <b>Registry Browser</b> parameter as relevant to select the required key from the resource tree.
<b>NS Editor</b>	You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. You can provide any number of namespace prefixes and URLs used in the XPath expression.
<b>Action</b>	This parameter allows the variable to be deleted.

## Note

You can configure the mediator using XML. Click **switch to source view** in the **Mediator** window.

Mediator  switch to source view

### Examples

In this configuration, the XQuery script is saved in the registry, and it can be accessed via the `xquery\example.xq` key. The XQuery configuration has one variable named `payload` of which the variable type is `ELEMENT`. As there is no expression in the variable definitions, the default value of the first child of the SOAP Body is used as the value of the variable `payload`. Within the XQuery script, you can access this variable by defining `declare variable $payload as document-node() external;`.

```
<xquery key="xquery\example.xq">
 <variable name="payload" type="ELEMENT" />
</xquery>
```

### Variables

The following variable picks an XML resource from the registry using key `misc/commission.xml` and binds into XQuery Runtime so that it can be accessed within the XQuery script.

```
<variable name="commission" type="ELEMENT" key="misc/commission.xml"></variable>
```

The value of the following variable is calculated from the current message SOAP Payload using an expression. The value type of the variable is `DOUBLE`.

```
<variable name="price" type="DOUBLE"
expression="self::node()//m0:return/m0:last/child::text()"
xmlns:m0="http://services.samples/xsd"/>
```

### XSLT Mediator

The **XSLT Mediator** applies a specified XSLT transformation to a selected element of the current message payload. In addition, you can:

- Specify properties already included in the mediation flow to be added to the XSLT script as XSLT parameters.
- Specify features to be enabled/disabled in the XSLT transformation.
- Import external XSLT scripts to the main XSLT script of the XSLT mediator by adding them as resources.

The XSLT mediator is a **content aware mediator**.

[Syntax](#) | [UI Configuration](#) | [Examples](#) | [Samples](#)

### Syntax

```
<xslt key="string" [source="xpath"]>
 <property name="string" (value="literal" | expression="xpath") />*
 <feature name="string" value="true| false" />*
 <resource location="string" key="string"/>*
</xslt>
```

## UI Configuration

The screenshot shows the XSLT Mediator configuration page. At the top, there are tabs for 'Mediator' and 'switch to source view'. Below the tabs, the title 'XSLT Mediator' is displayed along with a 'Help' link. The main configuration area starts with a 'Key Type' section where 'Static Key' is selected. There are input fields for 'Key' and 'Source', and links to 'Configuration Registry', 'Governance Registry', and 'NameSpaces'. Below this, there are three sections: 'Properties of the XSLT mediator', 'Features of the XSLT mediator', and 'Resources of the XSLT mediator', each with an 'Add' button.

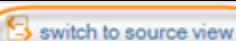
The parameters available for configuring the XSLT mediator are as follows.

Parameter Name	Description
<b>Key Type</b>	You can select one of the following options. <ul style="list-style-type: none"> <li><b>Static Key:</b> If this is selected, an existing key can be selected from the registry for the <b>Key</b> parameter.</li> <li><b>Dynamic Key:</b> If this is selected, the key can be entered dynamically in the <b>Key</b> parameter.</li> </ul>
<b>Key</b>	This specifies the registry key to refer the XSLT to. This supports static and dynamic keys.

<b>Source</b>	<p>This determines the element to which the given XSLT transformation should be applied via an XPath expression. If the source element is not specified, the XSLT transformation is applied to the first child of the SOAP body.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p><b>Tip</b></p> <p>You can click <b>NameSpaces</b> to add namespaces if you are providing an expression. Then the <b>Namespace Editor</b> panel would appear where you can provide any number of namespace prefixes and URLs used in the XPath expression.</p> </div>
<b>Properties of the XSLT mediator</b>	<p>This section is used to inject properties set in the mediation flow to the XSLT script as XSLT parameters. These are referred from the XSLT in transformation using the <code>get-property(prop-name)</code> XPath extension function.</p> <p>Parameters relating to the properties are as follows.</p> <ul style="list-style-type: none"> <li>• <b>Property Name:</b> The name of the property to be passed into the transformations.</li> <li>• <b>Property Type:</b> This specifies whether the property is given as a static value or an XPath expression.</li> <li>• <b>Value/Expression</b> - This defines the static value or the XPath expression.</li> <li>• <b>Action</b> - This parameter allows the property to be removed from the XSLT script if required.</li> </ul>
<b>Features of the XSLT mediator</b>	<p>This section is used to specify features to be enabled/disabled in the XSLT transformation. For example, adding the <code>http://ws.apache.org/ns/synapse/transform/feature/dom</code> feature turns on DOM-based transformations instead of serializing elements into byte streams and/or temporary files. This approach can improve performance but might not work for all transformations.</p> <p>Parameters relating to the features are as follows.</p> <ul style="list-style-type: none"> <li>• <b>Feature Name:</b> The name of the feature to be enabled/disabled in the XSLT transformation.</li> <li>• <b>Feature Value:</b> This specified whether the feature is enabled or not. Select <b>True</b> to enable the feature or <b>False</b> to disable it.</li> <li>• <b>Action:</b> This allows you to remove the feature from the XSLT transformation if required.</li> </ul>
<b>Resources of the XSLT mediator</b>	<p>This section is used to import external XSLT scripts to the main XSLT scripts defined in the XSLT mediator. The XSLT scripts to be imported are first added as resources in the registry.</p> <p>Parameters relating to the resources are as follows.</p> <ul style="list-style-type: none"> <li>• <b>Location:</b> The location where the XSLT script to be imported is saved as a resource.</li> <li>• <b>Key:</b> The registry key to which the XSLT should be referred. Browse for the relevant key in the Configuration registry or the Governance registry.</li> <li>• <b>Action:</b> This allows you to remove the imported XSLT script added as a resource if required.</li> </ul>

## Note

You can also configure the Mediator using XML. Click **switch to source view** in the **Mediator** window.

Mediator  switch to source view

## Examples

- Example 1 - Applying a XSLT transformation to a element selected based on an XPath expression
- Example 2 - Adding properties as XSLT parameters

- Example 3 - Adding XSLT imports as resources

#### ***Example 1 - Applying a XSLT transformation to a element selected based on an XPath expression***

In this example, the XSLT can be picked by the key `transform/example.xslt` and the XSLT would be applied to a part of the message that is specified as an XPath expression. In this case, it is applied to `s11:Body/child` the message.

```
<xslt xmlns="http://ws.apache.org/ns/synapse" key="transform/example.xslt"
source="s11:Body/child" />
```

#### ***Example 2 - Adding properties as XSLT parameters***

In this example, a property named `PARAM_NAME` is added to the XSLT script as an XSLT parameter. A XPath expression is used to assign this property the value of the `ORDER_ID` property in the default scope.

```
<xslt key="keyToXSLTFile">
 <property expression="$ctx:ORDER_ID" name="PARAM_NAME" >
 </property></xslt>
```

The XSLT script with the `PARAM_NAME` property added would look as follows.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:param name="PARAM_NAME"></xsl:param>
 <xsl:template match="/">
 <orders xmlns="http://services.samples">
 <xsl:attribute name="id">
 <xsl:value-of select="$PARAM_NAME" >
 </xsl:value-of></xsl:attribute>
 </orders>
 </xsl:template>
</xsl:stylesheet>
```

#### ***Example 3 - Adding XSLT imports as resources***

In this example, two XSLT files saved in the registry under `conf:/` as resources are imported to the main XSLT script of the XSLT mediator.

`xslt1.xslt`

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:template match="//people/person" name="FILL_PPL">
 <client>
 <firstname>
 <xsl:value-of select="firstname">
 </xsl:value-of></firstname>
 <lastname>
 <xsl:value-of select="lastname">
 </xsl:value-of></lastname>
 <age>
 <xsl:value-of select="age">
 </xsl:value-of></age>
 <country>
 <xsl:value-of select="country">
 </xsl:value-of></country>
 </client>
 </xsl:template>
</xsl:stylesheet>

```

### xslt2.xslt

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:include href="xslt1.xslt">
 <xsl:template match="/">
 <clients>
 <xsl:for-each select="//people/person">
 <xsl:call-template name="FILL_PPL">
 </xsl:call-template></xsl:for-each>
 </clients>
 </xsl:template>
 </xsl:include></xsl:stylesheet>

```

<xsl:include href="xslt1.xslt"> element indicates that the xslt1.xslt is included in xslt2.xslt.

These two files can be imported to the script of the XSLT mediator as follows.

```

<xslt key="conf:/xslt2.xslt">
 <resource key="conf:/xslt1.xslt" location="xslt1.xslt">
</resource></xslt>

```

The following SOAP request can be used to test the above configuration of the XSLT mediator included in a proxy configuration.

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
 <soapenv:Header>
 <soapenv:Body>
 <people>
 <person>
 <firstname>Isuru</firstname>
 <lastname>Udana</lastname>
 <gender>Male</gender>
 <age>26</age>
 <country>SriLanka</country>
 </person>
 <person>
 <firstname>Ishan</firstname>
 <lastname>Jayawardena</lastname>
 <gender>Male</gender>
 <age>26</age>
 <country>SriLanka</country>
 </person>
 </people>
 </soapenv:Body>
</soapenv:Envelope>

```

## Samples

[Sample 440: Converting JSON to XML Using XSLT](#)

[Sample 8: Introduction to Static and Dynamic Registry Resources and Using XSLT Transformations](#)

### Working with Mediators via WSO2 ESB Tooling

If you need to create a custom mediator that performs some logic on a message, you can either create a new mediator project, or import an existing mediator project using WSO2 ESB tooling.

You need to have WSO2 ESB tooling installed to create a new message store or to import an existing message store via ESB tooling. For instructions on installing WSO2 ESB tooling, see [Installing WSO2 ESB Tooling](#).

Once a mediator project is finalised, you can export it as a deployable artifact by right-clicking on the project and selecting **Export Project as Deployable Archive**. This creates a JAR file that you can deploy to the ESB.

Alternatively, you can group the mediator project as a Composite Application Project, create a Composite Application Archive (CAR), and deploy it to the ESB.

A URL classloader is used to load classes in the mediator (class mediators are not deployed as OSGi bundles). Therefore, it is only possible to refer to the class mediator from artifacts packed in the same CAR file in which the class mediator is packed. Accessing the class mediator from an artifact packed in another CAR file is not possible. However, it is possible to refer to the class mediator from a sequence packed in the same CAR file and call that sequence from any other artifact packed in other CAR files.

### Creating a mediator project

Follow these steps to create a new mediator. Alternatively, you can [import a mediator project](#).

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Mediator Project** on the **Developer Studio Dashboard**.

3. Leave the first option selected and click **Next**. The New Mediator Creation Wizard appears.
  
  
  
  
  
  
4. Do the following:
  - a. Type a unique name for the project.
  - b. Specify the package and class names you are creating.
  - c. Optionally specify the location where you want to save the project (or leave the default location specified).
  - d. Optionally specify the working set, if any, that you want to include in this project.
5. A Maven POM file will be generated automatically for this project. If you want to include parent POM information in the file from another project in this workspace, click **Next**, click the **Specify Parent from Workspace** check box, and then select the parent project.
6. Click **Finish**.

The mediator project is created in the workspace location you specified with a new mediator class that extends `org.apache.synapse.mediators.AbstractMediator`.

### Importing a mediator project

Follow these steps to import an existing mediator project into an ESB Config project. Alternatively, you can [create a new mediator project](#).

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Mediator Project** on the **Developer Studio Dashboard**.
3. Select **Import From Workspace** and click **Next**.
4. Specify the mediator project in this workspace that you want to import. Only projects with source files that extend `org.apache.synapse.mediators.AbstractMediator` are listed. Optionally, you can change the location where the mediator project will be created and add it to working sets.
5. Click **Finish**.

The mediator project you selected is created in the location you specified, and the project now has the added project nature `org.wso2.developerstudio.eclipse.artifact.mediator.project.nature`.

### Working with Mediators via the Management Console

You can add a mediator to a sequence, edit a mediator as well as delete a mediator via the ESB Management Console. You can also add child nodes to some of the mediators if necessary.

See the following topics for information on how to work with mediators via the Management Console:

- [Adding a Mediator to a Sequence](#)
- [Adding a Child Mediator](#)
- [Editing a Mediator](#)
- [Deleting a Mediator](#)

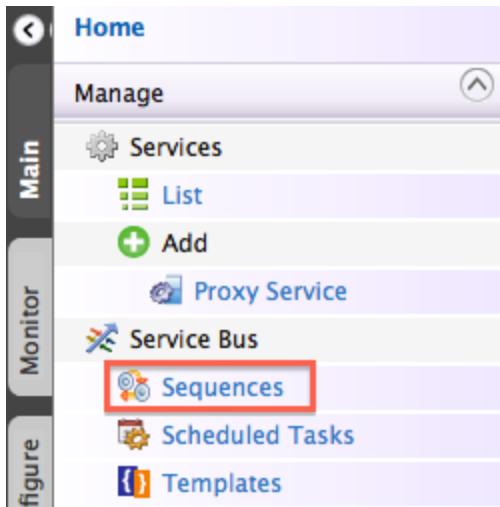
### Adding a Mediator to a Sequence

To make the process of [message mediation](#) feasible, [create a sequence](#) and add [mediators](#) to it.

Follow the instructions below to add a new mediator to a sequence in WSO2 ESB.

When adding mediators to a sequence, if you get the error "Error in loading the mediator design view", your session has expired and you must log in again. To prevent losing any work if your session times out, be sure to save your sequence periodically.

1. On the Main tab in the ESB Management Console, navigate to **Manage -> Service Bus** and click **Sequence**.



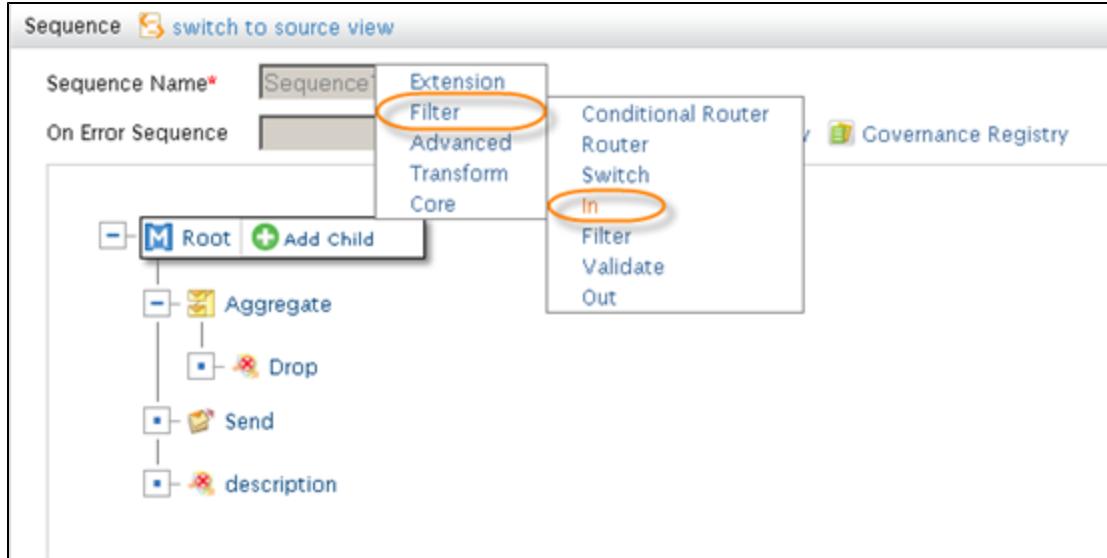
2. Select the tab where your sequence is located:
  - **Defined Sequences** - Shows sequences saved in the Synapse configuration.
  - **Dynamic Sequences** - Shows sequences saved in the [registry](#).
3. Click **Edit** for the sequence where you're adding the mediator. For example, the following illustration shows how to edit the test sequence.

**Mediation Sequences**

Select	Sequence Name	Actions
<input type="checkbox"/>	fault	<a href="#">Enable Statistics</a> <a href="#">Enable Tracing</a> <a href="#">Edit</a> <a href="#">Delete</a>
<input type="checkbox"/>	main	<a href="#">Enable Statistics</a> <a href="#">Enable Tracing</a> <a href="#">Edit</a> <a href="#">Delete</a>
<input type="checkbox"/>	test	<a href="#">Enable Statistics</a> <a href="#">Enable Tracing</a> <a href="#">Edit</a> <a href="#">Delete</a>

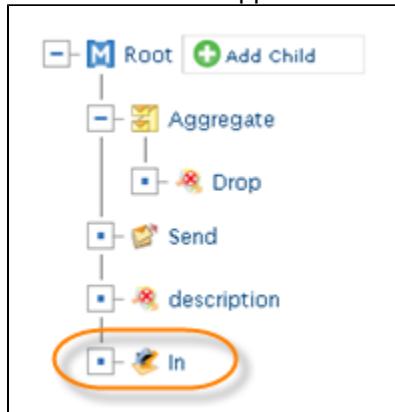
Select all in this page | Select none      [Delete](#)

4. In the sequence tree, do one of the following:
  - To add the mediator at the root level, click **Add Child** next to Root.
  - To add the mediator under or after an existing node in the tree, click that node, and click **Add Child** or **Add Sibling**.
5. Select the type of mediator you want to add. For example, to create an In mediator, click **Filter -> In**.



- In the configuration pane that appears below the sequence tree, set the properties for the mediator, and then click **Update**. You can define a mediator's properties using the controls in the user interface (design view) or by configuring the XML (source view). For detailed information about mediator properties, see [Mediators](#) and click the name of the mediator you are adding.

The new mediator appears in the sequence tree.



- Save the sequence.

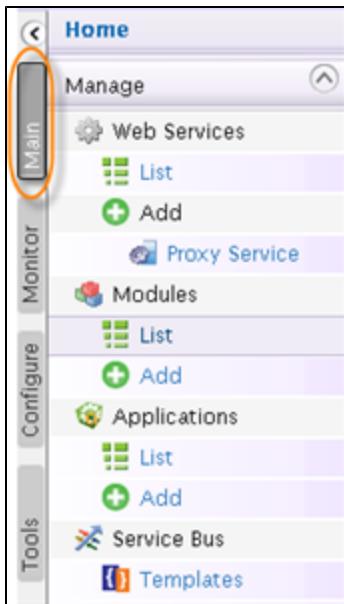
### Adding a Child Mediator

Some mediators can have child mediators (also called "child nodes"). You can add child nodes to the following mediators:

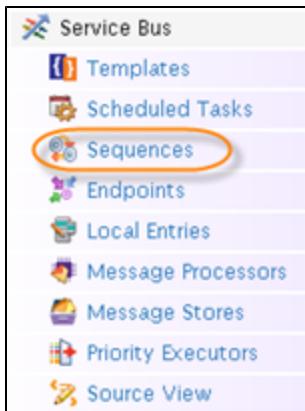
- Cache Mediator
- In and Out Mediators
- Iterate Mediator
- Rule Mediator
- Validate Mediator

Follow the instructions below to add a child node to a mediator in WSO2 ESB.

- Sign In. Enter your user name and password to log on to the ESB Management Console.
- Click on "Main" in the left menu to access the "Manage" menu.



3. In the "Manage" menu, click on the "Sequences" under the "Service Bus."



4. Select the necessary sequence and click on the "Edit" link to open the "Edit Sequence" window.

The screenshot shows the 'Mediation Sequences' page. At the top, there is a breadcrumb trail: Home > Manage > Service Bus > Sequences. On the right, there is a 'Help' link. Below the breadcrumb, there is a heading 'Mediation Sequences'. A button 'Add Sequence' is visible. Below it, there are two tabs: 'Defined Sequences' (selected) and 'Dynamic Sequences'. A message 'Available defined Sequences in the Synapse Configuration' is displayed. A table lists sequences with columns 'Sequence Name' and 'Actions'. The 'fault' sequence has actions 'Enable Statistics', 'Enable Tracing', and an 'Edit' link (circled in orange). The 'main' sequence also has these actions. There are 'Edit' and 'Delete' links for each row.

Sequence Name	Actions
fault	<input type="checkbox"/> Enable Statistics <input type="checkbox"/> Enable Tracing <a href="#">Edit</a>
main	<input type="checkbox"/> Enable Statistics <input type="checkbox"/> Enable Tracing <a href="#">Edit</a>

5. In the "Sequence" pane, click on the mediator to add a child.

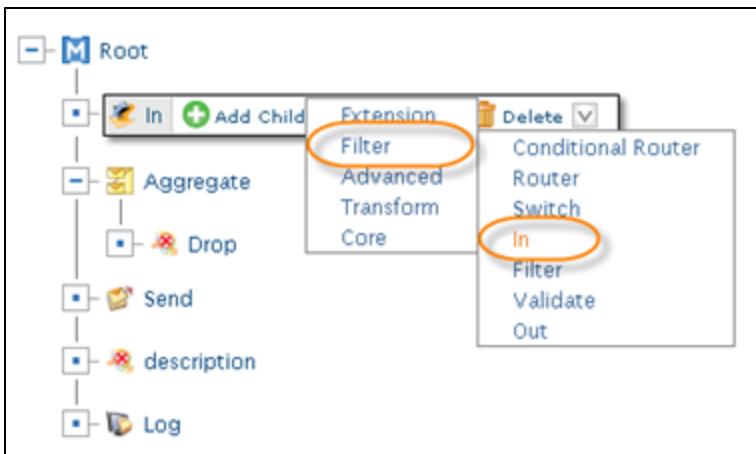


6. Click on the "Add Child" button.



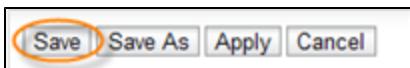
7. Choose the necessary category and mediator from the drop-down menu.

In the example below, the "In" mediator from the "Filter" category was chosen.

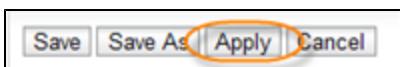


8. Set all the properties of the child mediator. The mediator properties window appears under the sequence tree. You can define a mediator's properties using UI or in XML mode. For detailed information about mediator properties, see [Mediators](#).

9. Click "Save" or "Apply" to add a child node to a mediator.



OR



### Editing a Mediator

Different mediators have their own XML configurations. You can edit a mediator using XML or in UI.

Follow the instructions below to edit a mediator in WSO2 ESB.

1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Click on "Main" in the left menu to access the "Manage" menu.



3. In the "Manage" menu, click "Sequences" under "Service Bus."



4. Select the tab where your sequence is located:

- **Defined Sequences** - Shows sequences saved in Synapse configuration.
- **Dynamic Sequences** - Shows sequences saved in the [Registry](#).

5. Select the necessary sequence and click on the "Edit" link to open the "Edit Sequence" window.

Home > Manage > Service Bus > Sequences

## Mediation Sequences

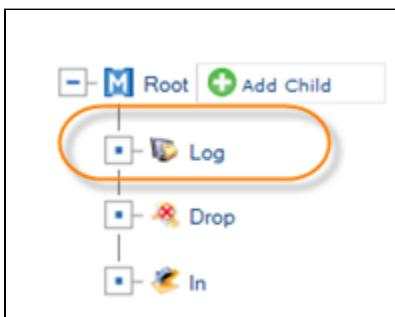
Add Sequence

Defined Sequences    Dynamic Sequences

Available defined Sequences in the Synapse Configuration

Sequence Name	Actions
fault	<input type="checkbox"/> Enable Statistics <input type="checkbox"/> Enable Tracing     Edit     Delete
main	<input type="checkbox"/> Enable Statistics <input type="checkbox"/> Enable Tracing     Edit     Delete

6. Click on the necessary mediator in the "Sequence" tree. In the following example, we select the "Log" mediator.



7. The mediator properties window appears under the sequence tree.

Mediator Help

### Log Mediator

Log Category: INFO

Log level: Full

Log Separator: ,

**Properties of the Log mediator**

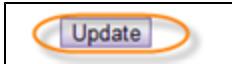
Property Name	Property Value	Value/Expression	Name Space Editor	Action
MESSAGE	Value	Executing default 'fault' se		
ERROR_CODE	Expression	get-property('ERROR_CO		
ERROR_MESSAGE	Expression	get-property('ERROR_ME		

Update

8. Edit the mediator properties.

For detailed information about specific mediator properties, see [Mediators](#).

9. Click on the "Update" button to save the new settings of the mediator.

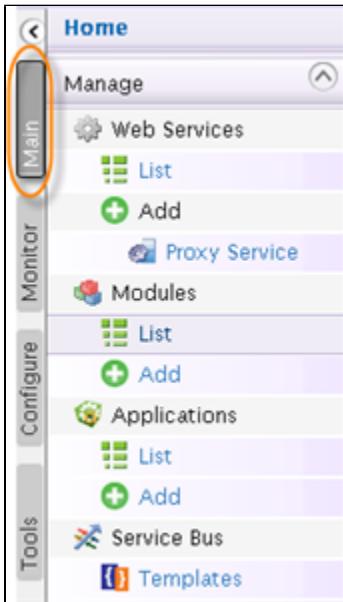


## Deleting a Mediator

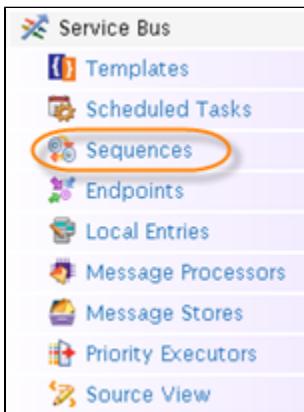
If there is no need to use a particular mediator for the message mediation, you can delete it from the sequence.

Follow the instructions below to delete a mediator from a sequence in WSO2 ESB.

1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Click on "Main" in the left menu to access the "Manage" menu.



3. In the "Manage" menu, click on "Sequences" under "Service Bus."



4. Select the tab where your sequence is located:

- **Defined Sequences** - Shows sequences saved in Synapse configuration.
- **Dynamic Sequences** - Shows sequences saved in the Registry.

5. Select the necessary sequence and click on the "Edit" link to open the "Edit Sequence" window.

Home > Manage > Service Bus > Sequences

**Mediation Sequences**

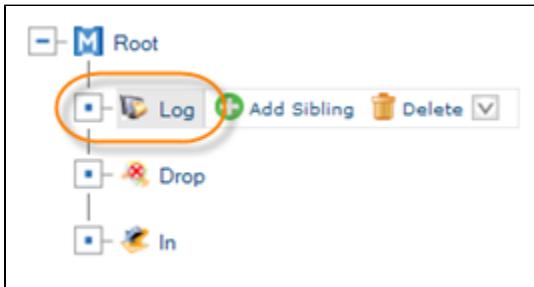
Add Sequence

Defined Sequences    Dynamic Sequences

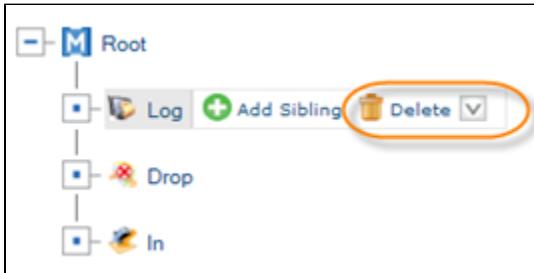
Available defined Sequences in the Synapse Configuration

Sequence Name	Actions
fault	<input type="checkbox"/> Enable Statistics <input type="checkbox"/> Enable Tracing     Edit     Delete
main	<input type="checkbox"/> Enable Statistics <input type="checkbox"/> Enable Tracing     Edit     Delete

6. Click on the necessary mediator in the "Sequence" Tree and the "Delete" button appears.



7. Click on the "Delete" button.



8. Confirm your request to delete the mediator by clicking on the "Yes" button in the "WSO2 Carbon" dialog window.



## Working with Local Registry Entries

The **local registry** acts as a memory registry where you can store static content as a key-value pair, where the value could be a static entry such as a text string, XML code, or a URL. This is useful for the type of static content often found in XSLT files, WSDL files, URLs, etc. Local entries can be referenced from mediators in the ESB mediation flows and resolved at runtime.

When you want to work with local registry entries, you can use the ESB tooling plug-in to create a new local entry as well as to import an existing local entry, or you can add, edit, and delete local registry entries via the Management Console.

The <localEntry> element is used to declare registry entries that are local to the ESB instance as shown below:

```
<localEntry key="string" src="url">text | xml</localEntry>
```

These entries are top-level entries and are globally visible within the entire system. Values of these entries can be retrieved via the extension XPath function `synapse:get-property(prop-name)`, and the keys of these entries could be specified wherever a registry key is expected within the configuration.

An entry can be static text specified as inline text or static XML specified as an inline XML fragment, or it can be specified as a URL (using the `src` attribute). A local entry shadows any entry with the same name from a remote Registry.

```
<localEntry key="version">0.1</localEntry>
<localEntry key="validate_schema">
 <xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema" ...>
 </xss:schema>
 </localEntry>
<localEntry key="xslt-key-req"
src="file:repository/samples/resources/transform/transform.xslt"/>
```

If you want to add local entries before deploying the server, you can add them to the top-level bootstrap file `synapse.xml`, or to separate XML files in the `local-entries` directory, which are located in under `<ESB_HOME>\repository\deployment\server\synapse-configs\default`. When the server is started, these configurations will be added to the registry.

#### **Working with local entries via WSO2 ESB Tooling**

You can create a new local entry or import an existing local entry from an XML file, such as a Synapse configuration file using WSO2 ESB tooling.

You need to have WSO2 ESB tooling installed to work with local entries via ESB tooling. For instructions on installing WSO2 ESB tooling, see [Installing WSO2 ESB Tooling](#).

#### **Creating a new local entry**

Follow these steps to create a new local entry. Alternatively, you can [import an existing local entry](#).

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Local Entry** on the **Developer Studio Dashboard**.
3. Select **Create a New Local Entry** and click **Next**.
4. Type a unique name for the local entry, specify one of the following types of local entries, and then fill in the advanced configuration as described below:
  - In-Line Text Entry: Type the text you want to store
  - In-Line XML Entry: Type the XML code you want to store
  - Source URL Entry: Type or browse to the URL you want to store
5. Do one of the following:
  - To save the local entry in an existing ESB Config project in your workspace, click **Browse** and select that project.
  - To save the local entry in a new ESB Config project, click **Create new Project** and create the new project.
6. Click **Finish**. The local entry is created in the local-entries folder under the ESB Config project you specified, and the local entry appears in the editor. Click its icon in the editor to view its properties.

#### **Importing a local entry**

Follow these steps to import an existing local entry from an XML file (such as a Synapse configuration file) into an ESB Config project. Alternatively, you can [create a new local entry](#).

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Local Entry** on the **Developer Studio Dashboard**.
3. Select **Import Local Entry** and click **Next**.
4. Specify the local entry file by typing its full path name or clicking **Browse** and navigating to the file.
5. In the **Save Local Entry In** field, specify an existing ESB Config project in your workspace where you want to save the local entry, or click **Create new Project** to create a new ESB Config project and save the local entry there.
6. In the **Advanced Configuration** section, select the local entries you want to import.
7. Click **Finish**. The local entries you selected are created in the `local-entries` folder under the ESB Config project you specified, and the first local entry appears in the editor.

## Using a local entry

After you create a local entry, you can reference it from a mediator in your mediation workflow. For example, if you created a local entry with XSLT code, you can add an XSLT mediator to the workflow and then reference the local entry as follows:

1. Click the XSLT mediator to view its properties, click the **XSLT Static Schema Key** property, and then click the browse [...] button on the far right of the property's value.
2. Click the **Workspace** link, and then navigate to and select the local entry that contains the XSLT code.
3. Click **OK**.

## Creating Custom Mediators

WSO2 ESB comes with an assortment of [mediators](#) to filter, transform, route and manipulate messages. When you have a scenario that requires functionality not provided by the existing mediators, you can write your own custom mediators to implement your specific business requirements. Your custom mediators then must be plugged into the WSO2 ESB. After adding them to the ESB, they function together with core mediators that come with the product. The custom mediators can be distributed in a packaged form that can be installed in another WSO2 ESB instance.

For more information, see [Writing a WSO2 ESB Mediator](#).

## Best Practices for Mediation

This section explains the best practices to be followed when creating mediation sequences in the following scenarios.

- General best practices
- Last mediator in a sequence
- Defining the In and Out sequences
- The correct usage of the Loopback mediator
- Reusing a defined sequence
- The correct usage of the Sequence mediator

### ***General best practices***

Following are some general best practices that you can follow when working with mediators.

- Use Iterator mediator in association with Aggregate mediator.
- Do not do any configuration after the Send mediator.
- Do proper error handling to handle mediation errors as well as endpoint errors.
- Use dollar context (i.e. `$ctx`) instead of using the `get` property.

This is because the `get-property` methods search even in Registry if the value is not available in the message context. Thus, it affects performance as Registry search is an expensive operation. However, `$ctx` only checks in the message context.

- Use appropriate intervals for tasks.

#### **Last mediator in a sequence**

The last mediator in a mediation sequence should be one of the following mediators depending on the scenario. Any mediator added after one of the following mediators will not be applied.

- **Drop mediator:** when you want to stop the mediation flow at a particular point (e.g., when a filter condition is not met).
- **Loopback mediator:** when you want the message to be moved from the In sequence to the Out sequence. Note that the Loopback mediator only prevents the subsequent mediators in the In sequence from being applied.
- **Respond mediator:** when you want the message to be sent back to the client.
- **Send Mediator:** when you want the message to be sent to the specified endpoint. If a message should be further mediated after it is sent, you can use the [Clone mediator](#) to make two copies of the message and process them separately, thereby avoiding conflicts.

#### **Defining the In and Out sequences**

The In and Out sequences are separately defined for [proxy services](#) and [REST APIs](#). Therefore, [In and Out mediators](#) should not be used in proxy service and REST API configurations. However, they should be used in the main sequence.

#### **The correct usage of the Loopback mediator**

Once a message has been passed from the In sequence (request path) to the Out sequence (response path), it cannot be moved to the Out sequence again via the [Loopback mediator](#).

#### **Reusing a defined sequence**

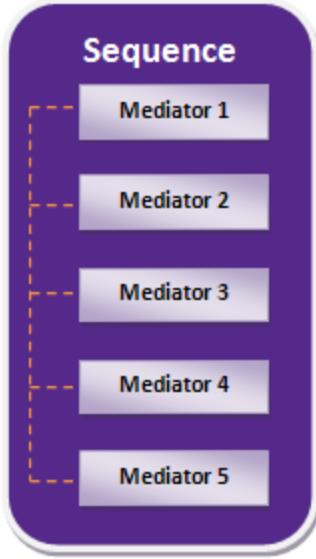
In order to repeatedly use the same mediation sequence, you can define it and save it in the Synapse configuration or the [Registry](#) with a unique name as described in [Adding a Mediation Sequence](#). This mediation sequence can then be called from the main sequence as well as multiple [proxy services](#) and [REST APIs](#). The saved sequence can be called via the [Sequence mediator](#) or selected as the In/Out/Fault sequence when defining a [proxy service](#) or a [REST API](#).

#### **The correct usage of the Sequence mediator**

The [Sequence mediator](#) calls a [Mediation Sequence](#) saved in the Synapse configuration or the [Registry](#) with a unique name. The In, Out or Fault sequence of a [proxy service](#), [REST API](#) or the Main sequence cannot be called via this mediator.

### **Mediation Sequences**

A **mediation sequence**, commonly called a **sequence**, is a tree of [mediators](#) that you can use in your mediation workflow. When a message is delivered to a sequence, the sequence sends it through all its mediators.



When you want to work with mediation sequences, you can use the ESB tooling plug-in to create a new sequence as well as to import an existing sequence, or you can add, edit, and delete sequences via the Management Console.

For information on how to use the ESB tooling plug-in to create a new sequence or to import an existing sequence, see [Working with Sequences via WSO2 ESB Tooling](#).

For information on working with local entries via the management console, see [Working with Sequences via the Management Console](#).

#### **Configuring a mediation sequence**

You can define mediation sequences using the Management Console as described in [Adding a Mediation Sequence](#). The underlying Synapse configuration uses the following syntax:

```
<sequence name="string" [onError="string"] [key="string"] [trace="enable"]
[statistics="enable"]>
 mediator*
</sequence>
```

You can list the mediators right in the sequence definition (referred to as an **in-line sequence**) and refer to other sequences by name. For example:

```
<sequence name="foo">
 <log/>
 <property name="test" value="test value"/>
 <sequence key="other_sequence" />
 <send/>
</sequence>
```

This sequence specifies three mediators in-line: the [log mediator](#), [property mediator](#), and the [send mediator](#). It also references the named sequence "other\_sequence" and therefore uses all the mediators defined in that sequence.

In addition to mediators and other sequences, you can configure the following:

- Create a dynamic sequence by referring to an entry in the [registry](#), in which case the sequence will change as the registry entry changes.

- Activate statistics collection by setting the statistics attribute to enable. In this mode the sequence will keep track of the number of messages processed and their processing times. For more information, see [Monitoring WSO2 ESB Using WSO2 Analytics](#).
- Activate trace collection by setting the trace attribute to enable. If tracing is enabled on a sequence, all messages being processed through the sequence will write tracing information through each mediation step.
- Use the onError attribute to define a custom error handler sequence. If an error occurs while executing the sequence, this error handler will be called. If you do not specify an error handler, the fault sequence will be used, as described in the next section.

#### **About the main and fault sequences**

A mediation configuration holds two special sequences named **main** and **fault**. All messages that are not destined for [proxy services](#) are sent through the main sequence. By default, the main sequence simply sends a message without mediation, so to add message mediation, you add mediators and/or named sequences in the main sequence.

By default, the fault sequence will log the message, the payload, and any error/exception encountered, and the [drop mediator](#) stops further processing. You should configure the fault sequence with the correct error handling instead of simply dropping messages. For more information, see [Error Handling](#).

## **Working with Sequences via WSO2 ESB Tooling**

You can create a sequence in your ESB Config project or in the registry and then add it right to that project's mediation workflow, or you can refer to it from a sequence mediator in the same ESB Config project or another project in this Eclipse workspace.

This section describes how to [create a new sequence](#) or [import an existing sequence](#) from an XML file (such as a Synapse Configuration file), and how to use the sequence in your mediation flow.

#### **About dynamic sequences**

WSO2 ESB tooling allows you to create a Registry Resource project, which can be used to store Resources and Collections you want to deploy to the registry of a Carbon Server through a Composite Application (C-App) project. When you create a sequence, you can save it as a dynamic sequence in the Registry Resource project and refer to that sequence from the mediation flow. At runtime, when you deploy the CAR file with both the Registry Resource project and mediation flow, WSO2 ESB looks up and uses the sequence from the registry.

#### **Creating a new sequence**

Follow these steps to create a new, reusable sequence that you can add to your mediation workflow or refer to from a sequence mediator, or to create a sequence mediator and its underlying sequence all at once.

##### **To create a reusable sequence:**

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Sequence** on the **Developer Studio Dashboard**.
3. Select **Create New Sequence** and click **Next**.
4. Specify a unique name for the sequence.

### **Creating a Main Sequence**

If you want to create the default main sequence that just sends messages without mediation, be sure to name it `main`, which automatically populates the sequence with the default in and out sequences.

5. Do one of the following:
  - To save the sequence in an existing ESB Config project in your workspace, click **Browse** and select

- that project.
- To save the sequence in a new ESB Config project, click **Create new Project** and create the new project.
  - To save the sequence as a **dynamic sequence** in a Registry Resource project, click **Make this as Dynamic Sequence**, specify the registry space (Governance or Configuration), click the **Browse** button at the top of the dialog box next to **Save Sequence in** and select the registry resource project, and then type the sequence name in the Registry Path box.
6. Optionally, in the **Advanced Configuration** section, specify another sequence to run when there is an error and the endpoint where messages should be sent.
  7. Click **Finish**. The sequence is created in the sequences folder under the ESB Config or Registry Resource project you specified, and the sequence is open in the editor.
  8. Add the endpoints and other sequences you want in this sequence and then click **File > Save**.

The sequence is now available in the **Defined Sequences** section of the tool palette and ready for use.

#### To create a sequence when creating a sequence mediator:

1. With your proxy service open in the editor, click **Sequence Mediator** in the tool palette and then click the location in the mediation workflow where you want to add this sequence.  
The sequence mediator is added to the workflow with a default name, which is highlighted and ready for you to change.
2. Type the name you want for this sequence mediator and press **Enter**.
3. Double-click the sequence mediator you just added. A sequence is created and opened in the editor using the same name you entered for the sequence mediator.
4. Add the endpoints and other sequences you want in this sequence, and then click **Save**.

The mediation workflow is updated with the endpoints you added to the sequence. The sequence is also now available in the **Defined Sequences** section of the tool palette and ready for use in other mediation workflows.

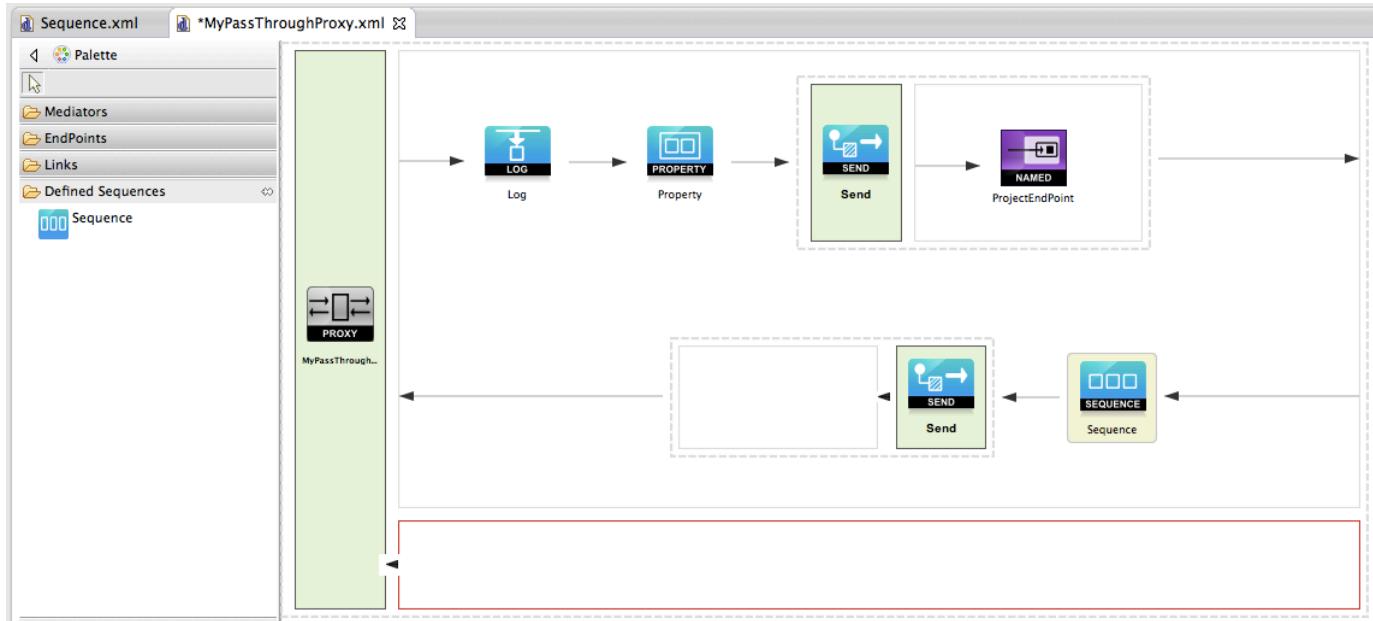
#### *Importing a sequence*

Follow these steps to import an existing sequence from an XML file (such as a Synapse configuration file) into an ESB Config project.

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Sequence** on the **Developer Studio Dashboard**.
3. Select **Import Sequence** and click **Next**.
4. Specify the sequence file by typing its full path name or clicking **Browse** and navigating to the file.
5. In the **Save Sequence In** field, specify an existing ESB Config project in your workspace where you want to save the sequence, or click **Create new Project** to create a new ESB Config project and save the sequence there.
6. In the **Advanced Configuration** section, select the sequences you want to import.
7. Click **Finish**. The sequences you selected are created in the sequences folder under the ESB Config project you specified, and the first sequence is open in the editor.

#### *Using a sequence*

After you create a sequence, it appears in the **Defined Sequences** section of the tool palette. To use this sequence in a mediation flow, click the sequence in the tool palette and then click the spot on the canvas where you want the sequence to appear in the flow. The editor automatically adds any endpoints you used in your sequence.



If you want to use a sequence from a different project or from the registry, you need to create a sequence mediator and then refer to the sequence as follows:

1. Click **Sequence Mediator** on the tool palette, and then click the spot on the canvas where you want the sequence to appear in the mediation workflow.
2. Press **Enter** to accept the default name for now.
3. In the **Properties** pane at the bottom of the window, click **Static Reference Key**, and then click the browse [...] button on the right.

Core	Property	Value
Appearance	Referring Sequence Type	Static
	Static Reference Key	Sequence

4. In the Resource Key Editor, click **Registry** if the sequence is stored in the registry or **Workspace** if it's in another ESB Config project in this Eclipse workspace.
5. If you are trying to select a sequence from the registry and no entries appear in the dialog box, click the add registry connection button (the first button in the upper right corner) and connect to the registry where the sequence resides.
6. Navigate to the sequence you want, select it and click **OK**, and then click **OK** again.

The sequence mediator name and static reference key are updated to point to the sequence you selected.

## Working with Sequences via the Management Console

You can add, edit and delete sequences via the ESB Management Console.

See the following topics for information on how to work with sequences via the Management Console:

- Adding a Mediation Sequence
- Deleting a Sequence
- Editing a Mediation Sequence

### Adding a Mediation Sequence

Sequences can be created as named sequences and referred later from other parts of the ESB configuration. Some configurations expect sequences to be defined in the place they are used. We call this type of sequences "in-line sequences."

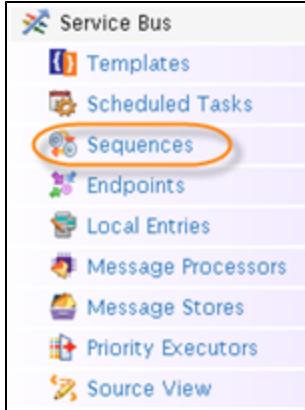
A sequence editor is used in both cases for creating a sequence. Adding a sequence in the WSO2 ESB Management Console is equivalent to directly specifying the mediators of the sequence within the <definitions> element.

Follow the instructions below to add a new sequence in the ESB Management Console.

1. Click the Main tab in the Management Console.



2. Under Manage -> Service Bus, click **Sequences**.



3. In the Mediation Sequences dialog, click **Add Sequence**.

## Mediation Sequences

The screenshot shows the 'Mediation Sequences' page. At the top, there is a button labeled 'Add Sequence' with a plus sign icon. Below it, there are two tabs: 'Defined Sequences' (selected) and 'Dynamic Sequences'. A search bar labeled 'Search Sequence' with a magnifying glass icon is positioned below the tabs. The main area displays a table titled 'Available defined Sequences in the Synapse Configuration'. The table has columns for 'Sequence Name' and 'Actions'. It lists two sequences: 'fault' and 'main'. For each sequence, there are two buttons: 'Edit' (pencil icon) and 'Delete' (trash bin icon). The 'Actions' column contains icons for 'Enable Statistics' (bar chart) and 'Enable Tracing' (telescope).

Sequence Name	Actions
fault	
main	

- In the Design Sequence dialog, enter a unique name for this sequence.

The screenshot shows the 'Design Sequence' dialog. At the top, it says 'Sequence' and has a link 'switch to source view'. Below that, there is a field 'Sequence Name\*' with a red asterisk indicating it is required. Next to it is a 'Configuration Registry' button. An 'On Error Sequence' field is followed by 'Configuration Registry' and 'Governance Registry' buttons. The main area is a tree view with a root node 'Root' and a 'Add Child' button. Below the tree is a 'Sequence Description' section with a dropdown arrow. At the bottom, there are four buttons: 'Save', 'Save As', 'Apply', and 'Cancel'.

- Optionally, add the "On Error" sequence. You can select it from "Configuration Registry" or "Governance Registry."
 

For more information, see in [Working with the Registry](#).
- Click **Add Child** to add the first mediator to your sequence tree.
 

For more information on mediators, see [Mediators](#).
- Click a mediator to edit its properties under the sequence tree, or if you are familiar with the Synapse configuration language and want to edit the XML directly, click **Switch to source view**.

The screenshot shows the 'Sequence Source' editor in the WSO2 ESB interface. The XML code for the sequence is displayed in a syntax-highlighted text area:

```

1 <sequence xmlns="http://ws.apache.org/ns/synapse" name="main">
2 <in>
3 <log level="full" />
4 <filter xmlns:ns="http://org.apache.synapse/xsd" source="get-property('To')" regex="http://localhost:9000"
5 <then>
6 <send />
7 </then>
8 <else />
9 </filter>
10 </in>
11 <out>
12 <send />
13 </out>
14 </sequence>

```

Below the code, there are buttons for 'Save', 'Save As', and 'Cancel'. A status bar at the bottom shows 'Position: Ln 1, Ch 1' and 'Total: Ln 14, Ch 343'.

- Continue adding mediators as children at the root level or as children/siblings of an existing mediator.



For more information, see [Adding a Mediator to a Sequence](#), [Editing a Mediator](#), [Adding a Child Mediator](#), and [Deleting a Mediator](#). You can add siblings to all mediators, while you can add a child only to some mediators (called "nodes").

- Place the added mediators in the necessary order. To move a mediator up or down, select it and then click on the arrow icon displayed in the node menu.



For more information, see [Editing a Mediation Sequence](#).

- Optionally, click **Sequence Description** and add a description of this sequence.
- Click **Save** to save the sequence as a Defined sequence in the Synapse configuration (default path is `/repository/synapse/sequences`), or click **Save As** to save the sequence as a Dynamic sequence in the registry. The sequence will appear on the appropriate tab in the Mediation Sequences dialog.

The screenshot shows the 'Mediation Sequences' dialog. It has tabs for 'Defined Sequences' (selected) and 'Dynamic Sequences'. Below the tabs, a message says 'Dynamic Sequences Saved in Registry'. A table lists sequences:

Sequence Name	Actions
gov/Sequence_1	Edit  Delete

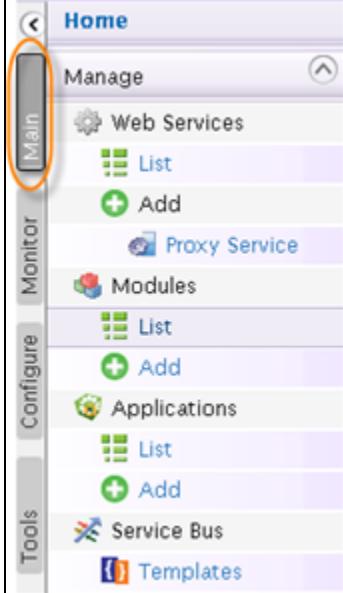
For more information, see [Working with the Registry](#).

## Deleting a Sequence

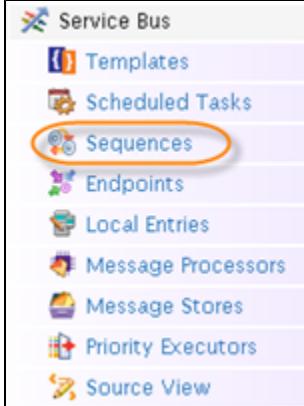
If you no longer need an existing [sequence](#), you can delete it from the ESB.

To delete a sequence:

1. In the ESB Management Console, click **Main** in the left menu to access the Manage menu.



2. In the Manage menu, click **Sequences** under Service Bus.



3. Select the tab where your sequence is located:

**Defined Sequences** - Shows sequences saved in Synapse configuration.

**Dynamic Sequences** - Shows sequences saved in the [Registry](#).

4. Click the check box for each sequence you want to delete, and then click the **Delete** link above or below the table.

Clicking the Delete link in a specific row will only delete the sequence in that row. To delete multiple sequences, be sure to click the Delete link that appears above or below the table.

5. Click **Yes** to confirm that you want to delete the selected sequences.

## Editing a Mediation Sequence

You can change a sequence properties and its mediators through the Management Console UI or using XML. The "main" and "fault" sequences are created and pre-configured automatically when you install the ESB, but you should edit them to suit your needs. Note that you should not rename a sequence once it has been created.

The following topics describe how to edit a sequence:

- Accessing the Sequence Editor

- Moving a Mediator
- Adding On Error Sequence
- Configuring XML
- Enabling/Disabling Statistics
- Enabling/Disabling Tracing

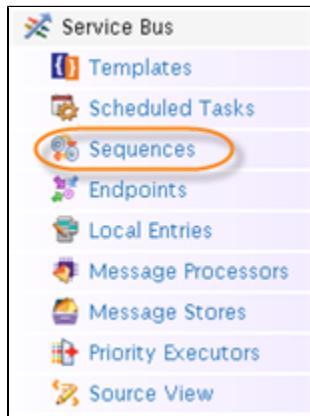
### **Accessing the Sequence Editor**

Follow the instructions below to access the Sequence Editor in the ESB Management Console.

1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Click "Main" in the left menu to access the "Manage" menu.



3. In the "Manage" menu, click on the "Sequences" under the "Service Bus."



4. Select the tab where your sequence is located:

- **Defined Sequences** - Shows sequences saved in Synapse configuration.
- **Dynamic Sequences** - Shows sequences saved in the [Registry](#).

5. Locate a sequence to edit and click on the "Edit" link in the "Actions" column. If you have a long list of defined sequences, you can easily find a sequence by typing part or all of its name in the Search Sequence field and clicking the search icon.

## Mediation Sequences

The screenshot shows the 'Mediation Sequences' page. At the top, there is a button labeled '+ Add Sequence'. Below it, two tabs are visible: 'Defined Sequences' (selected) and 'Dynamic Sequences'. A search bar labeled 'Search Sequence' with a magnifying glass icon is present. The main area displays a table titled 'Sequence Actions' with two rows:

Sequence	Name	Actions
fault	fault	Enable Statistics     Enable Tracing     Edit     Delete
main	main	Enable Statistics     Enable Tracing     Edit     Delete

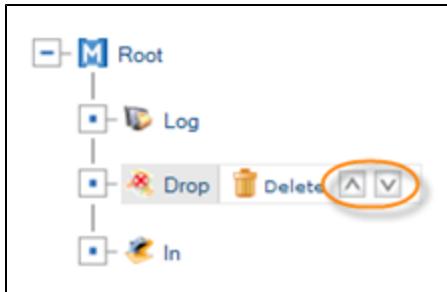
6. The "Edit Sequence" page appears.

The screenshot shows the 'Edit Sequence' page. The title is 'Edit Sequence' with a 'switch to source view' link. The 'Sequence Name\*' field contains 'fault'. The 'On Error Sequence' field has a dropdown menu with 'Configuration Registry' and 'Governance Registry' options. The main area shows a sequence tree under 'Root': 'Log' and 'Drop'. At the bottom, there is a 'Sequence Description' section with a checked checkbox. The footer contains 'Save', 'Save As', 'Apply', and 'Cancel' buttons.

### Moving a Mediator

Follow the instructions below to move a mediator in a sequence in the Sequence Editor.

1. Click on a mediator in the sequence tree.
2. Click on the arrow icon in the mediator menu to move the selected node up or down.



- Click "Save," "Save as" or "Apply" to add "On Error Sequence."

### **Adding On Error Sequence**

The ESB executes the specified error handler sequence closest to the point where the error was encountered.

Follow the instructions below to add an "On error sequence" in the [Sequence Editor](#).

- Click on the "Governance Registry" or the "Configuration Registry" link to select a previously created sequence to be used on error.

**Edit Sequence**

Sequence [switch to source view](#)

Sequence Name\*

On Error Sequence [Configuration Registry](#) [Governance Registry](#)

Root

- Log
- Drop

Sequence Description

Save Save As Apply Cancel

- In the displayed Registry window, select a sequence and click on the "OK" button.

- Click "Save," "Save As" or "Apply" to add "On Error Sequence."

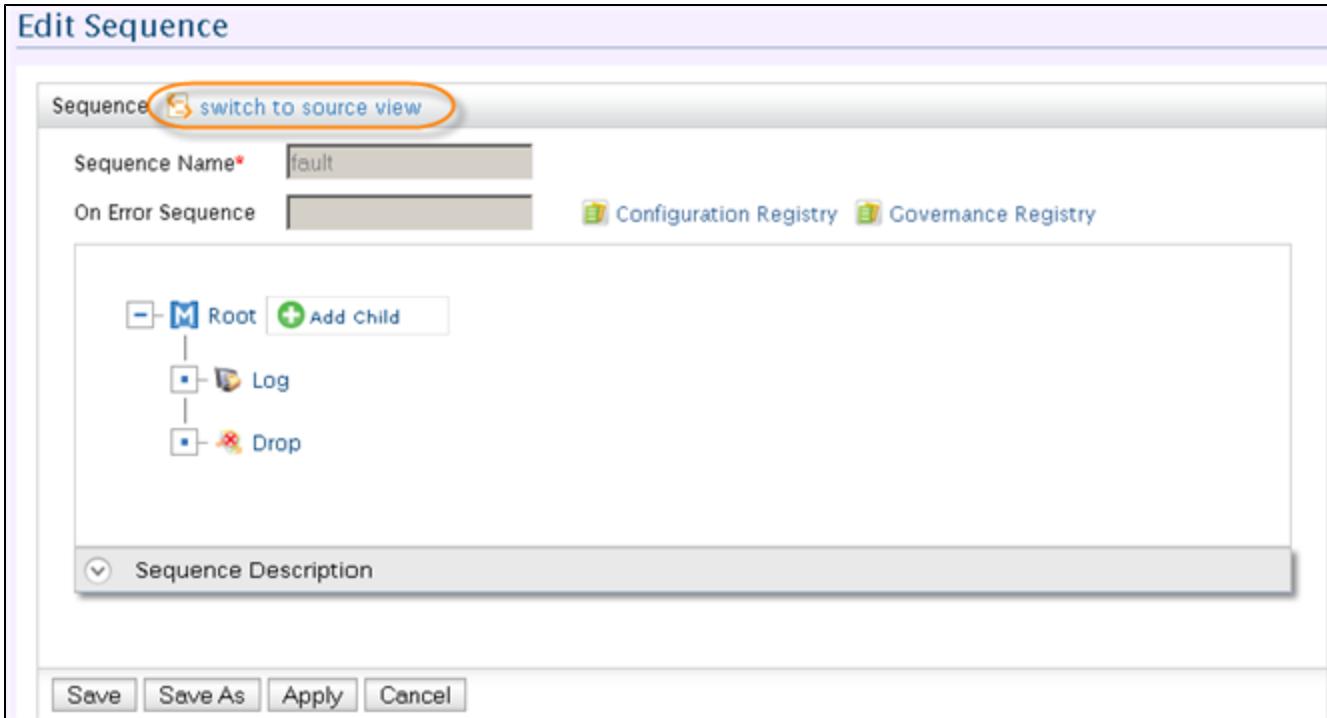


### **Configuring XML**

If you are familiar with the Synapse configuration language, you can edit the XML directly.

Follow the instructions below to edit a sequence using XML in the [Sequence Editor](#).

- Click on the "switch to source view" link to view the XML configuration of a particular sequence.



2. Edit the sequence using XML in the following window:

The screenshot shows an XML editor window with the title 'Sequence switch to design view'. It displays the XML code for a sequence named 'fault'. The code includes properties for log level ('full'), error code ('MESSAGE'), and error message ('ERROR\_CODE'). The XML is as follows:

```

<sequence xmlns="http://ws.apache.org/ns/synapse" name="fault" onerror="conf:/repository/synapse/default/sequence/fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default 'fault' sequence" />
 <property xmlns:ns="http://org.apache.synapse/xsd" name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property xmlns:ns="http://org.apache.synapse/xsd" name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop />
</sequence>

```

At the bottom of the editor are buttons for 'Save', 'Save As', and 'Cancel'. There is also a checked 'Toggle editor' checkbox.

## Note

When you are editing a sequence the name of the sequence should not be edited and the changes to the sequence name will not be effective.

3. Click "Save" or "Save As."



### **Enabling/Disabling Statistics**

Follow the instructions below to enable/disable statistics for a sequence.

- In the "Mediation Sequence" page, click on the "Enable Statistics" or "Disable Statistics" links associated with a certain sequence. See [Accessing the Sequence Editor](#) to reach the "Mediation Sequence" page.

## Mediation Sequences

Sequence Name	Actions
fault	Disable Statistics     Disable Tracing     Edit     Delete
main	Enable Statistics     Enable Tracing     Edit     Delete

2. Statistics will be enabled/disabled for that particular sequence.

### Enabling/Disabling Tracing

Follow the instructions below to enable/disable tracing for a sequence.

1. In the "Mediation Sequence" page, click on the "Enable Tracing" or "Disable Tracing" links associated with a certain sequence. See [Accessing the Sequence Editor](#) to reach the "Mediation Sequence" page.

Sequence Name	Actions
fault	Disable Statistics     Disable Tracing     Edit     Delete
main	Enable Statistics     Enable Tracing     Edit     Delete

2. Tracing will be enabled/disabled for that particular sequence.

## Working with Message Stores and Message Processors

This section provides information on WSO2 ESB's message stores and message processors as well as how message stores and message processors can be used to store and forward messages while guaranteeing reliable message delivery.

For detailed information on message stores, message processors and guaranteed delivery see the following topics:

- [Message Stores](#)
- [Message Processors](#)
- [Guaranteed Delivery with Failover Message Store and Scheduled Failover Message Forwarding Processor](#)

### Message Stores

A **message store** is used to temporarily store messages before they are delivered to their destination by a **message processor**. This approach is useful for serving traffic to back-end services that can only accept messages at a given rate, whereas incoming traffic to the ESB arrives at different rates. To store incoming traffic in a message store, use the [Store mediator](#), and then use a message processor to deliver messages to the back-end service at a given rate.

Multiple message processors can use the same message store. For example, in a clustered environment, each of the nodes would have an instance of the same message processor, each of which would connect to the same

message store and evenly consume messages from it. The message store acts as a manager of these consumers and their connections and ensures that messages are processed by only one message processor, preventing message duplication. You can further control which nodes a message processor runs on by specifying pinned servers.

You can implement your own message store by implementing the `MessageStore` interface and adding the store to the configuration. The ESB ships with the following message store implementations:

- [In Memory Message Store](#)
- [JMS Message Store](#)
- [RabbitMQ Message Store](#)
- [JDBC Message Store](#)
- [Custom Message Store](#)

## Message Store Configuration

```
<messageStore name="string" class="className">
 <parameter name="string" > "string" </parameter>/*
</messageStore>
```

You enable a message store in the ESB configuration by adding the `<messageStore>` element. This element is a top-level entry in the configuration and must have a unique name. The `class` attribute value is the fully qualified class name of the underlying message store implementation, and the parameters section is used to configure the parameters that are needed by that implementation.

You can [add](#), [edit](#), and [delete](#) message stores in the Management Console.

### Note

Message Store does not work in tenant mode. It is a limitation in the current implementation.

## Sample

[Sample 700: Introduction to Message Store](#)

[Working with Message Stores via WSO2 ESB Tooling](#)

You can create a new message store or import an existing message store from the file system using WSO2 ESB tooling.

You need to have WSO2 ESB tooling installed to create a new message store or to import an existing message store via ESB tooling. For instructions on installing WSO2 ESB tooling, see [Installing WSO2 ESB Tooling](#).

Creating a message store

Follow these steps to create a new message store. Alternatively, you can [import an existing message store](#).

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Message Store** on the **Developer Studio Dashboard**.
3. Leave the first option selected and click **Next**.
4. Type a unique name for this message store, specify the type of store you are creating, and then specify values for the other fields required to create the store type you selected.
5. Do one of the following:
  - To save the message store in an existing ESB Config project in your workspace, click **Browse** and

select that project.

- To save the message store in a new ESB Config project, click **Create a new ESB Project** and create the new project.
6. Click **Finish**. The message store is created in the `src/main/synapse-config/message-stores` folder under the ESB Config project you specified and appears in the editor. You can click its icon in the editor to view its properties.

#### Importing a message store

Follow these steps to import an existing message store into an ESB Config project. Alternatively, you can [create a new message store](#).

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Message Store** on the **Developer Studio Dashboard**.
3. Select **Import a Message Store** and click **Next**.
4. Specify the XML file that defines the message store by typing its full path name or clicking **Browse** and navigating to the file.
5. In the **Save In** field, specify an existing ESB Config project in your workspace where you want to save the message store, or click **Create new Project** to create a new ESB Config project and save the message store configuration there.
6. Click **Finish**. The message store is created in the `src/main/synapse-config/message-stores` folder under the ESB Config project you specified and appears in the editor.

#### Working with Message Stores via the Management Console

You can easily add a message store, edit a message store as well as delete message stores that are no longer required via the Management Console. The following topics describe how you can work with message stores via the Management Console:

- [Adding a Message Store](#)
- [Deleting a Message Store](#)
- [Editing a Message Store](#)

#### **Adding a Message Store**

WSO2 ESB allows you to add a required type of [message store](#) to your ESB implementation via the **Manage Message Stores** screen of the ESB Management Console.

WSO2 ESB supports the following message store types:

- [In Memory Message Store](#)
- [JMS Message Store](#)
- [RabbitMQ Message Store](#)
- [JDBC Message Store](#)
- [Custom Message Store](#)

#### **To add a required message store to your ESB implementation**

1. Click the **Main** tab on the Management Console, go to **Manage -> Service Bus** and then click **Message Stores**. The **Manage Message Stores** screen appears.
2. Click **Add Message Stores**. You will see the list of available message stores.
3. Click the required type of message store you need to add.
4. Enter the required values to create the message store, and then click **Save**.

#### **Note**

When adding a message store, the parameters that you need to specify can vary depending on the type of message store that you add. For more information on each message store and for

descriptions of the parameters you need to specify when adding each type of message store, see the relevant topic from the following:

- In Memory Message Store
- JMS Message Store
- RabbitMQ Message Store
- JDBC Message Store
- Custom Message Store

5. To view the message store that you added, click **Available Message Stores** on the **Manage Message Stores** screen. This displays details of all the message stores you have added.

The screenshot shows the 'Manage Message Stores' interface. At the top, there are two buttons: 'Available Message Stores' (which is selected) and 'Add Message Stores'. Below this is a table with three rows, each representing a message store:

Message Store Name	Type	Messages	Actions
store2	org.apache.synapse.message.store.impl.memory.InMemoryStore	0	Edit  Delete
store3	org.apache.synapse.message.store.impl.jdbc.JDBCMessageStore	0	Edit  Delete
store1	org.apache.synapse.message.store.impl.jms.JmsStore	Not Applicable	Edit  Delete

## In Memory Message Store

In memory message store is a basic **Message Store** that stores messages in an in-memory queue. Since the messages are stored in an in-memory queue in case of a ESB restart, all the messages stored will be lost.

The in memory message store is lot faster than a persistent message store implementation, so it can be used to temporarily store messages for use cases such as the implementation of a high-speed store and forwarded pattern where message persistence is not a requirement.

## Note

In memory message stores are not recommended for use in production as well as in scenarios where large scale message storing is required. You can use an external message store (e.g., [JMS message store](#)) for such scenarios.

### UI Configuration

Following is the **Add In Memory Message Store** screen that you will see on the ESB Management Console.

The screenshot shows the 'Add In Memory Message Store' dialog box. At the top, there is a header 'Add In Memory Message Store' with a 'Switch to source view' link. Below the header is a form with a 'Name \*' field containing 'InMemoryMS'. At the bottom of the dialog are two buttons: 'Save' and 'Cancel'.

When you add an in memory message store, you need to specify a unique name for the message store.

For instructions on adding a required type of message store via the ESB Management Console, see [Adding a Message Store](#).

Following is a sample in memory message store configuration:

```
<messageStore name="InMemoryMS"
class="org.apache.synapse.message.store.impl.memory.InMemoryStore"
xmlns="http://ws.apache.org/ns/synapse"></messageStore>
```

## JMS Message Store

JMS message store persists messages in a JMS queue inside a **JMS Broker**. The JMS message store can be configured by specifying the class as `org.apache.synapse.message.store.impl.jms.JmsStore`.

Since the JMS message stores persist messages in a JMS queue in an ordered manner, JMS message stores can be used to implement the store-and-forward pattern.

UI Configuration

Following is the **Add JMS Message Store** screen that you will see on the ESB Management Console.

Add JMS Message Store	
Name *	JMSMS
Initial Context Factory *	org.apache.activemq.jndi.ActiveMQInitialContextFactory
Provider URL *	tcp://localhost:61616
<input type="checkbox"/> Show Additional Parameters <input type="checkbox"/> Show Guaranteed Delivery Parameters	
<input type="button" value="Save"/> <input type="button" value="Cancel"/>	

When you add a JMS message store, it is required to specify values for the following:

- **Name** - A unique name for the JMS message store.
- **Initial Context Factory** (`java.naming.factory.initial`) - The JNDI initial context factory class. This class must implement the `java.naming.spi.InitialContextFactory` interface.
- **Provider URL** (`java.naming.provider.url`) - The URL of the JNDI provider.

In addition to specifying the required parameters, you can click **Show Additional Parameters** and set any of the additional parameters as necessary. For descriptions of each of the additional parameters you can set, see [Additional JMS message store parameters](#).

If you need to ensure guaranteed delivery when you store incoming messages to a JMS message store, and later deliver them to a particular backend, click **Show Guaranteed Delivery Parameters** and specify values for the following parameters:

- **Enable Producer Guaranteed Delivery** (`store.producer.guaranteed.delivery.enable`) - Whether it is required to enable guaranteed delivery on the producer side.
- **Failover Message Store** (`store.failover.message.store.name`) - The message store to which the store mediator should send messages when the original message store fails.

For instructions on adding a required type of message store via the ESB Management Console, see [Adding a Message Store](#).

[Additional JMS message store parameters](#)

Parameter Name	Value	Required
JNDI Queue Name ( <code>store.jms.destination</code> )	The message store queue name.	Though this is not a required parameter, we recommend specifying a value for this.
Connection factory ( <code>store.jms.connection.factory</code> )	The JNDI name of the connection factory that is used to create jms connections	Though this is not a required parameter, we recommend specifying a value for this.
User Name ( <code>store.jms.username</code> )	The user name to connect to the broker.	No
Password ( <code>store.jms.password</code> )	The password to connect to the broker.	No
JMS API Specification Version ( <code>store.jms.JMSSpecVersion</code> )	The JMS API version to be used. Possible values are 1.1 or 1.0.	No. The default value is 1.1.
vender.class.loader.enabled	Set to <b>false</b> when using IBM MQ, which requires skipping the external class loader.	No, except when using IBM MQ

Following is a sample JMS message store configuration that uses WSO2 MB as the message broker:

```
<messageStore name="JMSMS" class="org.apache.synapse.message.store.impl.jms.JmsStore"
 xmlns="http://ws.apache.org/ns/synapse">
 <parameter
 name="java.naming.factory.initial">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 <parameter
 name="java.naming.provider.url">repository/conf/jndi.properties</parameter>
 <parameter name="store.jms.destination">ordersQueue</parameter>
 <parameter name="store.jms.connection.factory">queue</parameter>
 <parameter name="store.jms.JMSSpecVersion">1.1</parameter>
</messageStore>
```

When using WSO2 MB as the message broker and configuring a [Message Processor](#) with `max.delivery.attempts`, if WSO2 MB does not get an acknowledgment from ESB, it re-sends the message resulting in duplicate messages being delivered. To avoid this, add the following line in `<ESB_HOME>/bin/wso2server.sh` file:

```
-DAndesAckWaitTimeOut=3600000 \
```

Following is a sample JMS message store configuration that uses ActiveMQ as the message broker:

```

<messageStore name="JMSMS" class="org.apache.synapse.message.store.impl.jms.JmsStore"
xmlns="http://ws.apache.org/ns/synapse">
<parameter name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
<parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
<parameter name="store.jms.destination">ordersQueue</parameter>
<parameter name="store.jms.connection.factory">queue</parameter>
<parameter name="store.jms.JMSSpecVersion">1.1</parameter>
</messageStore>

```

## Note

When configuring a JMS message store with WSO2 MB or Active MQ you need to copy the required client libraries to the <ESB\_HOME>/repository/component/lib directory. If the relevant client libraries are not copied you will see a `java.lang.ClassNotFoundException`.

For information on the client libraries you need to copy when configuring a JMS message store with WSO2 MB, see [Configure with WSO2 Message Broker](#).

For information on the client libraries you need to copy when configuring a JMS message store with ActiveMQ, see [Configure with ActiveMQ](#).

Individual message priorities can be set using the following property on the provider. For example, the value can be 0-9 for ActiveMQ.

```
<property name="JMS_PRIORITY" value="9" scope="axis2"/>
```

## Note

If you are using ActiveMQ 5.12.2 and above, you need to set the following system property on server start up for WSO2 ESB's JMS message store to work as expected.

```
-Dorg.apache.activemq.SERIALIZABLE_PACKAGES="*"
```

With ActiveMQ 5.12.2 and above, you need to set the above property because users are enforced to explicitly whitelist packages that can be exchanged using ObjectMessages, and due to this restriction the message processor fails to read messages from ActiveMQ with the following error:

```

ERROR - JmsConsumer [JMS-C-1] cannot receive message from store. Error:Failed to
build body from content. Serializable class not available to broker. Reason:
java.lang.ClassNotFoundException: Forbidden class
org.apache.synapse.message.store.impl.commons.StorableMessage! This class is not
trusted to be serialized as ObjectMessage payload.

```

For information on configuring the JMS message store with different message brokers, see [Store and Forward Using JMS Message Stores](#).

## RabbitMQ Message Store

RabbitMQ message store persists messages in a RabbitMQ queue inside a RabbitMQ broker. The RabbitMQ message store can be configured by specifying the class as `org.apache.synapse.message.store.impl.rabbitmq.RabbitmqStore` and then setting all other required parameters to connect to a RabbitMQ broker.

#### UI Configuration

Following is the **Add RabbitMQ Message Store** screen that you will see on the ESB Management Console.

Add RabbitMQ Message Store	
Name *	<input type="text" value="RabbitMS"/>
RabbitMQ Server Host Name *	<input type="text" value="localhost"/>
RabbitMQ Server Host Port *	<input type="text" value="5672"/>
SSL Enabled	<input checked="" type="checkbox" value="false"/> false
<input checked="" type="checkbox"/> Show Additional Parameters <input type="checkbox"/> Show Guaranteed Delivery Parameters	
<input type="button" value="Save"/> <input type="button" value="Cancel"/>	

When you add a RabbitMQ message store, it is required to specify values for the following:

- **Name** - A unique name for the RabbitMQ message store.
- **RabbitMQ Server Host Name** (`store.rabbitmq.host.name`) - The address of the RabbitMQ broker.
- **RabbitMQ Server Host Port** (`store.rabbitmq.host.port`) - The port number of the RabbitMQ message broker.
- **SSL Enabled** - Whether or not SSL is enabled on the message store.

When **SSL Enabled** is set to true, you can set the parameters relating to the SSL configuration. For descriptions of each of these parameters you can set, see [SSL enabled RabbitMQ message store parameters](#).

In addition to specifying the required parameters, you can click **Show Additional Parameters** and set any of the additional parameters as necessary. For descriptions of each of the additional parameters you can set, see [Additional RabbitMQ message store parameters](#).

If you need to ensure guaranteed delivery when you store incoming messages to a RabbitMQ message store, and then deliver them to a particular backend, click **Show Guaranteed Delivery Parameters** and specify values for the following parameters:

- **Enable Producer Guaranteed Delivery** (`store.producer.guaranteed.delivery.enable`) - Whether it is required to enable guaranteed delivery on the producer side.
- **Failover Message Store** (`store.failover.message.store.name`) - The message store to which the store mediator should send messages when the original message store fails.

For instructions on adding a required type of message store via the ESB Management Console, see [Adding a Message Store](#).

[SSL enabled RabbitMQ message store parameters](#)

Parameter Name	Value
SSL Key Store Location ( <code>rabbitmq.connection.ssl.keystore.location</code> )	The location of the keystore file.
SSL Key Store Type ( <code>rabbitmq.connection.ssl.keystore.type</code> )	The type of the keystore used (e.g., JKS, PKCS12).
SSL Key Store Password ( <code>rabbitmq.connection.ssl.keystore.password</code> )	The password to access the keystore.

SSL Trust Store Location ( <code>rabbitmq.connection.ssl.truststore.location</code> )	The location of the Java keystore file containing the collection of CA certificates trusted by this application process (truststore).
SSL Trust Store Type ( <code>rabbitmq.connection.ssl.truststore.type</code> )	The type of the truststore used.
SSL Trust Store Password ( <code>rabbitmq.connection.ssl.truststore.password</code> )	The password to unlock the trust store file specified in <code>rabbitmq.connection.ssl.truststore.location</code> .
SSL Version ( <code>rabbitmq.connection.ssl.version</code> )	SSL protocol version (e.g., SSL, TLSV1, TLSV1.2)

Following is a sample configuration for SSL enabled RabbitMQ message store:

```
<messageStore xmlns="http://ws.apache.org/ns/synapse"
 class="org.apache.synapse.message.store.impl.rabbitmq.RabbitMQStore"
 name="store1">
 <parameter name="store.producer.guaranteed.delivery.enable">false</parameter>
 <parameter name="store.failover.message.store.name">store1</parameter>
 <parameter name="store.rabbitmq.host.name">localhost</parameter>
 <parameter name="store.rabbitmq.queue.name">WithoutClientCertQueue</parameter>
 <parameter name="store.rabbitmq.host.port">5671</parameter>
 <parameter name="rabbitmq.connection.ssl.enabled">true</parameter>
 <parameter
 name="rabbitmq.connection.ssl.truststore.location">path/to/truststore</parameter>
 <parameter name="rabbitmq.connection.ssl.truststore.type">JKS</parameter>
 <parameter
 name="rabbitmq.connection.ssl.truststore.password">truststorepassword</parameter>
 <parameter
 name="rabbitmq.connection.ssl.keystore.location">path/to/keystore</parameter>
 <parameter name="rabbitmq.connection.ssl.keystore.type">PKCS12</parameter>
 <parameter
 name="rabbitmq.connection.ssl.keystore.password">keystorepassword</parameter>
 <parameter name="rabbitmq.connection.ssl.version">SSL</parameter>
 </messageStore>
```

Configuring parameters that provide information related to keystores and truststores can be optional based on your broker configuration.

For example, if `fail_if_no_peer_cert` is set to `false` in the RabbitMQ broker configuration, then you only need to specify `<parameter name="rabbitmq.connection.ssl.enabled" locked="false">true</parameter>`. Additionally, you can also set `<parameter name="rabbitmq.connection.ssl.version" locked="false">true</parameter>` parameter to specify the SSL version. If `fail_if_no_peer_cert` is set to `true`, you need to provide keystore and truststore information.

Following is a sample message store configuration where `fail_if_no_peer_cert` is set to `false`:

```
{ssl_options, [{cacertfile,"/path/to/testca/cacert.pem"}, {certfile,"/path/to/server/cert.pem"}, {keyfile,"/path/to/server/key.pem"}, {verify,verify_peer}, {fail_if_no_peer_cert,false}]}
```

Additional RabbitMQ message store parameters

Parameter Name	Value	Required
----------------	-------	----------

RabbitMQ Queue Name (store.rabbitmq.queue.name)	The message store queue name.	Though this is not a required parameter, we recommend specifying a value for this.
RabbitMQ Exchange Name (store.rabbitmq.exchange.name)	The name of the RabbitMQ exchange to which the queue is bound (If the ESB has to declare this exchange it will be declared as a direct exchange).	No
Routing key (store.rabbitmq.route.key)	The exchange and queue binding value. Messages will be routed using this.	No (This is considered only when both the exchange name and type are provided. Else messages will be routed using the default exchange and queue name as the routing key).
User Name (store.rabbitmq.username)	The user name to connect to the broker.	No
Password (store.rabbitmq.password)	The password to connect to the broker.	No
Virtual Host (store.rabbitmq.virtual.host)	The virtual host name of the broker.	No

Following is a basic sample RabbitMQ message store configuration:

```
<messageStore
class="org.apache.synapse.message.store.impl.rabbitmq.RabbitmqStore" name="RabbitMS">
 <parameter name="store.rabbitmq.host.name">localhost</parameter>
 <parameter name="store.rabbitmq.queue.name">ESBStore</parameter>
 <parameter name="store.rabbitmq.host.port">5672</parameter>
</messageStore>
```

## JDBC Message Store

The JDBC message store can be used to store and retrieve messages more efficiently in comparison with other message stores.

The JDBC message store implementation is a variation of the already existing synapse message store implementation and is designed in a manner similar to the WSO2 ESB JMS message store. The JDBC message store uses a JDBC connector to connect to external relational databases.

The advantages of using a JDBC message store instead of any other message store are as follows:

- Easy to connect – You only need to have a JDBC connector to connect to an external relational database.
- Quick transactions – JDBC message stores are capable of handling a large number of transactions per second.
- Ability to work with a high capacity for a long period of time – Since JDBC stores use databases as the medium to store data, it can store a large volume of data and is capable of handling data for a longer period of time.

Configuring the JDBC message store

The syntax of the JDBC message store can be different depending on whether you connect to the database using a connection pool, or using a datasource. Click on the relevant tab to view the syntax based on how you want to connect to the database.

Syntax to connect using a connection poolSyntax to connect using an external datasource

```
<messageStore class="org.apache.synapse.message.store.jdbc.JDBCMessagesStore"
name="MyStore">

<parameter name="store.jdbc.driver"/>
<parameter name="store.jdbc.connection.url"/>
<parameter name="store.jdbc.username"/>
<parameter name="store.jdbc.password"/>
<parameter name="store.jdbc.table"/>

</messageStore>
```

Following are the parameters that should be specified and the description for each:

Parameter	Description	Required
store.jdbc.driver	The class name of the database driver.	YES
store.jdbc.connection.url	The database URL.	YES
store.jdbc.username	The user name to access the database.	YES
store.jdbc.password	The password to access the database.	NO
store.jdbc.table	Table name of the database.	YES

```
<messageStore class="org.apache.synapse.message.store.jdbc.JDBCMessagesStore"
name="MyStore">

<parameter name="store.jdbc.dsName"/>
<parameter name="store.jdbc.table"/>

</messageStore>
```

Following are the parameters that should be specified and the description for each:

Parameter	Description	Required
store.jdbc.dsName	The name of the datasource to be looked up	YES
store.jdbc.table	The table name of the database.	YES

#### UI Configuration

The UI of the JDBC Message Store that is displayed can vary depending on whether you connect to the database

using a connection pool, or using a datasource. Click on the relevant tab to view the UI that is displayed based on how you want to connect to the database.

Connect to the database via a connection pool Connect to the database via an external datasource

When adding a JDBC message store, if you select **Pool** as the **Connection Information**, the following screen appears:

The screenshot shows the 'Add JDBC Message Store' configuration page. At the top, there's a breadcrumb navigation: Home > Manage > Service Bus > Message Stores > Add JDBC Message Store. Below the title 'Add JDBC Message Store', there's a 'Switch to source view' link. The form contains the following fields:

- Name \***: A text input field.
- Database Table**: A text input field.
- Connection Information**: A radio button group with two options: **Pool** (selected) and **Carbon Datasource**.
- Driver \***: A text input field.
- Url \***: A text input field.
- User \***: A text input field.
- Password**: A text input field.

At the bottom of the form are 'Save' and 'Cancel' buttons.

Following are descriptions of the fields that are displayed:

Field	Description
Name	The name of the message store
Database Table	The name of the database table.
Driver	The class name of the database driver.
Url	The JDBC URL of the database that the data will be written to.
User	The user name used to connect to the database.
Password	The password used to connect to the database.

When adding a JDBC message store, if you select **Carbon Datasource** as the **Connection Information**, the following screen appears:

## Add JDBC Message Store

The screenshot shows the 'Add JDBC Message Store' configuration dialog. It includes fields for 'Name' (mandatory), 'Database Table', 'Connection Information' (with options for 'Pool' or 'Carbon Datasource'), and a dropdown for 'Datasource Name'. Buttons for 'Save' and 'Cancel' are at the bottom.

To make sure that the datasource appears in the **Datasource Name** list, you need to expose it as a JNDI datasource. For information on configuring a JNDI datasources, see [Configuring a JNDI Datasource](#)

Following are descriptions of the fields that are displayed:

Field	Description
Name	The name for the message store
Database Table	The name of the database table.
Datasource Name	The class name of the datasource.

Sample scenario

In this sample:

- The client sends requests to a proxy service.
- The proxy service stores the messages in a JDBC message store.
- The back-end service is invoked by a message forwarding processor, which picks the messages stored in the JDBC message store.

Prerequisites:

1. Setup the database.

- If you are setting up a MySQL database, the DB script to create the required table is as follows:

```
CREATE TABLE jdbc_message_store(
indexId BIGINT(20) NOT NULL AUTO_INCREMENT ,
msg_id VARCHAR(200) NOT NULL ,
message BLOB NOT NULL ,
PRIMARY KEY (indexId)
)
```

- If you are setting up a H2 database, the DB script to create the required table is as follows:

```
CREATE TABLE jdbc_message_store(
indexId BIGINT(20) NOT NULL AUTO_INCREMENT ,
msg_id VARCHAR(200) NOT NULL ,
message BLOB NOT NULL ,
PRIMARY KEY (indexId)
)
```

## Note

You can create a similar script based on the database you want to set up.

2. Add the relevant database driver into the repository/components/lib folder.

Configure the sample

1. Create a proxy service that stores messages to the JDBC message store.
2. Create a JDBC message store.
3. Create a message forwarding processor to consumes the messages stored in the message store.

**Sample configuration that uses a MySQL database named sampleDB and the database table jdbc\_message\_store**

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="MessageStoreProxy"
 transports="https http"
 startOnLoad="true"
 trace="disable">
 <description/>
 <target>
 <inSequence>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
 <property name="OUT_ONLY" value="true"/>
 <property name="target.endpoint" value="StockQuoteServiceEp"/>
 <store messageStore="SampleStore"/>
 </inSequence>
 </target>
 <publishWSDL
 uri="http://localhost:9000/services/SimpleStockQuoteService?wsdl"/>
</proxy>

<messageStore xmlns="http://ws.apache.org/ns/synapse"
 class="org.apache.synapse.message.store.impl.jdbc.JDBCMessageStore"
 name="SampleStore">
 <parameter name="store.jdbc.password"/>
 <parameter name="store.jdbc.username">root</parameter>
 <parameter name="store.jdbc.driver">com.mysql.jdbc.Driver</parameter>
 <parameter name="store.jdbc.table">jdbc_message_store</parameter>
 <parameter
 name="store.jdbc.connection.url">jdbc:mysql://localhost:3306/sampleDB</parameter>
</messageStore>

<messageProcessor xmlns="http://ws.apache.org/ns/synapse"
 class="org.apache.synapse.message.processor.impl.forwarder.ScheduledMessageForwardingProcessor"
 name="ScheduledProcessor"
 messageStore="SampleStore">
 <parameter name="max.delivery.attempts">5</parameter>
 <parameter name="interval">10</parameter>
 <parameter name="is.active">true</parameter>
</messageProcessor>

```

Configure the back-end service

1. Deploy the **SimpleStockQuoteService** client by navigating to `<ESB_HOME>/samples/axis2Server/src/SimpleStockQuoteService`, and running the **ant** command in the command prompt or shell script. This will build the sample and deploy the service for you. For more information on sample back-end services, see [Deploying sample back-end services](#).
2. WSO2 ESB comes with a default Axis2 server, which you can use as the back-end service for this sample. To start the Axis2 server, navigate to `<ESB_HOME>/samples/axis2server`, and run `axis2Server.sh` on Linux or `axis2Server.bat` on Windows.
3. Go to <http://localhost:9000/services/SimpleStockQuoteService?wsdl> and verify that the service is running.

Execute the sample

To invoke the proxy service, navigate to <ESB\_HOME>/repository/samples/axis2client, and execute the following command:

```
ant stockquote -Daddurl=http://localhost:8280/services/MessageStoreProxy
```

## Custom Message Store

**Custom Message Store** allows users to create a message store with their own message store implementation. It can be configured using configuration by giving the fully qualified class name of the message store implementation as the class value.

Messages will be stored as specified in the underlying message store implementation. Parameter configuration can be used to pass any configuration parameters that is needed by the message store implementation class.  
UI Configuration

1. In the "Add Message Stores" tab, click "Custom Message Store" (See [Adding a Message Store](#)). The "Add Custom Message Store" page appears with its default view.

Home > Manage > Service Bus > Message Stores > Custom

Custom Message store

Custom Switch to source view

Name \*

Provider class \*

Message Store Parameters  Name:  Value:

Save Cancel

Custom Message Store parameters:

- **Name** - Unique name of the message store
- **Provider Class** - Fully qualified name of the message store implementation class

2. Parameters can be added to the message store by giving a "Name" and "Value" to "Message Store Parameters" and clicking Add Parameter button.

Home > Manage > Service Bus > Message Stores > Add Custom Message Store

Add Custom Message Store

Add Custom Message Store Switch to source view

Name \* MyStore3

Provider class \* org.wso2.carbon.message.store.sample.CustomMessageStore

Message Store Parameters  Name: param1 Value: paramValue

Save Cancel

3. Added parameters will be listed as follows.

## Add Custom Message Store

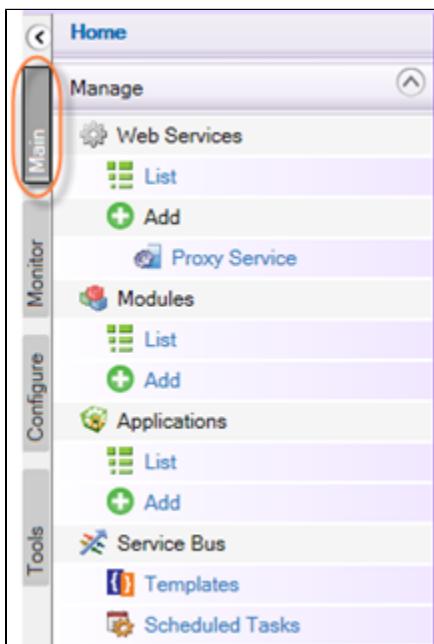
Add Custom Message Store [Switch to source view](#)

Name *	MyStore3		
Provider class *	org.wso2.carbon.message.store.sample.CustomMessageStore		
Name: <input type="text"/> Value: <input type="text"/> <a href="#">Add Parameter</a>			
Message Store Parameters	Name	Value	Action
	param1	paramValue	<a href="#">Delete</a>
<input type="button" value="Save"/> <input type="button" value="Cancel"/>			

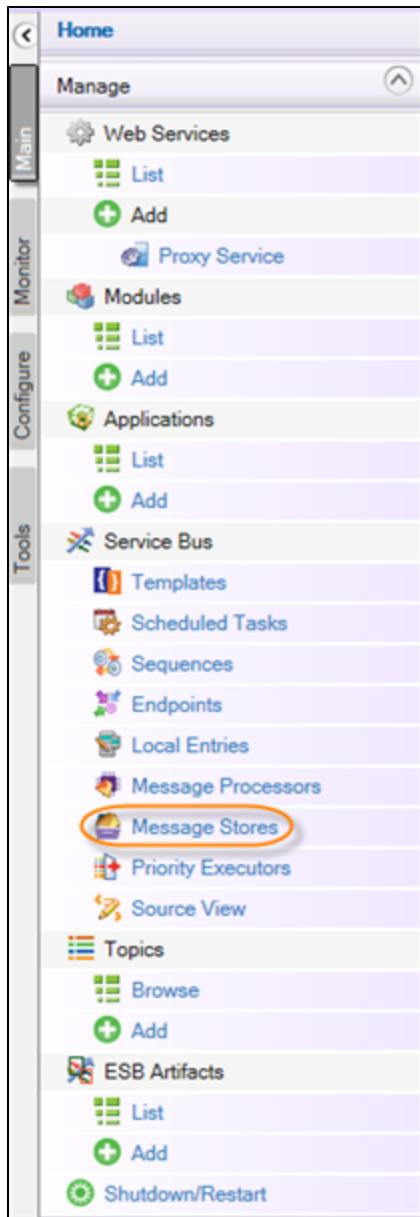
### Deleting a Message Store

Follow the instruction below to delete a Message Store.

1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Select the "Main" tab to access the "Manage" menu.



3. Click on the "Message Stores" link to access the "Manage Message Stores" page.



4. The "Manage Message Stores" page appears. Choose the Message Store you want to delete and click on the "Delete" link.

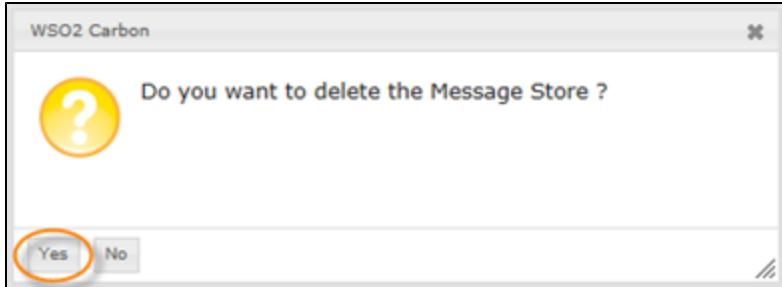
Home > Manage > Service Bus > Message Stores Help

### Manage Message Stores

[Available Message Stores](#) [Add Message Stores](#)

Message Store Name	Type	Messages	Actions
MyStore1	org.apache.synapse.message.store.InMemoryMessageStore	0	<a href="#">Edit</a> <a href="#">Delete</a>
MyStore2	org.wso2.carbon.message.store.persistence.jms.JMSMessageStore	0	<a href="#">Edit</a> <a href="#">Delete</a>
MyStore3	org.wso2.carbon.message.store.sample.CustomMessageStore	0	<a href="#">Edit</a> <a href="#">Delete</a>

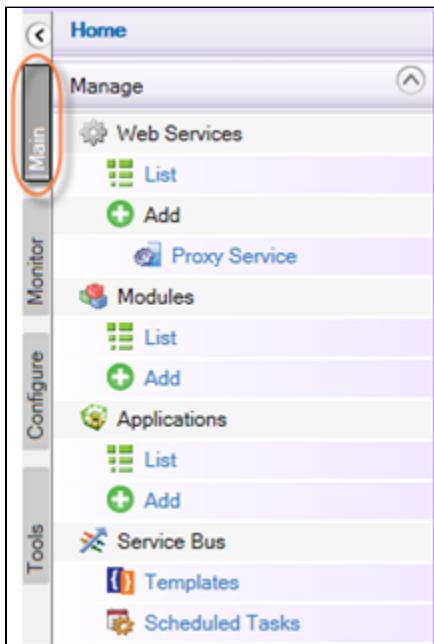
5. Click "Yes" to confirm your request in the "WSO2 Carbon" window.



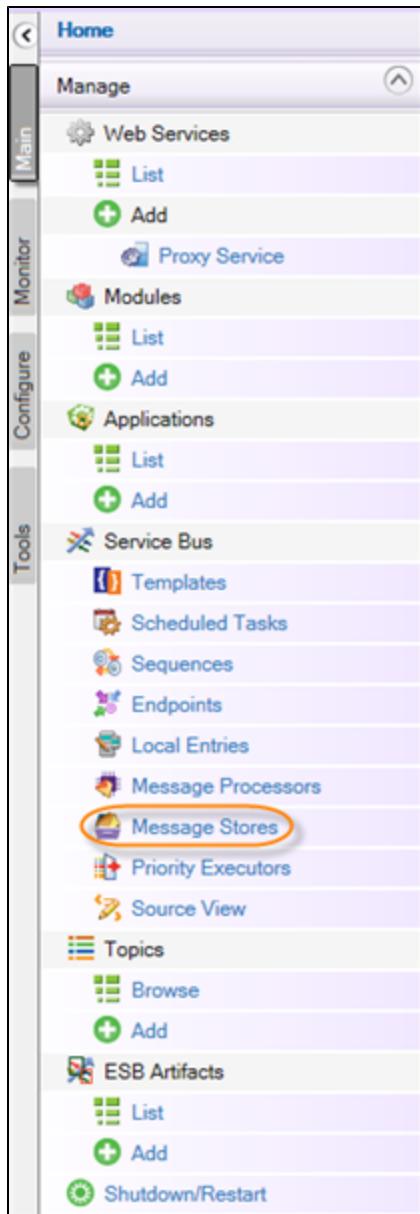
### ***Editing a Message Store***

WSO2 ESB allows to change a Message Store parameters. Follow the instruction below to edit a [Message Store](#).

1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Select the "Main" tab to access the "Manage" menu.



3. Click on the "Message Stores" link to access the "Manage Message Stores" page.



4. The "Manage Message Stores" page appears. Choose the Message Store you want to edit and click on the "Edit" link.

Message Store Name	Type	Messages	Actions
MyStore1	org.apache.synapse.message.store.InMemoryMessageStore	0	<a href="#">Edit</a> <a href="#">Delete</a>
MyStore2	org.wso2.carbon.message.store.persistence.jms.JMSMessageStore	0	<a href="#">Edit</a> <a href="#">Delete</a>
MyStore3	org.wso2.carbon.message.store.sample.CustomMessageStore	0	<a href="#">Edit</a> <a href="#">Delete</a>

5. Edit the options of the message store according to its type (For more information about the options of message stores, see [Adding a Message Store](#)).

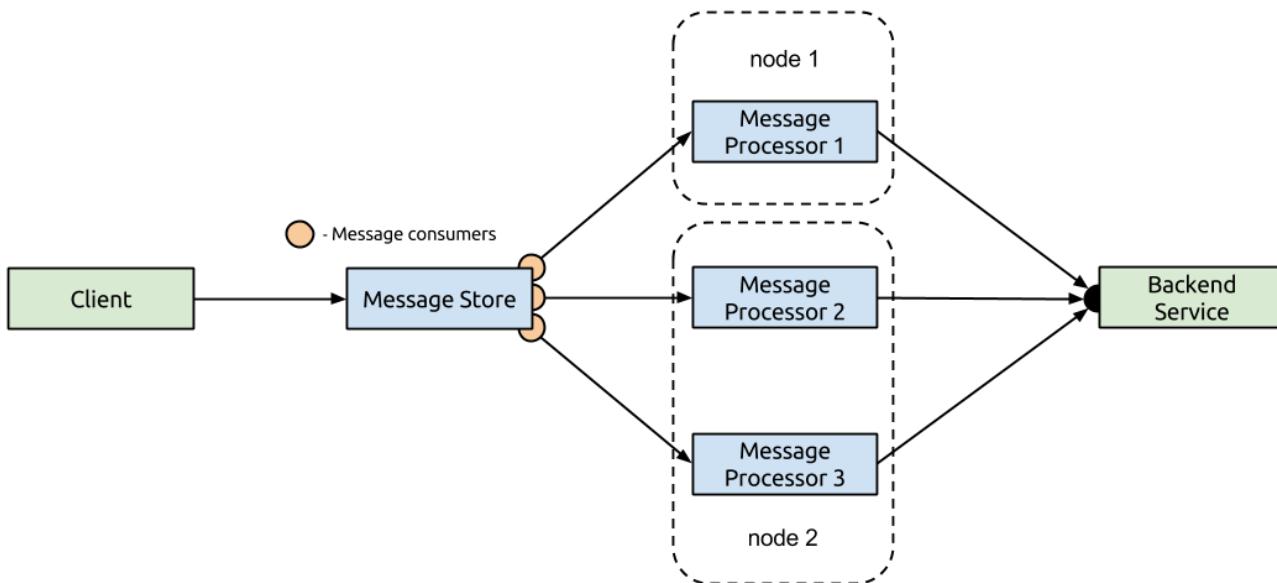
6. Click "Save" to add the alterations to the message store.



## Message Processors

A **message processor** is used to deliver messages that have been temporarily stored in a [message store](#). This approach is useful for serving traffic to back-end services that can only accept messages at a given rate, whereas incoming traffic to the ESB arrives at different rates. You use the [Store mediator](#) to store messages in the message store, and then you use a message processor to deliver messages from the message store to the back-end service at a given rate. Using message processors and message stores allows you to implement different messaging and integration patterns.

Multiple message processors can use the same message store. For example, in a clustered environment, each of the nodes might have an instance of the same message processor, each of which would connect to the same message store and evenly consume messages from it. The message store acts as a manager of these consumers and their connections and ensures that messages are processed by only one message processor, preventing message duplication. You can further control which nodes a message processor runs on by specifying pinned servers, so that you might have each message processor run on one node only, as shown in the following diagram.



The ESB ships with the following message processor implementations:

- [Scheduled Message Forwarding Processor](#)
- [Message Sampling Processor](#)

You can also implement your own message processor by implementing the [MessageProcessor](#) interface and adding the message processor to the configuration.

## Message Processor Configuration

You can [add](#), [edit](#), [delete](#), and [activate/deactivate](#) message processors by filling in the fields in the relevant screens in the Management Console. Note that you must have [added a message store](#) before you can add a message processor. You can also configure a message processor in the source view in the Management Console by adding the `<messageProcessor>` element as a top-level entry in the configuration as follows:

Forwarding processor

```
<messageProcessor class="classname" name="unique string" targetEndpoint="endpoint
name" messageStore="associated message store name">
<parameter name="string">"string"</parameter>*
</messageProcessor>
```

#### Sampling and custom processors

```
<messageProcessor class="classname" name="unique string" messageStore="associated
message store name">
<parameter name="string">"string"</parameter>*
</messageProcessor>
```

### Note

Message Processor does not work in tenant mode. It is a limitation in the current implementation.

#### Working with Message Processors via WSO2 ESB Tooling

You can create a new message processor or import an existing message processor from the file system using WSO2 ESB tooling.

- You need to have WSO2 ESB tooling installed to create a new message processor or to import an existing message processor via ESB tooling. For instructions on installing WSO2 ESB tooling, see [Installing WSO2 ESB Tooling](#).
- Be sure to create the message store before creating the message processor, as you will need to specify the message store that this processor applies to.

#### Creating a message processor

Follow these steps to create a new message processor. Alternatively, you can [import an existing message processor](#).

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Message Processor** on the **Developer Studio Dashboard**.
3. Leave the first option selected and click **Next**.
4. Type a unique name for this message processor, specify the type of processor you're creating, specify the message store this processor applies to, and then specify values for the other fields required to create the processor type you selected.
5. Do one of the following:
  - To save the message processor in an existing ESB Config project in your workspace, click **Browse** and select that project.
  - To save the message processor in a new ESB Config project, click **Create a new ESB Project** and create the new project.
6. Click **Finish**. The message processor is created in the `src/main/synapse-config/message-processors` folder under the ESB Config project you specified and appears in the editor. You can click its icon in the editor to view its properties.

#### Importing a message processor

Follow these steps to import an existing message processor into an ESB Config project. Alternatively, you can [create a new message processor](#).

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer**

### **Studio Dashboard.**

2. Click **Message Processor** on the **Developer Studio Dashboard**.
3. Select **Import a Message Processor** and click **Next**.
4. Specify the XML file that defines the message processor by typing its full path name or clicking **Browse** and navigating to the file.
5. In the **Save In** field, specify an existing ESB Config project in your workspace where you want to save the message processor, or click **Create new Project** to create a new ESB Config project and save the message processor configuration there.
6. Click **Finish**. The message processor is created in the `src/main/synapse-config/message-processors` folder under the ESB Config project you specified and appears in the editor.

### Working with Message Processors via the Management Console

You can easily add, edit and delete a message processor, as well as activate or deactivate a message processor via the Management Console. The following topics describe how you can work with message processors via the Management Console:

- Activating and Deactivating a Message Processor
- Adding a Message Processor
- Deleting a Message Processor
- Editing a Message Processor

#### **Activating and Deactivating a Message Processor**

You can activate or deactivate a message processor via the **Manage Message Processors** screen on the ESB Management Console.

#### **To activate or deactivate a message processor**

1. Click the **Main** tab on the Management Console, go to **Manage -> Service Bus** and then click **Message Processors**. The **Manage Message Processors** screen appears.
2. Click **Available Message Processors**. You will see all the message processors that are created.
3. In the **Actions** column of the required message processor, click either **Activate** or **Deactivate** based on your requirement. A confirmation message appears.
4. Click **Yes** to confirm activating or deactivating the required message processor.

#### **Adding a Message Processor**

WSO2 ESB allows you to add a required type of message processor to your ESB implementation via the **Manage Message Processors** screen of the ESB Management Console.

The types of message processors available with WSO2 ESB are as follows:

- Scheduled Message Forwarding Processor
- Scheduled Failover Message Forwarding Processor
- Message Sampling Processor
- Custom Message Processor

#### **To add a required message processor to your ESB implementation**

1. Click the **Main** tab on the Management Console, go to **Manage -> Service Bus** and then click **Message Processors**. The **Manage Message Processors** screen appears.
2. Click **Add Message Processors**. You will see the list of available message processors.
3. Click the required type of message processor you need to add.
4. Enter the required values to create the message processor, and then click **Save**.

#### **Note**

When adding a message processor, the parameters that you need to specify can vary depending on

the type of message processor that you add. For more information on each message processor and for descriptions of the parameters you need to specify when creating each type of message processor, see the relevant message processor page.

- Scheduled Message Forwarding Processor
- Scheduled Failover Message Forwarding Processor
- Message Sampling Processor
- Custom Message Processor

## Scheduled Message Forwarding Processor

The scheduled message forwarding processor is a [message processor](#) that consumes messages in a message store and sends them to an [endpoint](#). If a message is successfully delivered to the endpoint, the processor deletes the message from the message store. In case of a failure, it will retry after a specified interval.

Scheduled message forwarding processor parameters

Following are the parameters and the description for each of the parameters you need to specify when [creating a scheduled message forwarding processor](#).

Parameter Name	Description
Name	The name of the scheduled message forwarding processor.
Endpoint name	The endpoint to which the scheduled message forwarding processor forwards messages.
Message Store	The message store from which the scheduled message processor consumes messages.
Processor state ( <code>is.active</code> )	Activate ( <code>true</code> ) or Deactivate ( <code>false</code> )
Forwarding interval ( <code>interval</code> )	Interval in milliseconds in which processor consumes message. If both Cron Expression and Forwarding Interval are specified in configuration, Cron Expression will precede the Forwarding Interval.
Retry interval ( <code>client.retry.interval</code> )	Message retry interval in milliseconds.
Non retry http status codes ( <code>non.retry.status.codes</code> )	The parameter based on which the message processor decides to retry. If the HTTP status code of the response is specified as a status code, it will not retry.
Maximum redelivery attempts ( <code>max.delivery.attempts</code> )	Maximum redelivery attempts before deactivating the processor when the backend server is inactive and the ESB tries to resend the message.
Drop message after maximum delivery attempts ( <code>max.delivery.drop</code> )	If this parameter is set to <code>Enabled</code> , the message will be dropped from the message store after the maximum number of delivery attempts. The message processor will remain activated. This parameter will have effect when no value is specified for the <b>Maximum Delivery Attempts</b> parameter.  The <b>Maximum Delivery Attempts</b> parameter can be used when the processor is inactive and the message is resent.  If this parameter is disabled, the undeliverable message will not be dropped from the message store and the message processor will be deactivated.

Axis2 Client repository (axis2.repo)	The location path of the Axis2 Client repository. This repository it is needed to process messages prior to sending them to the endpoint.
Axis2 Configuration (axis2.config)	The location path of the Axis2 Configuration file to be used to process messages prior to sending them to the endpoint.
Reply sequence name (message.processor.reply.sequence)	The name of the sequence where the message reply should be sent.
Fault sequence name (message.processor.fault.sequence)	The name of the sequence where the fault message should be sent as a SOAP fault.
Deactivate sequence name (message.processor.deactivate.sequence)	The deactivate sequence that will be executed when the processor is deactivated automatically. Automatic deactivation occurs when delivery attempts is exceeded and the Drop message after max attempts parameter is disabled.
Quartz configuration file path (quartz.conf)	The Quartz configuration file path. This properties file contains configuration parameters for fine tuning the Quartz engine. More details of this can be found at <a href="http://quartz-scheduler.org/documentation/quartz-2.x/configMain">http://quartz-scheduler.org/documentation/quartz-2.x/configMain</a> .
Cron Expression (cronExpression)	Interval in milliseconds in which processor consumes message. If both Cron Expression and Forwarding Interval are specified in the configuration, Cron Expression will precede the Forwarding Interval.
Task Count (Cluster Mode)	The required number of worker nodes when you need to run the more than 1 worker node. Specifying this will not guarantee that the task will run on each worker node. There can be instances where the task will not run in some workers nodes.

In addition to specifying the required parameters, you can click **Show Additional Parameters** and set any of the additional parameters if necessary.

Message context properties to be used with the scheduled message forwarding processor

Property Name	Value	Required
targetEndpoint	Name of the Address Endpoint where the message should be delivered. This property is deprecated and is no longer required, but for backward compatibility, it does not cause errors if it is included.	Yes
OUT_ONLY	Set to <code>true</code> if this is an out-only message	Required for out-only scenarios
FORCE_ERROR_ON_SOAPFAULT	Set to <code>true</code> if it is required to retry in case of SOAP fault.	NO

#### Samples

For samples illustrating how to use the message forwarding processor, see:

- [Sample 702: Introduction to Message Forwarding Processor](#)
- [Sample 703: Adding Security to Message Forwarding Processor](#)
- [Sample 704: RESTful Invocations with Message Forwarding Processor](#)

- Sample 705: Load Balancing with Message Forwarding Processor

### Scheduled Failover Message Forwarding Processor

The scheduled failover message forwarding processor is a [message processor](#) that ensures reliable message delivery. This message processor is useful when it comes to scenarios where a message store failure takes place and it is necessary to ensure guaranteed message delivery.

The only difference of the scheduled failover message forwarding processor from the scheduled message forwarding processor is that the scheduled message forwarding processor forwards messages to a defined endpoint, whereas the scheduled failover message forwarding processor forwards messages to a target message store.

For an example scenario where the scheduled failover message forwarding processor is used, see [Guaranteed Delivery with Failover Message Store and Scheduled Failover Message Forwarding Processor](#).

Scheduled failover message forwarding processor parameters

Following are the parameters and the description for each of the parameters you need to specify when creating a scheduled failover message forwarding processor.

Parameter Name	Description
Name	The name of the scheduled failover message forwarding processor.
Source Message Store	The message store from which the scheduled failover message processor consumes messages.
Target Message Store	The message store to which the scheduled failover message processor forwards messages.
Processor state (is.active)	Activate (true) or Deactivate (false)
Forwarding interval (interval)	Interval in milliseconds in which processor consumes message
Retry interval (client.retry.interval)	Message retry interval in milliseconds.
Maximum delivery attempts (max.delivery.attempts)	Maximum redelivery attempts before deactivating the processor when the backend server is inactive and the ESB tries to resend message.
Drop message after maximum delivery attempts (max.delivery.drop)	If this parameter is set to Enabled, the message will be dropped from the message store after the maximum number of delivery attempts. If the message processor will remain activated. This parameter will have effect when no value is specified for the <b>Maximum Delivery Attempts</b> parameter.  The <b>Maximum Delivery Attempts</b> parameter can be used when the message store is inactive and the message is resent.  If this parameter is disabled, the undeliverable message will not be dropped from the message store and the message processor will be deactivated.

Fault sequence name (message.processor.fault.sequence)	The name of the sequence where the fault message should be of a SOAP fault.
Deactivate sequence name (message.processor.deactivate.sequence)	The deactivate sequence that will be executed when the process deactivated automatically. Automatic deactivation occurs when delivery attempts is exceeded and the Drop message after max attempts parameter is disabled.
Quartz configuration file path (quartz.conf)	The Quartz configuration file path. This properties file contains configuration parameters for fine tuning the Quartz engine. More details of this can be found at <a href="http://quartz-scheduler.org/documentation/quartz-2.x/configMain">http://quartz-scheduler.org/documentation/quartz-2.x/configMain</a> .
Cron Expression (cronExpression)	The cron expression to be used to configure the retry pattern.
Task Count (Cluster Mode)	The required number of worker nodes when you need to run the more than 1 worker node. Specifying this will not guarantee that will run on each worker node. There can be instances where the will not run in some workers nodes.

In addition to specifying the required parameters, you can click **Show Additional Parameters** and set any of the additional parameters if necessary.

### Message Sampling Processor

The message sampling processor consumes messages in a [message store](#) and sends them to a configured [sequence](#). This process happens in a preconfigured interval. This message processor does not ensure reliable messaging. Message sampling processor parameters

Following are the additional parameters you can set when adding a message sampling processor:

Processor State (is.active)	Activate (true) or Deactivate (false)	No. The default is Activate (true).
Sampling Interval (interval)	Interval in milliseconds in which processor consumes messages	No. The default value is 1000.
Sampling Concurrency (concurrency)	The number of messages the processor consumes at a time	No. The default value is 1.

Quartz Configuration File Path ( <code>quartz.conf</code> )	Quartz configuration file path. This properties file contains the Quartz configuration parameters for fine tuning the Quartz engine. More details of the configuration can be found at <a href="http://quartz-scheduler.org/documentation/quartz-2.x/configuration/ConfigMain">http://quartz-scheduler.org/documentation/quartz-2.x/configuration/ConfigMain</a> .	No
Cron Expression ( <code>cronExpression</code> )	Cron expression to be used to configure the retry pattern	No

#### Samples

For samples illustrating how to use the message sampling processor, see:

- [Sample 701: Introduction to Message Sampling Processor](#)

## Custom Message Processor

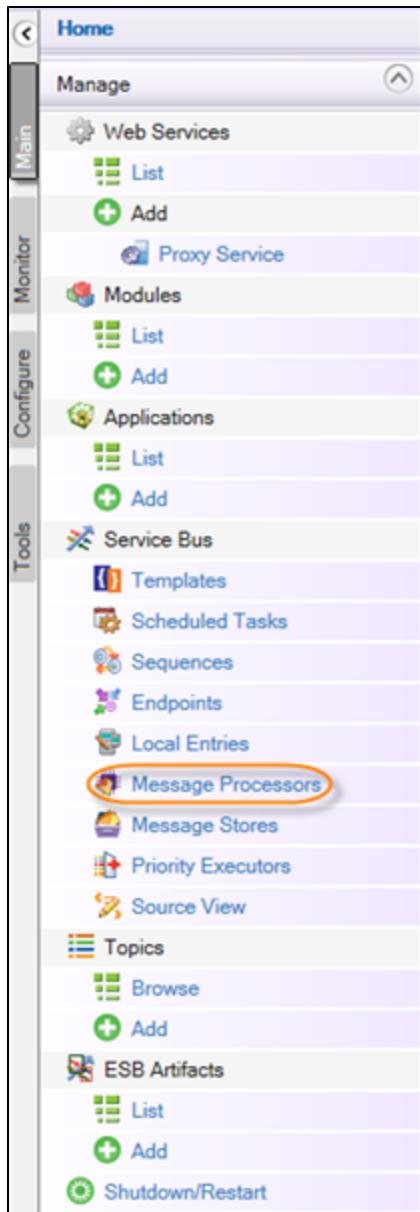
Existing message processor implementations are created using the `Quartz` enterprise job scheduler. If needed, you can create your own implementations of message processors by creating a Java class that implements the `MessageProcessor` interface. You then select the **Add Custom Message Processor** option when [adding a message processor](#) and specify your implementation class.

Note that message processors go through several life-cycle stages, so you must take great care when creating your own implementation. Because existing implementations are tested and proven under high loads, the best practice is to use the existing implementations whenever possible.

### ***Deleting a Message Processor***

Follow the instructions below to delete a [message processor](#).

1. On the Main tab of the ESB Management Console, click **Message Processors**.

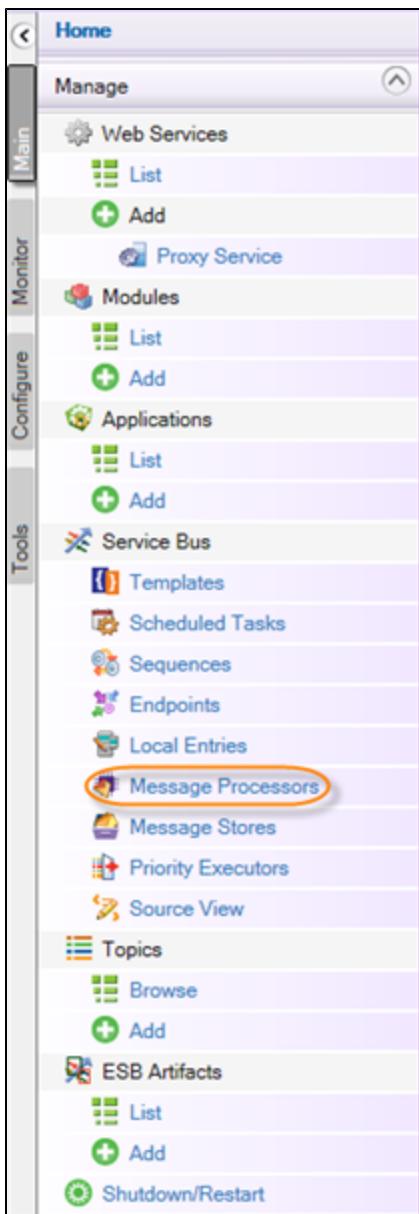


2. Click the **Delete** link next to the message processor you want to delete, and then confirm that you want to delete it by clicking **Yes**.

#### ***Editing a Message Processor***

Follow the instructions below to edit a message processor.

1. On the Main tab of the ESB Management Console, click **Message Processors** under Service Bus.



2. Click the **Edit** link next to the message processor you want to edit, make the necessary changes, and click **Save**.

#### Message Processing in a Worker-Manager Cluster Mode

This section explains the possible scenarios when a message processor is run in a worker-manager cluster node. In such situations, only workers are required to process/serve client requests while managers are mainly required to carry out administration activities and synchronize the administrative information of the workers in the cluster. A message processor may be executed by one or more worker nodes.

The following scenarios are possible when one or more message processors are run in a worker-manager cluster node.

##### ***Message processor execution by one node***

When a system administrator starts the WSO2 ESB in a worker-manager cluster mode with the WSO2 Elastic Load Balancer, only one worker can run a given message processor by default. No manager can run the message processor in this scenario.

### **Message processor execution by a given/selected set of workers**

In this scenario, a message processor can be simultaneously executed by more than one worker node in the cluster to optimise performance. The number of worker nodes that is allowed to execute the message processor at a given time can be controlled by specifying the required number in the message processor configuration as follows:

```
<parameter name="member.count">2</parameter>
```

All the configuration changes done to the message processor will take effect after the next ESB instance/cluster restart.

### **Shutting down a member with one or more message processors**

When a worker node on which one or more message processors are run is shut down, those message processors can be transferred to other worker nodes in the cluster.

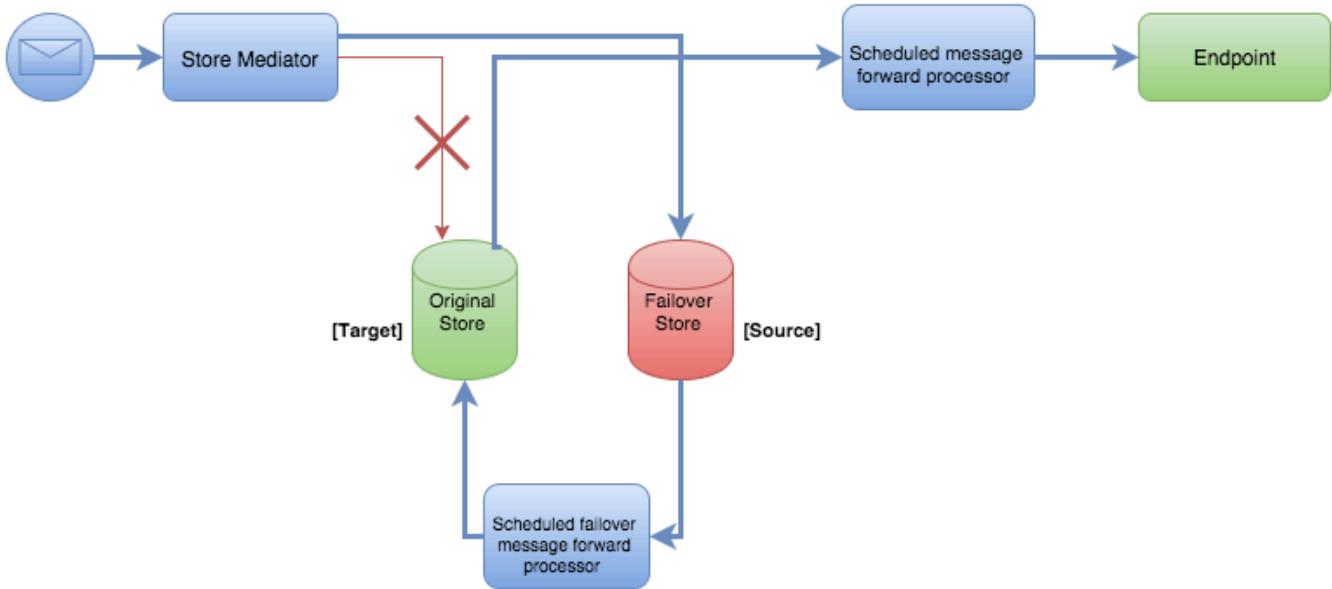
### **Guaranteed Delivery with Failover Message Store and Scheduled Failover Message Forwarding Processor**

WSO2 ESB ensures guaranteed delivery with the failover message store and scheduled failover message processor.

The topics in the following section describe how you can setup guaranteed message delivery with failover configurations.

### **Example Scenario**

The following diagram illustrates a scenario where a failover message store and scheduled failover message forwarding processor is used to ensure guaranteed delivery:



In this scenario, the original message store fails due to either network failure, message store crash or system shutdown for maintenance, and the failover message store is used as the solution for the original message store failure. So now the store mediator sends messages to the failover message store. Then, when the original message store is available again, the messages that were sent to the failover message store need to be forwarded to the original message store. The [scheduled failover message forwarding processor](#) is used for this purpose. The scheduled failover message forwarding processor is almost the same as the scheduled message

forwarding processor, the only difference is that the scheduled message forwarding processor forwards messages to a defined endpoint, whereas the scheduled failover message forwarding processor forwards messages to the original message store that the message was supposed to be temporarily stored.

### Setting up the example scenario

Create the failover message store | Create the original message store | Create the proxy service to send messages to the original message store via the store mediator | Define the endpoint for the scheduled message forwarding processor | Create a scheduled message forwarding processor to forward messages to the defined endpoint | Create a scheduled failover message forwarding processor | Send the request to the proxy service

Create the failover message store

In this example an in-memory message store is used to create the failover message store. If you have a cluster setup, it will not be possible to use an in-memory message store since it is not possible to share in-memory stores among nodes in a cluster. This step does not involve any special configuration.

```
<messageStore name="failover"/>
```

### Add In Memory Message Store

Create the original message store

In this example a JMS message store is used to create the original message store. When creating the original message store, you need to enable guaranteed delivery on the producer side. To do this, set the following parameters in the message store configuration:

```
<parameter name="store.failover.message.store.name">failover</parameter>
<parameter name="store.producer.guaranteed.delivery.enable">true</parameter>
```

Following is the message store configuration used in this example:

```
<messageStore
 class="org.apache.synapse.message.store.impl.jms.JmsStore" name="Orginal">
 <parameter name="store.failover.message.store.name">failover</parameter>
 <parameter name="store.producer.guaranteed.delivery.enable">true</parameter>
 <parameter
 name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
 <parameter name="store.jms.JMSSpecVersion">1.1</parameter>
 </messageStore>
```

## Add JMS Message Store

Add JMS Message Store [Switch to source view](#)

Name *	Original
Initial Context Factory *	<code>org.apache.activemq.jndi.ActiveMQInitialContextFactory</code>
Provider URL *	<code>tcp://localhost:61616</code>
<input checked="" type="checkbox"/> Show Additional Parameters <input type="checkbox"/> Hide Guaranteed Delivery Parameters	
<b>Failover Configuration Parameters</b>	
Enable Producer Guaranteed Delivery	<input checked="" type="checkbox"/> True
Failover Message Store	<input type="button" value="failover"/>
<input type="button" value="Save"/> <input type="button" value="Cancel"/>	

Create the proxy service to send messages to the original message store via the store mediator

Following is the proxy service used in this example:

```
<proxy name="Proxyl" transports="https http" startOnLoad="true" trace="disable">
 <target>
 <inSequence>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
 <property name="OUT_ONLY" value="true"/>
 <log level="full"/>
 <store messageStore="Orginal"/>
 </inSequence>
 </target>
</proxy>
```

Define the endpoint for the scheduled message forwarding processor

In this example, the SimpleStockquote service is used as the back-end service.

```
<endpoint name="SimpleStockQuoteService">
 <address uri="http://127.0.0.1:9000/services/SimpleStockQuoteService"/>
</endpoint>
```

Create a scheduled message forwarding processor to forward messages to the defined endpoint

Following is the scheduled message forwarding processor configuration used in this example:

```

<messageProcessor

 class="org.apache.synapse.message.processor.impl.forwarder.ScheduledMessageForwardingProcessor"
 messageStore="Orginal" name="ForwardMessageProcessor"
 targetEndpoint="SimpleStockQuoteService">
 <parameter name="client.retry.interval">1000</parameter>
 <parameter name="throttle">false</parameter>
 <parameter name="max.delivery.attempts">4</parameter>
 <parameter name="member.count">1</parameter>
 <parameter name="max.delivery.drop">Disabled</parameter>
 <parameter name="interval">1000</parameter>
 <parameter name="is.active">true</parameter>
 <parameter name="target.endpoint">SimpleStockQuoteService</parameter>
</messageProcessor>

```

For more information, see [Scheduled Message Forwarding Processor](#).

Create a scheduled failover message forwarding processor

When creating the scheduled failover message forwarding processor, you need to specify the following two mandatory parameters that are important in the failover scenario.

- **Source Message Store**
- **Target Message Store**

The scheduled failover message forwarding processor sends messages from the failover store to the original store when it is available in the failover scenario. In this configuration, the source message store should be the failover message store and target message store should be the original message store.

Following is the scheduled failover message forwarding processor configuration used in this example:

```

<messageProcessor

 class="org.apache.synapse.message.processor.impl.failover.FailoverScheduledMessageForwardingProcessor"
 messageStore="failover" name="FailoverMessageProcessor">
 <parameter name="client.retry.interval">60000</parameter>
 <parameter name="throttle">false</parameter>
 <parameter name="max.delivery.attempts">1000</parameter>
 <parameter name="member.count">1</parameter>
 <parameter name="max.delivery.drop">Disabled</parameter>
 <parameter name="interval">1000</parameter>
 <parameter name="is.active">true</parameter>
 <parameter name="message.target.store.name">Orginal</parameter>
</messageProcessor>

```

## Add Scheduled Failover Message Forwarding Processor

Add Scheduled Failover Message Forwarding Processor [Switch to source view](#)

Name *	<input type="text" value="FailoverMessageProcessor"/>
Source Message Store *	<input type="button" value="failover"/>
Target Message Store *	<input type="button" value="Original"/>
<a href="#">Hide Additional Parameters</a>	
Message Forwarding Processor Parameters	
Processor state *	<input type="button" value="Activate"/>
Forwarding interval (Millis)	<input type="text" value="1000"/>
Retry Interval (Millis)	<input type="text" value="60000"/>
Maximum delivery attempts	<input type="text" value="1000"/>
Drop message after maximum delivery attempts	<input type="button" value="Disabled"/>
Fault sequence name	<input type="text"/> <a href="#">Configuration Registry</a> <a href="#">Governance Registry</a>
Deactivate sequence name	<input type="text"/> <a href="#">Configuration Registry</a> <a href="#">Governance Registry</a>
Quartz configuration file path	<input type="text"/>
Cron Expression	<input type="text"/>
Task Count (Cluster Mode)	<input type="text" value="1"/>

Send the request to the proxy service

Navigate to the <ESB\_HOMES>/samples/axis2client directory, and execute the following command to invoke the proxy service:

```
ant stockquote -Daddurl=http://localhost:8280/services/Proxy1 -Dmode=placeorder
```

You will see the following message on the Axis2 server console:

```
SimpleStockQuoteService :: Accepted order for : 7482 stocks of IBM at $ 169.27205579038733
```

### Testing the example scenario

To test the failover scenario, shut down the JMS broker(i.e., the original message store) and send a few messages to the proxy service.

You will see that the messages are not sent to the back-end since the original message store is not available. You will also see that the messages are stored in the failover message store.

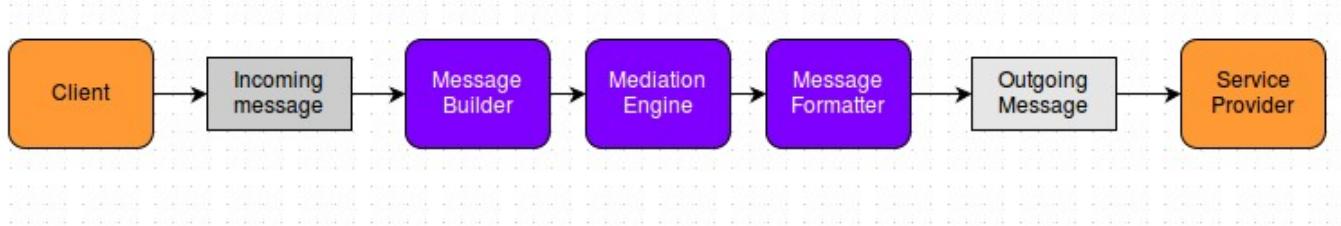
If you analyze the ESB log, you will see the failover message processor trying to forward messages to the original message store periodically. Once the original message store is available, you will see that the scheduled failover message forwarding processor sends the messages to the original store and then that the scheduled message forwarding processor forwards the messages to the back-end service.

## Working with Message Builders and Formatters

- Overview
- Configuring message builders and formatters
- Using message relay
- Handling messages with no content type
- Handling text/csv messages

### Overview

When a message comes in to the ESB, the receiving transport selects a **message builder** based on the message's content type. It uses that builder to process the message's raw payload data and convert it into SOAP. Conversely, when sending a message out from the ESB, a **message formatter** is used to build the outgoing stream from the message. As with message builders, the message formatter is selected based on the message's content type. In a typical ESB routing scenario, here is the flow:



You can use the [messageType](#) property to change the message's content type as it flows through the ESB. For example, if the incoming message is in JSON format and you want to transform it to XML, you could add the `messageType` property before your mediators in the configuration:

```
<property name="messageType" value="application/xml" scope="axis2" />
```

### Configuring message builders and formatters

Message builders and formatters are specified in `<ESB_HOME>/repository/conf/axis2/axis2.xml` under the `messageBuilders` and `messageFormatters` configuration sections. The ESB has a few default message builders, so even if you do not specify them explicitly in `axis2.xml`, they will take effect when messages of those content types come into the ESB. If you want to use different builders, specify them in `axis2.xml` to override the defaults. The ESB does not have default message formatters, so it is important to specify all of them in the `axis2.xml` configuration. Following are the default message builders:

Content type	Message Builder
application/soap+xml	org.apache.axis2.builder.SOAPBuilder
multipart/related	org.apache.axis2.builder.MIMEBuilder
text/xml	org.apache.axis2.builder.SOAPBuilder
application/xop+xml	org.apache.axis2.builder.MTOMBuilder
application/xml	org.apache.axis2.builder.ApplicationXMLBuilder
application/x-www-form-urlencoded	org.apache.axis2.builder.XFormURLEncodedBuilder

### Using message relay

If you want to enable message relay, so that messages of a specific content type are not built or formatted but simply pass through the ESB, you can specify the message relay builder (`org.wso2.carbon.relay.BinaryRel`

`ayBuilder`) for that content type. For more information, see [Configuring Message Relay](#).

#### **Handling messages with no content type**

To ensure that messages with no content type are handled gracefully without causing errors, add the following to `axis2.xml`:

- In the parameters section: `<parameter name="defaultContentType" locked="false">empty/content</parameter>`
- In the message builders section: `<messageBuilder contentType="empty/content" class="org.wso2.carbon.relay.BinaryRelayBuilder"/>`
- In the message formatters section: `<messageFormatter contentType="empty/content" class="org.wso2.carbon.relay.ExpandingMessageFormatter"/>`

#### **Handling text/csv messages**

There is no default builder or formatter for messages with the text/csv content type. If you just want to pass these messages through the ESB, you can configure the message relay builder and formatter. If you want to process these messages, you can access the content inside the request/response payload of CSV by configuring the `org.apache.axis2.format.PlainTextBuilder` and `org.apache.axis2.format.PlainTextFormatter` for the text/csv content type in `axis2.xml`. For example:

```
<messagebuilder contenttype="text/csv"
class="org.apache.axis2.format.PlainTextBuilder"/>
<messageformatter contenttype="text/csv"
class="org.apache.axis2.format.PlainTextFormatter"/>
```

When a text/csv message comes into the ESB, the log will include an entry similar to the following, and you can observe that the CSV data is placed inside the payload:

```
[2013-05-09 13:59:03,478] INFO - LogMediator To: , WSAction: urn:mediate, SOAPAction: urn:mediate, MessageID: urn:uuid:5B9A211341DCC368241368088143463, Direction: request, Envelope: <?xml version='1.0' encoding='utf-8'?><soapenv:envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:body><text xmlns="http://ws.apache.org/commons/ns/payload">charitha,wso2,colombo chamara,wso2G,galle </text></soapenv:body></soapenv:envelope>
```

## **Message Relay**

**Message Relay** enables WSO2 ESB to pass messages along without building or processing them unless specifically requested to do so. When Message Relay is enabled, an incoming message is wrapped inside a default SOAP envelope as binary content and sent through the ESB. This is useful for scenarios where the ESB does not need to work on the full message but can work on [message properties](#) like request URLs or transport headers instead. With Message Relay, the ESB can achieve a very high throughput.

The following topics provide more information about Message Relay:

- [Working with Message Builders and Formatters](#)
- [Configuring Message Relay](#)
- [Builder Mediator](#)
- [Message Relay Module](#)
- [Message Relay Module Policy](#)

See also [HTTP PassThrough Transport](#).

#### [Configuring Message Relay](#)

In the `axis2.xml` file, there are two configuration sections called `messageBuilders` and `messageFormatters`. The user can replace the expected content types with the Message Relay builder and formatter to pass these messages through the ESB without building them.

Message Relay Builder and Formatter Class Names

<b>Builder</b>	org.wso2.carbon.relay.BinaryRelayBuilder
<b>Formatter</b>	org.wso2.carbon.relay.ExpandingMessageFormatter

Sample Configuration for Message Builder

```
<messageBuilders>
 <messageBuilder contentType="application/xml"
 class="org.wso2.carbon.relay.BinaryRelayBuilder"/>
 <messageBuilder contentType="application/x-www-form-urlencoded"
 class="org.wso2.carbon.relay.BinaryRelayBuilder"/>
 <messageBuilder contentType="multipart/form-data"
 class="org.wso2.carbon.relay.BinaryRelayBuilder"/>
 <messageBuilder contentType="text/plain"
 class="org.wso2.carbon.relay.BinaryRelayBuilder"/>
 <messageBuilder contentType="text/xml"
 class="org.wso2.carbon.relay.BinaryRelayBuilder"/>
</messageBuilders>
```

Sample Configuration for Message Formatter

```
<messageFormatters>
 <messageFormatter contentType="application/x-www-form-urlencoded"
 class="org.wso2.carbon.relay.ExpandingMessageFormatter"/>
 <messageFormatter contentType="multipart/form-data"
 class="org.wso2.carbon.relay.ExpandingMessageFormatter"/>
 <messageFormatter contentType="application/xml"
 class="org.wso2.carbon.relay.ExpandingMessageFormatter"/>
 <messageFormatter contentType="text/xml"
 class="org.wso2.carbon.relay.ExpandingMessageFormatter"/>
 <messageFormatter contentType="text/plain"
 class="org.wso2.carbon.relay.ExpandingMessageFormatter"/>
 <!--<messageFormatter contentType="text/plain"
 class="org.apache.axis2.format.PlainTextBuilder"/>-->
 <messageFormatter contentType="application/soap+xml"
 class="org.wso2.carbon.relay.ExpandingMessageFormatter"/>
</messageFormatters>
```

### Example

If you want the ESB to receive messages of the `image/png` content type, add the following configurations to the `<ESB_HOME>/repository/conf/axis2/axis2.xml` file.

In the Message Builders section:

```
<messageBuilder contentType="image/png"
 class="org.wso2.carbon.relay.BinaryRelayBuilder"/>
```

In the Message Formatters section:

```
<messageFormatter contentType="image/png"
 class="org.wso2.carbon.relay.ExpandingMessageFormatter"/>
```

## Builder Mediator

The **Builder Mediator** can be used to build the actual SOAP message from a message coming in to ESB through the Binary Relay. One usage is to use this before trying to log the actual message in case of an error. Also with the Builder Mediator ESB can be configured to build some of the messages while passing the others along.

In order to use the Builder mediator, `BinaryRelayBuilder` should be specified as the message builder in the `<ESB_HOME>/repository/conf/axis2/axis2.xml` file for at least one content type. The message formatter specified for the same content types should be `ExpandingMessageFormatter`. Unlike other message builders defined in `axis2.xml`, the `BinaryRelayBuilder` works by passing through a binary stream of the received content. The Builder mediator is used in conjunction with the `BinaryRelayBuilder` when we require to build the binary stream into a particular content type during mediation. We can specify which message builder should be used to build the binary stream using the Builder mediator.

By default, Builder Mediator uses the `axis2` default Message builders for the content types. User can override those by using the optional `messageBuilder` configuration. For more information, see [Working with Message Builders and Formatters](#).

Like in `axis2.xml` user has to specify the content type and the implementation class of the `messageBuilder`. Also user can specify the message formatter for this content type. This is used by the `ExpandingMessageFormatter` to format the message before sending to the destination.

## Syntax

```
<syn:builder xmlns:syn="http://ws.apache.org/ns/synapse">
 <syn:messageBuilder contentType="" class="" [formatterClass=""]/>
</syn:builder>
```

## Message Relay Module

Message Relay has an `axis2` module as well. This is an optional feature. This module can be used to build the actual SOAP message from the messages that went through the Message Relay. See [Working with Modules](#).

To enable this module, the user has to enable the relay module globally in the `axis2.xml`.

```
<module ref="relay" />
```

Also user has to put the following phase into the InFlow of `axis2`.

```
<phase name="BuildingPhase" />
```

This module is designed to be used by Admin Services that runs inside the ESB. All the admin services are running with content type: `application/soap+xml`. So if a user wants to use admin console and use the ESB for receiving messages with content type `application/soap+xml`, this module should be used.

User can configure the module by going to the modules section in admin console and then going to the relay

module. The module configuration is specified as a module [policy](#).

## Note

After changing the policy user has to restart the ESB for changes to take effect.

### Message Relay Module Policy

Syntax of Relay Module Policy.

```
<wsp:Policy wsu:Id="MessageRelayPolicy"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
 xmlns:wsmr="http://www.wso2.org/ns/2010/01/carbon/message-relay"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd">
 <wsmr:RelayAssertion>
 <wsp:Policy>
 <wsp:All>
 <wsmr:includeHiddenServices>false | false</wsmr:includeHiddenServices>
 <wsmr:services>
 <wsmr:service>Name of the service</wsmr:service>*
 </wsmr:services>
 <wsmr:builders>
 <wsmr:messageBuilder contentType="content type of the message"
class="message builder implementation class" class="message formatter implementation
class"/>
 </wsmr:builders>
 </wsp:All>
 </wsp:Policy>
 </wsmr:RelayAssertion>
</wsp:Policy>
```

These are the assertions:

- **includeHiddenServices** - If this is true message going to the services with `hiddenService` parameter will be built.
- **wsmr:services** - Messages going to these services will be built.
- **wsmr:service** - Name of the service.
- **wsmr:builders** - Message builders to be used for building the message.
- **wsmr:builder** - A message builder to be used for a content type.

## Note

After changing the policy, user has to restart the ESB for changes to take effect.

## Note

If the Message Relay is enabled for particular content type, there cannot be services with security enabled for that content type.

## Prioritizing Messages

You can prioritize messages to ensure that high-priority messages are not dropped. Prioritization is implemented at two levels in WSO2 ESB:

- **HTTP transport level** - If users would like to use the ESB as a pure router.
- **Message mediation level** - If users use ESB for heavy processing like XSLT and XQuery.

From the users perspective, key to any priority mediation is to determine the priority of an incoming message.

At the message mediation layer, this can be done using content filters. This means the full power of ESB configuration language is available to the user for determining the priority of a given message. For example, a message may contain an element called "priority" and depending on its value the priority can be determined.

At the HTTP layer, user has access to HTTP headers, HTTP parameters and URL values. By looking at these values, user can determine the priority of a given message.

The priority mediation implementation is based on `Queues` and `ThreadPoolExecutors`.

`ThreadPoolExecutor` accepts a `BlockingQueue` implementation. A custom blocking queue that can be used to order the jobs based on priority was implemented. `ThreadPoolExecutor` starts queuing only when all the core threads are busy. Every message should get equal priority until all the core threads are used.

Internally custom `BlockingQueue` uses multiple queues for accepting jobs with different priorities. Once jobs are put into the queue, it uses a pluggable algorithm for choosing the next job. The default algorithm chooses the jobs based on a priority-based, round-robin algorithm. For example, let's say we have two priorities, 10 and 1. This algorithm tries to fetch 10 items with priority 10 and then 1 item with the priority 1.

### Priority Executors

**Priority executors** can be used with the [Enqueue Mediator](#) to execute sequences with a given priority. Priority executors are used in high-load scenarios when you want to execute different sequences for messages with different priorities. This approach allows you to control the resources allocated to executing sequences and to prevent high-priority messages from getting delayed and dropped. A priority has a valid meaning comparing to other priorities specified. For example, if there are two priorities with value 10 and 1, a message with priority 10 will get 10 times more resources than messages with priority 1.

### Priority Executor Configuration

```
<priority-executor name="priority-executor">
 <queues isFixed="true|false" nextQueue="class implementing NextQueueAlgorithm">
 <queue [size="size of the queue"] priority="priority of the messages put in to this queue"/>*
 </queues>
 <threads core="core number of threads" max="max number of threads" keep-alive="keep alive time"/>
</priority-executor>
```

Core priority executors' attributes:

- **queues** - Defines separate queues for different priorities in a Thread Pool Executor.
- **isFixed** - Controls the queues to have a fixed depths or un-bounded capacities.

### Note

An executor should have at least two or more queues. If only one queue is used, there is no point in using a priority executor.

- **size** - Defines a size of a queue.
- **priority** -Defines a priority of a queue.

## Note

If the queues has unlimited length, no more than core threads will be used.

- **core** - Defines a core number of Priority Executor threads. When ESB is started with the priority executor, this number of threads will be created.
- **max** - Defines the maximum number of threads this executor will have.
- **keep-alive** - Defines the keep-alive time of threads. If the number of threads in the executor exceeds the core threads, they will be in active for the keep-alive time only. After the keep-alive time, those threads will be removed from the system.

The following topics describe how to manage your priority executors:

- Adding a Priority Executor
- Editing a Priority Executor
- Deleting a Priority Executor

To see a sample of priority-based mediation, see [Sample 652: Priority Based Message Mediation](#).

## Adding a Priority Executor

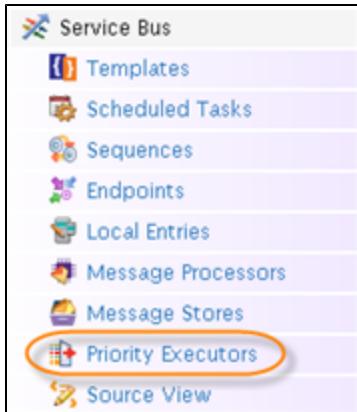
**Priority executors** can be used to execute sequences with a given priority. This allows the user to control the resources allocated to executing sequences and prevent high priority messages from getting delayed and dropped.

Follow the instructions below to add a new priority executor to the WSO2 ESB.

1. Sign in. Enter your user name and password to log on to the ESB Management Console.
2. Click on "Main" in the left menu to access the "Manage" menu.



3. In the "Manage" menu, click on "Priority Executors" under "Service Bus."



4. In the "Priority Executors" window, click on the "Add Executor" link.

The screenshot shows a table titled 'Priority Executors' with one row containing 'Executor\_1'. It includes 'Actions' for Edit and Delete, and a 'Add Executor' button at the bottom left, which is circled in red.

5. Specify the options of a new priority executor in the "Priority Executor Design" widow.

- **Executor Name**- Name of the Executor.
- **Fixed Size Queues** - Whether fixed size queues are used or not.
- **Queues** - Individual Queue Configurations. See the detailed information below.
- **Next Queue Algorithm**
- **Max** - Maximum Number of Threads in the Executor.
- **Core** - Core Number of Threads in the Executor.
- **Keep-Alive** - Keep Alive time for Threads.

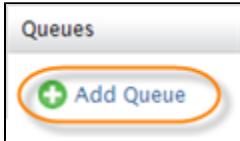
The screenshot shows the 'Priority Executor Design' window with the following fields:

- Priority Executor Design View** (button) and **Switch to Source View** (button).
- Executor Name\***: A text input field.
- Fixed Size Queues\***: A dropdown menu set to **true**.
- Queues** section: A table with columns **Priority**, **Size**, and **Action**. It includes a **+ Add Queue** button.
- Next Queue Algoritm** section: A table with column **Class**.
- Threads** section: Fields for **Core** (20), **Max** (100), and **Keep-Alive** (5).
- Save** and **Cancel** buttons at the bottom.

**Tip**

An executor should have at least two or more queues. If only one queue is used, there is no point in using a priority executor.

6. Each and every priority has a queue associated with it. A message is put in to one of the queues corresponding to its priority. To add a queue to an executor, click on the "Add Queue" link.



7. Specify the "Queues" options:

- **Priority** - Priority of the Queue.
- **Size** - Size of the Queue. This option is visible only if fixed size queues are selected.

Queues		
Priority	Size	Action
10	5	Delete

**Tip**

You can remove a queue from an executor clicking on the "Delete" link in the actions column.

Queues		
Priority	Size	Action
10	5	Delete

8. Click on the "Save" button to add an executor to the list.



9. A new priority executor appears in the list.

Priority Executors		
Name	Actions	
Executor_2	Edit	Delete
Executor_1	Edit	Delete

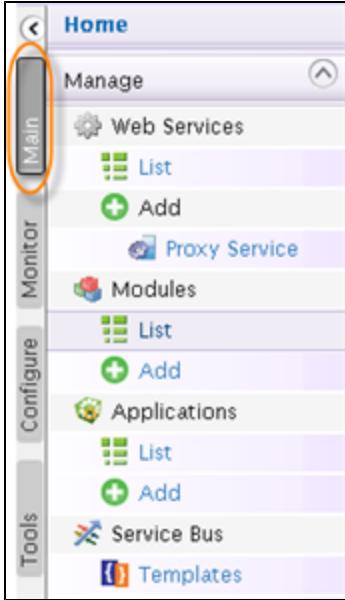
**Editing a Priority Executor**

**Priority executors** can be used to execute sequences with a given priority. This allows user to control the

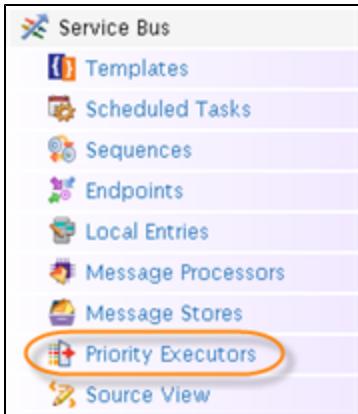
resources allocated to executing sequences and prevent high priority messages from getting delayed and dropped. Once adding a priority executor, you can edit it options according to new requirements.

Follow the instructions below to edit a priority executor in the WSO2 ESB.

1. Sign in. Enter your user name and password to log on to the ESB Management Console.
2. Click on "Main" in the left menu to access the "Manage" menu.



3. In the "Manage" menu, click on "Priority Executors" under "Service Bus."



4. In the "Priority Executors" window, click on the "Edit" link in the "Actions" column.

Priority Executors	
Name	Actions
Executor_2	Edit  Delete
Executor_1	Edit  Delete
Add Executor	

5. The "Priority Executor Design" window appears.

## Priority Executor Design

Priority Executor Design View [Switch to Source View](#)

Executor Name*	<input type="text" value="Executor_2"/>	
Fixed Size Queues*	<input type="checkbox"/> true	
<b>Queues</b>		
Priority	Size	Action
5	<input type="text" value="5"/>	Delete
4	<input type="text" value="5"/>	Delete
<a href="#"> Add Queue</a>		
<b>Next Queue Algorithm</b>		
Class	<input type="text"/>	
<b>Threads</b>		
Core	<input type="text" value="20"/>	
Max	<input type="text" value="100"/>	
Keep-Alive	<input type="text" value="5"/>	
<input type="button" value="Save"/> <input type="button" value="Cancel"/>		

6. Edit the options of the priority executor. For the detailed information about the executors options see [Adding a Priority Executor](#).

7. Click on the "Save" button.

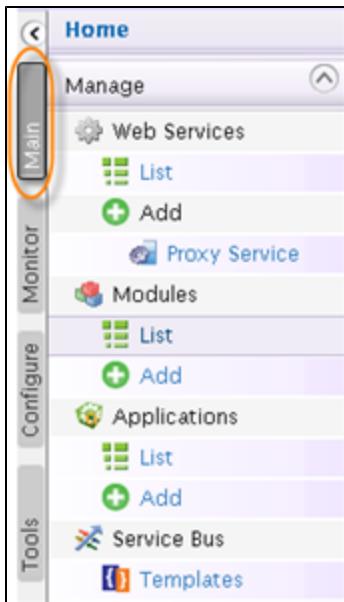


### Deleting a Priority Executor

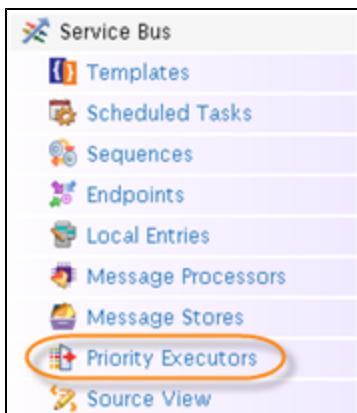
**Priority executors** can be used to execute sequences with a given priority. This allows user to control the resources allocated to executing sequences and prevent high priority messages from getting delayed and dropped. If there is no need to use the existing priority order, you can delete a priority executor in WSO2 ESB.

Follow the instructions below to delete a priority executor from the WSO2 ESB.

1. Sign in. Enter your user name and password to log on to the ESB Management Console.
2. Click on "Main" in the left menu to access the "Manage" menu.



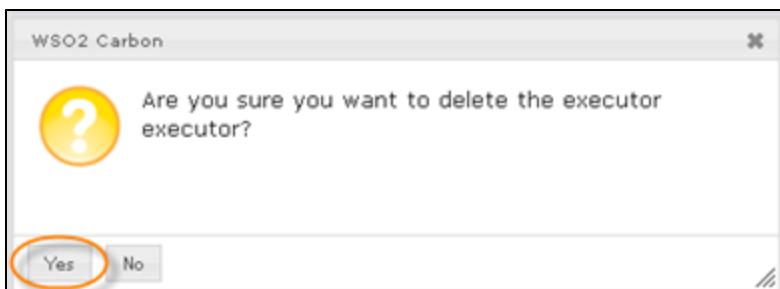
3. In the "Manage" menu, click on "Priority Executors" under "Service Bus."



4. In the "Priority Executors" window, click on the "Delete" link in the "Actions" column.

Priority Executors	
Name	Actions
Executor_2	Edit  Delete (highlighted with a red circle)
Executor_1	Edit  Delete
Add Executor	

5. Confirm your request in the "WSO2 Carbon" window.



## Transactions

A **transaction** is a set of operations executed as a single unit. It also can be defined as an agreement, which is carried out between separate entities or objects. A transaction can be considered as indivisible or atomic when it has the characteristic of either being completed in its entirety or not at all. During the event of a failure for a transaction update, atomic transaction type guarantees transaction integrity such that any partial updates are rolled back automatically.

Transactions have many different forms, such as financial transactions, database transactions etc.

From the ESB point of view, there are two types of transactions:

- Distributed transactions
- Java Message Service (JMS) transactions

### **Distributed transactions**

A **distributed transaction** is a transaction that updates data on two or more networked computer systems, such as two databases or a database and a message queue such as JMS. Implementing robust distributed applications is difficult because these applications are subject to multiple failures, including failure of the client, the server, and the network connection between the client and server. For distributed transactions, each computer has a local transaction manager. When a transaction works at multiple computers, the transaction managers interact with other transaction managers via either a superior or subordinate relationship. These relationships are relevant only for a particular transaction.

For an example that demonstrates how the [transaction mediator](#) can be used to manage distributed transactions, see [Transaction Mediator Example](#).

### **Java Message Service (JMS) transactions**

In addition to distributed transactions, WSO2 ESB also supports JMS transactions.

For more information on JMS transactions, see [JMS Transactions](#).

## Debugging Mediation

Message mediation mode is one of the operational modes of WSO2 ESB where ESB functions as an intermediate message router. When operating in this mode, it can filter, transform, drop or forward messages to an endpoint based on the given parameters. A unit of the mediation flow is a mediator. Sequences define the message mediation behavior of the ESB. A sequence is a series of mediators, where each mediator is a unit entity that can input a message, carry out a predefined processing task on the message, and output the message for further processing.

- What is debugging with respect to mediation
- Creating the ESB artifact
- Enabling mediation debugging
- Information provided by the Debugger Tool
- Changing the property values
- Viewing wire logs

### **What is debugging with respect to mediation**

Debugging is where you want to know if these units, which function as separate entities are operating as intended, or if a combination of these units are operating as a whole as intended. WSO2 ESB packs the ESB Mediation debugger that enables you to debug the ESB message mediation flow in the server. Tooling support for the ESB Mediation debugger is provided by the WSO2 ESB Tooling Plugin.

### **Creating the ESB artifact**

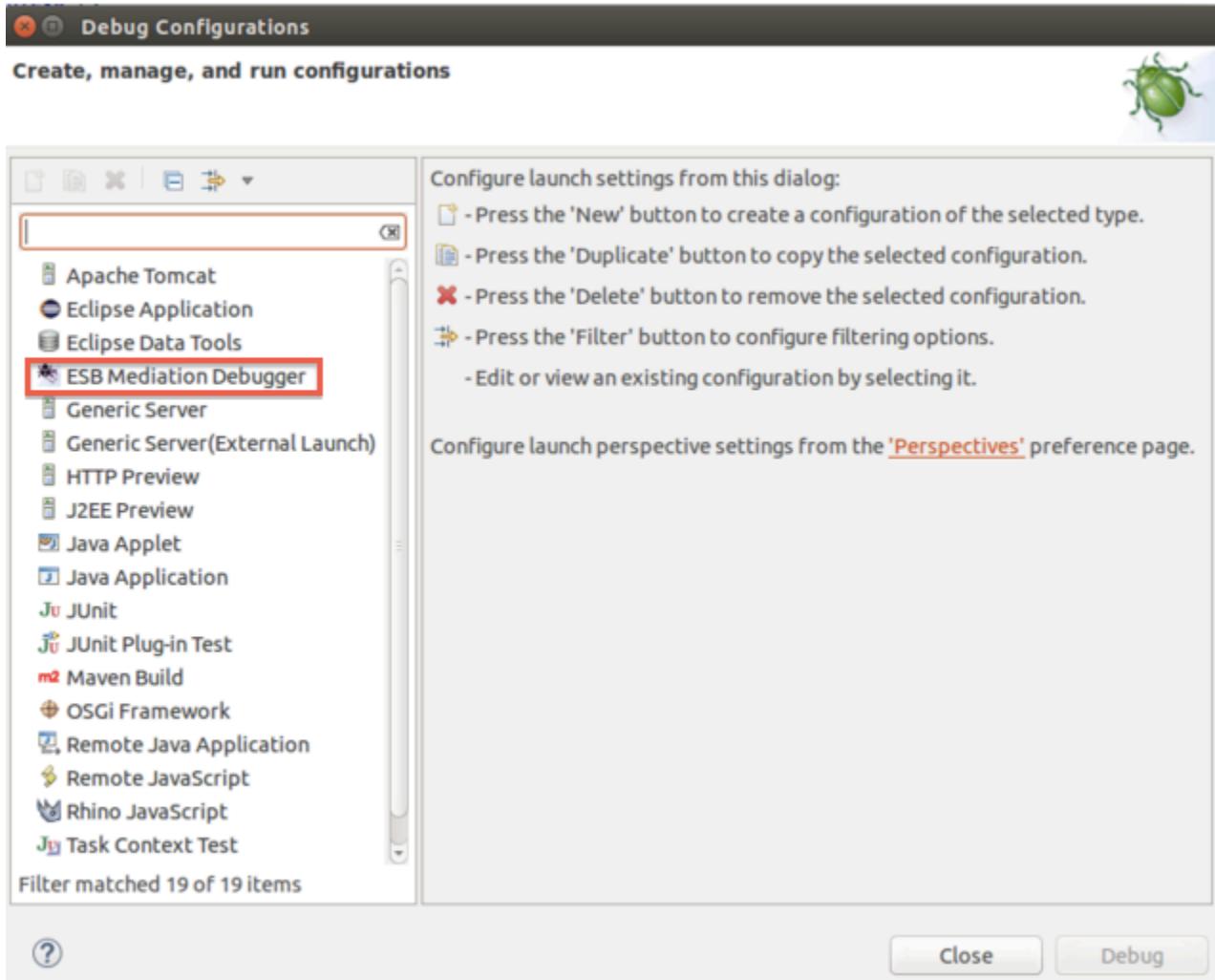
Follow the steps below to create a sample ESB artifact, to which you apply mediation.

1. Install the WSO2 ESB Tooling Plugin and run it. For instructions, see [Installing the ESB Tooling Plug-In](#).
2. Create the ESB artifact using the WSO2 ESB Tooling Plugin. For instructions, see [Creating ESB Artifacts](#).
3. Deploy the artifact you created on WSO2 ESB. For instructions, go to [Deploying the ESB Config project](#).

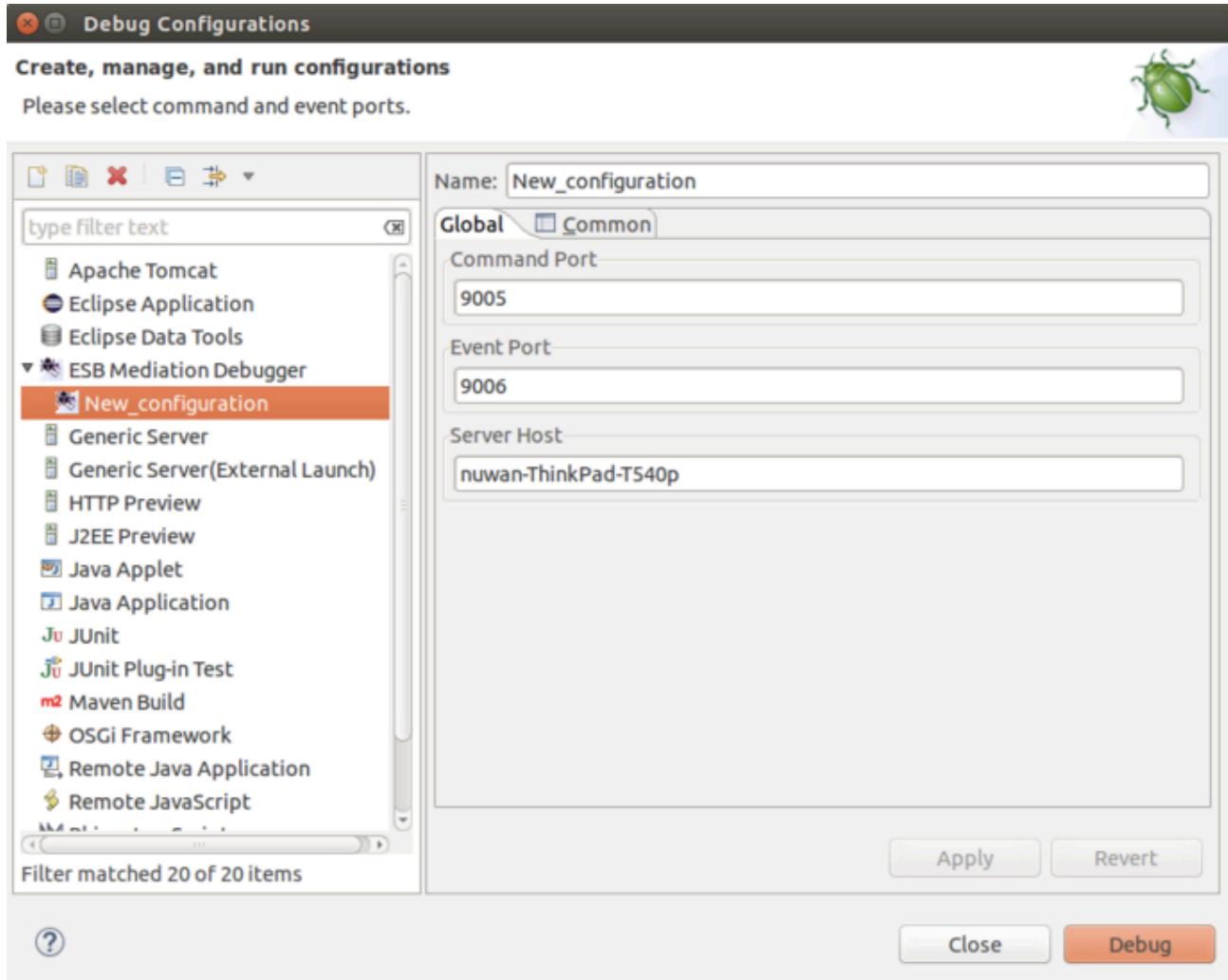
#### ***Enabling mediation debugging***

Follow the steps below to enable debugging with respect to mediation.

1. Click **Run** in the top menu of the WSO2 ESB Tooling Plugin, and then click **Debug Configurations**.
2. Double click **ESB Mediation Debugger** as shown below.



3. Enter the details to create a new configuration as shown in the example below. You need to define two port numbers and a hostname to connect the ESB server with the WSO2 ESB Tooling Plugin in the mediation debug mode.

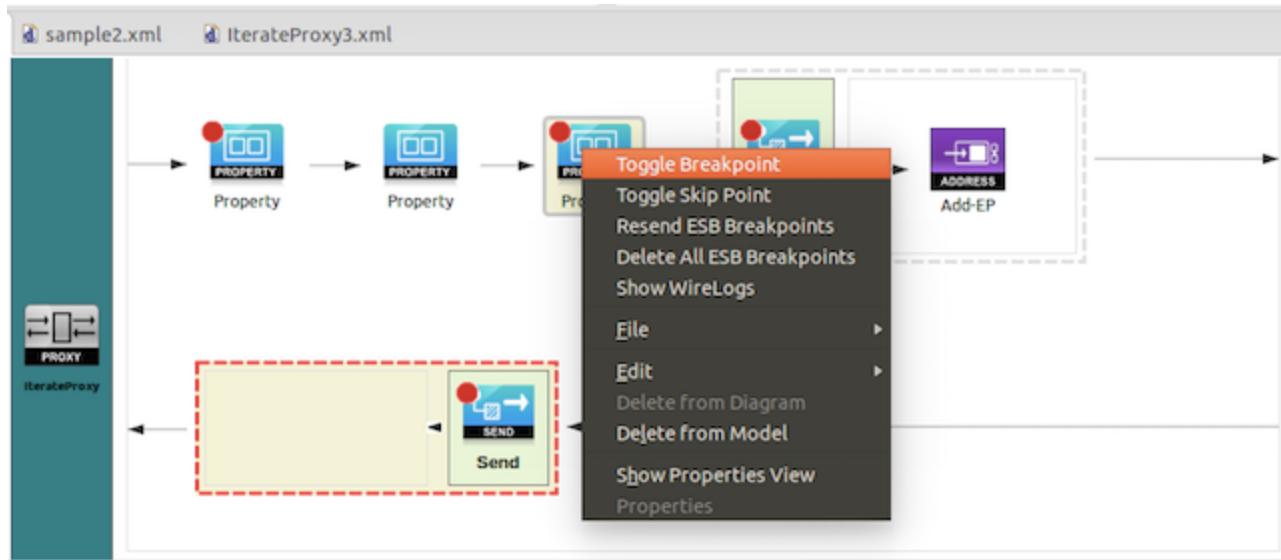


4. Execute the following command to start WSO2 ESB server in the debug mode by passing a system variable at start up: sh wso2server.sh -Desb.debug=true
5. Click **Debug** in the WSO2 ESB Tooling Plugin when the WSO2 ESB Console indicates the following.

You have approximately a one-minute time span to connect the WSO2 ESB Tooling Plugin with the ESB server for the execution of the above created debug configuration. Otherwise, the ESB server will stop listening and start without connecting with the debugger tool.

```
2016-01-22 12:12:01,501] INFO - DeploymentEngine Deploying Web service: org.wso2.carbon.tryit-4.4.8 -
2016-01-22 12:12:01,714] INFO - CarbonServerManager Repository : /home/nuwan/Documents/WSO2Servers/ESB/wso2esb-4.10.0-M3-SNAPSHOT/repository/deployment/server/
2016-01-22 12:12:01,776] INFO - TenantLoadingConfig Using tenant lazy loading policy...
2016-01-22 12:12:01,785] INFO - PermissionUpdater Permission cache updated for tenant -1234
2016-01-22 12:12:01,841] INFO - RuleEngineConfigDS Successfully registered the Rule Config service
2016-01-22 12:12:01,898] INFO - ServiceBusInitializer Starting ESB...
2016-01-22 12:12:01,928] INFO - ServiceBusInitializer Initializing Apache Synapse...
2016-01-22 12:12:01,935] INFO - ServiceBusInitializer ESB Started in Debug mode
2016-01-22 12:12:01,936] INFO - SynapseDebugInterface Listen on ports : Command 9005 - Event 9006
```

6. In the WSO2 ESB Tooling Plugin, right click and add breakpoints or skip points on the desired mediators to start debugging as shown in the example below.

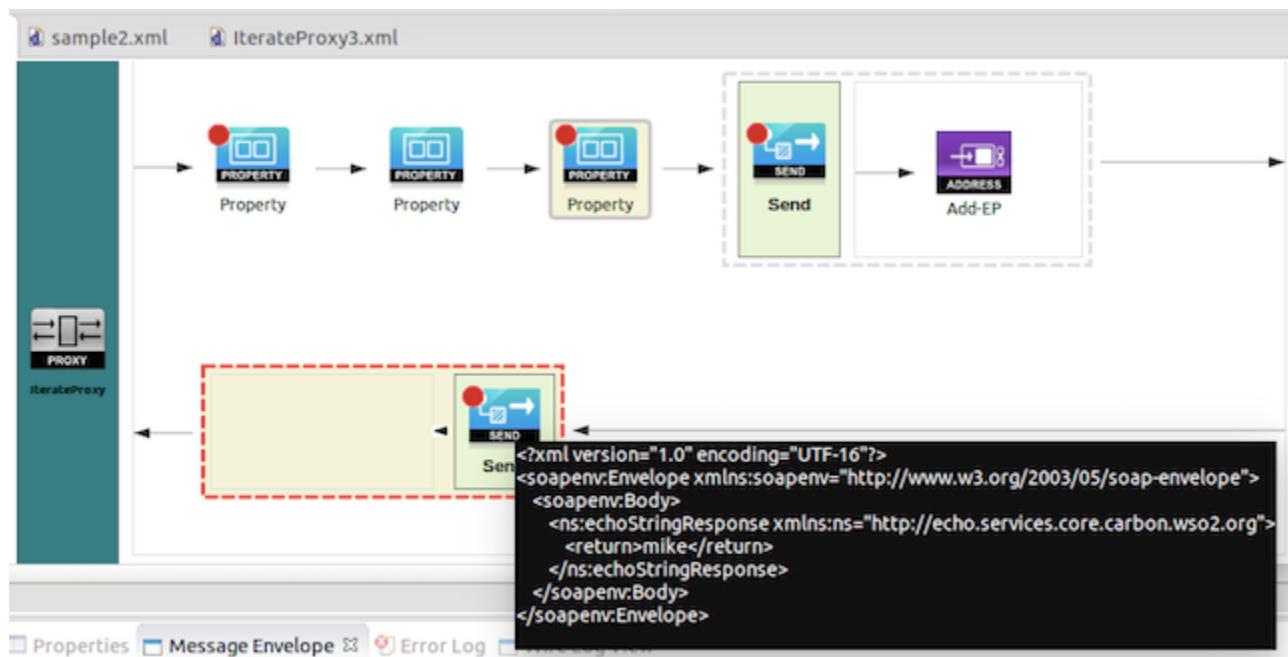


You can add the following debugging options on the mediators using the right click context menu.

- **Toggle Breakpoint:** Adds a breakpoint to the selected mediator
- **Toggle Skip Point:** Adds a skip point to the selected mediator
- **Resend ESB Debug Points:** If you re-start the ESB server, or if you re-deploy the proxy service after changing its Synapse configuration, you need to re-send the information on breakpoints to the WSO2 ESB server. This re-sends all registered debugging points to the ESB Server.
- **Delete All ESB Debug Points:** Deletes all registered debug points from the ESB Server and the WSO2 ESB Tooling Plugin

#### **Information provided by the Debugger Tool**

When your target artifact gets a request message and when the mediation flow reaches a mediator marked as a breakpoint, the message mediation process suspends at that point. A tool tip message of the suspended mediator displays the message envelope of the message payload at that point as shown in the example below.



You can view the message payload at that point of the message flow also in the **Message Envelope** tab as shown below.

```
<?xml version="1.0" encoding="UTF-16"?>
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
<soapenv:Body>
<ns:echoStringResponse xmlns:ns="http://echo.services.core.carbon.wso2.org">
<return>mike</return>
</ns:echoStringResponse>
</soapenv:Body>
</soapenv:Envelope>
```

Also, you can view the message mediation properties in the **Variables** view as shown in the example below.

Name	Value
Transfer-Encoding	chunked
User-Agent	Axis2
Operation Scope Properties	{}
Synapse Scope Properties	{"TRANSPORT_IN_NAME":"http","proxy.name":"IterateProxy","tenant.info.domain":"carbon.super","tenant.info.id":-1234}
TRANSPORT_IN_NAME	http
proxy.name	IterateProxy
tenant.info.domain	carbon.super
tenant.info.id	-1234

The **Variable** view contains properties of the following property scopes.

- Axis2-Client Scope Properties
- Axis2 Scope Properties
- Operation Scope Properties
- Synapse Scope Properties
- Transport Scope Properties

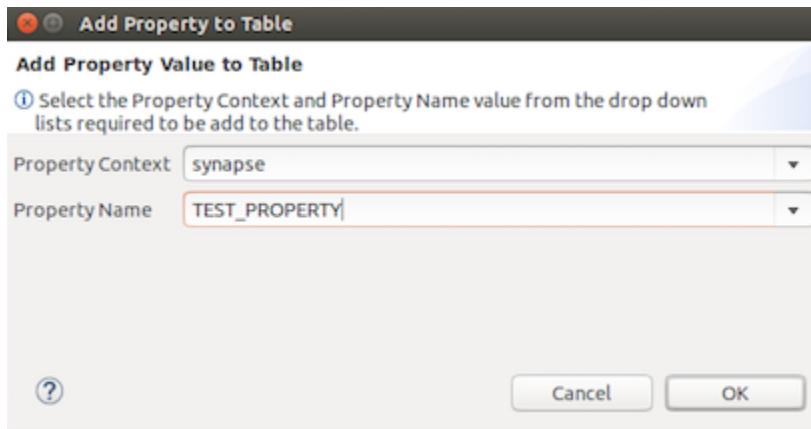
You can have a list of selected properties out of the above, in the properties table of the **Message Envelope** tab, and view information on the property keys and values of them as shown below.

Property Key	Property Value
To	http://www.w3.org/2005/08/addressing/anonymous
From	
Content-Length	248
Accept-Encoding	gzip,deflate
Connection	Keep-Alive
Content-Type	application/soap+xml; charset=UTF-8; action="urn:echoStringResponse"
Host	localhost.localdomain:8280
SOAPAction	
User-Agent	Axis2
TEST_PROPERTY	va

[Add Property](#)[Clear Property](#)

Click **Add Property**, specify the context and name of the property, and then click **OK**, to add that property to the properties table in the **Message Envelope** tab as shown below.

Click **Clear Property**, to remove a property from the properties table.



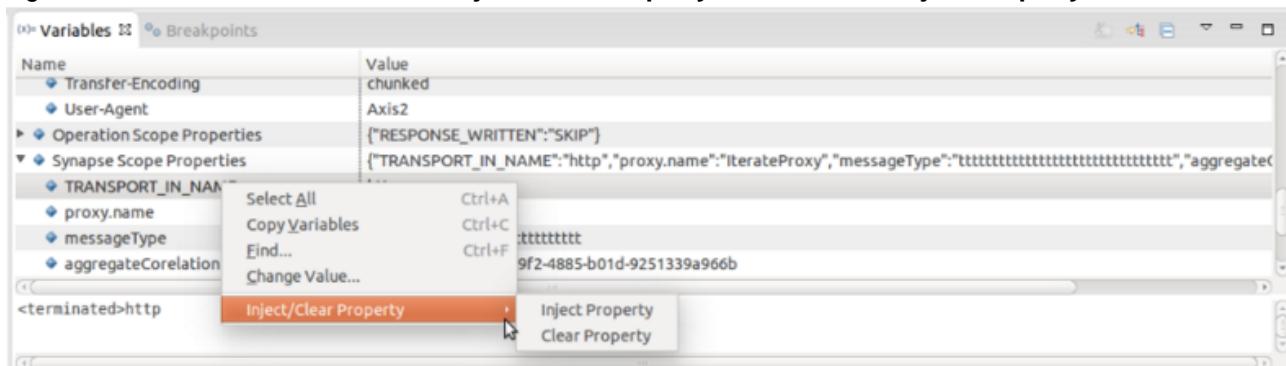
### Changing the property values

There are three operations that you can perform on message mediation property values as described below.

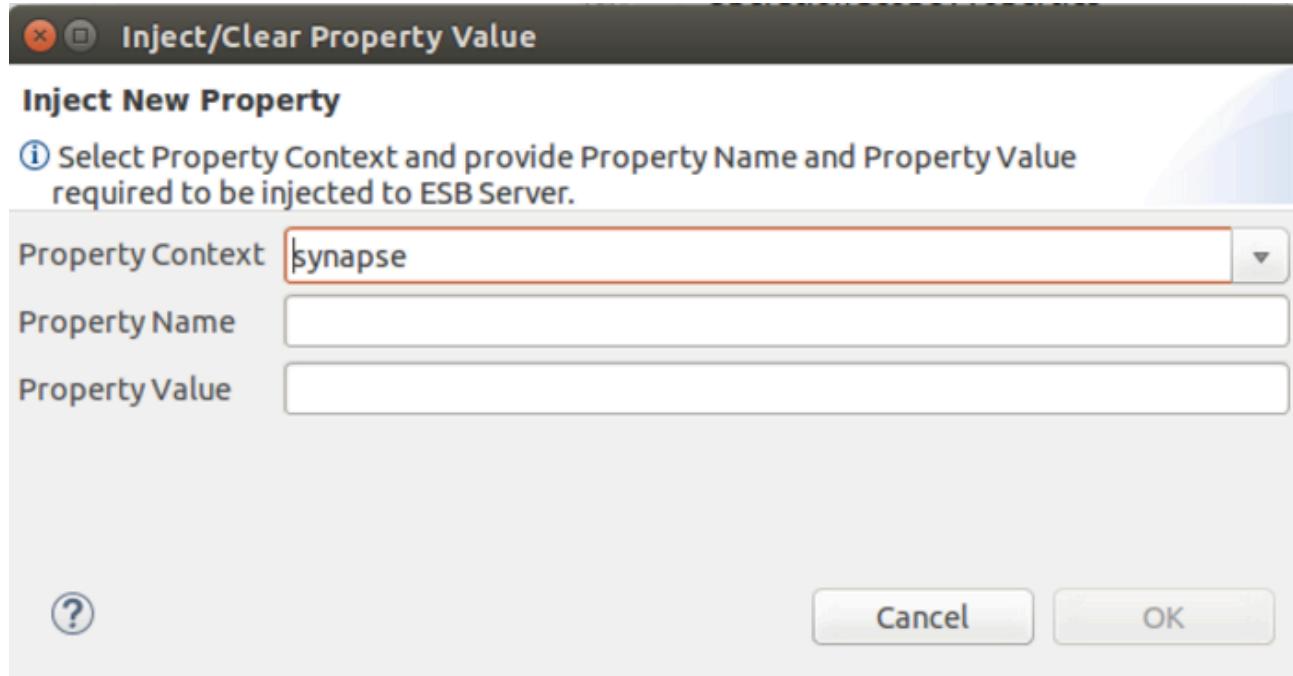
#### Injecting new properties

Follow the steps below to inject new properties to the ESB server while debugging.

- Right click on the **Variable** view, click **Inject/Clear Property**, and then click **Inject Property** as shown below.



- Enter the details about the property you prefer to add as shown in the example below.

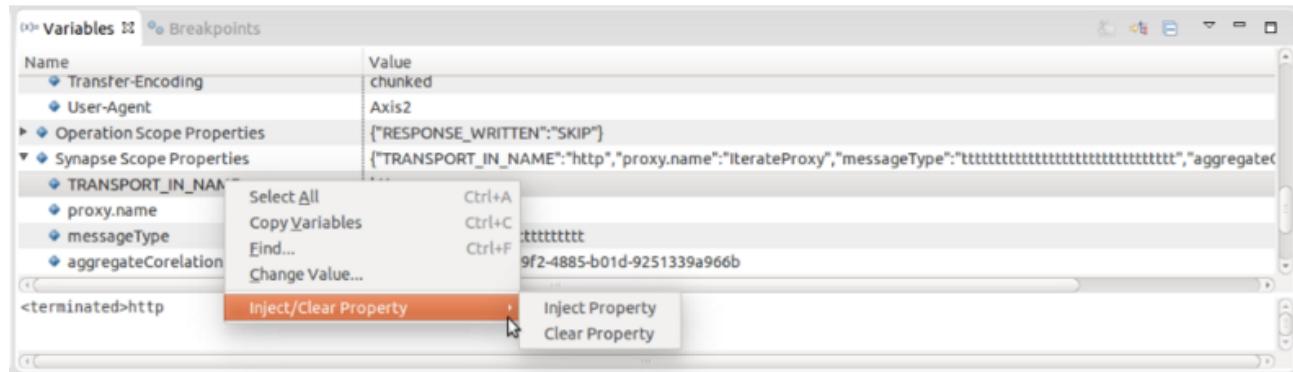


3. Click **OK**.

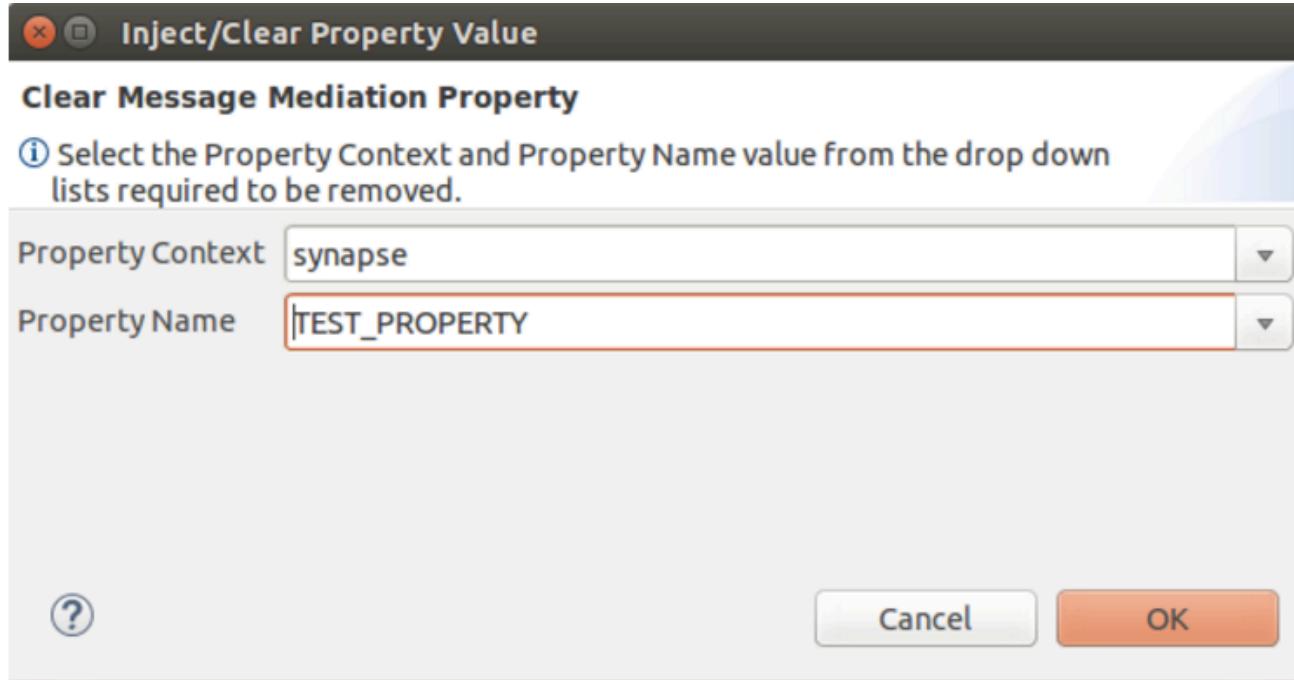
### Clearing a property

Follow the steps below to clear an existing property from the ESB server.

1. Right click on the **Variable** view, click **Inject/Clear Property**, and then click **Clear Property** as shown below.



2. Enter the details about the property you want to clear as shown in the example below.



3. Click **OK**.

### Modifying a property

Click on the value section of the preferred property and change the value in the **Variable** view as shown in the example below, to modify it.

Name	Value
Synapse Scope Properties	{ "TRANSPORT_IN_NAME": "http", "proxy.name": "IterateProxy", "messageType": "text/plain", "ERROR_MESSAGE": "value", "TEST_PROPERTY": "modifyTest" }
TRANSPORT_IN_NAME	http
proxy.name	IterateProxy
messageType	text/plain
ERROR_MESSAGE	value
TEST_PROPERTY	modifyTest
tenant.info.domain	carbon.super
tenant.info.id	-1234

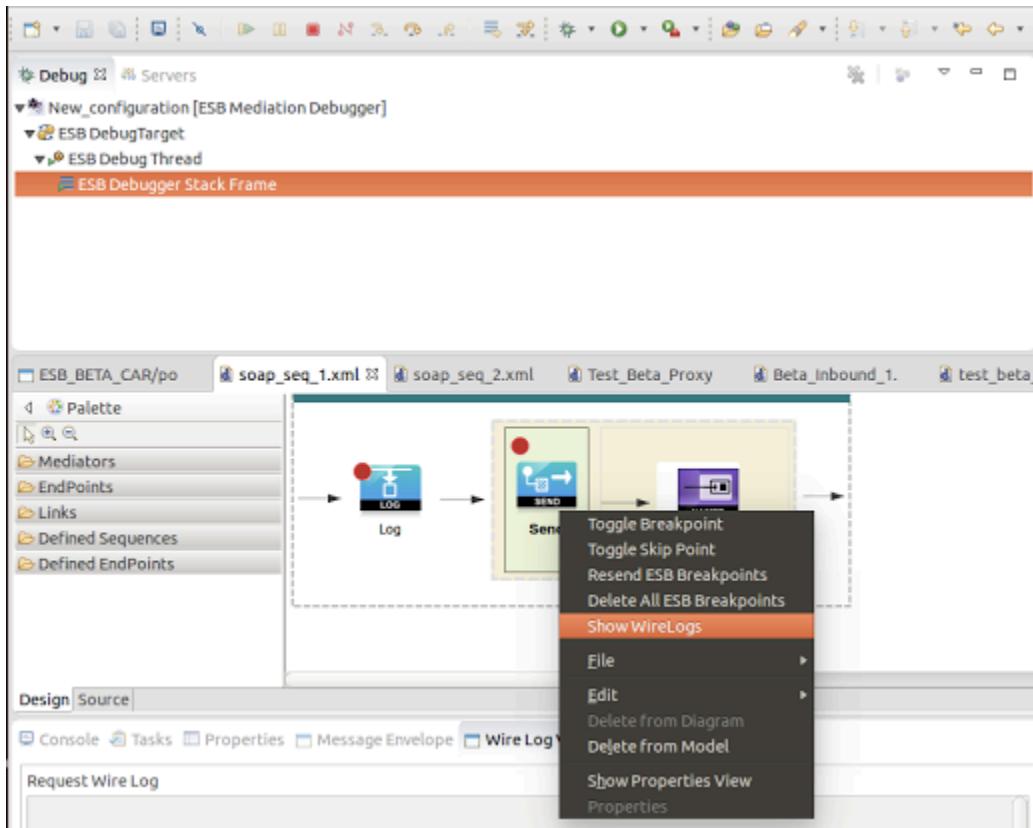
### Viewing wire logs

You can view wire logs at the entry point of WSO2 ESB, while debugging a Synapse flow. For example, you can view wire logs of the incoming flow and the final response of a proxy service. Also, you can view wire logs for points, where it goes out from the ESB. For example, you can see the outgoing and incoming wire logs for specific mediators (i.e. Call mediator, Send mediator etc.).

### Viewing wire logs of a specific mediator

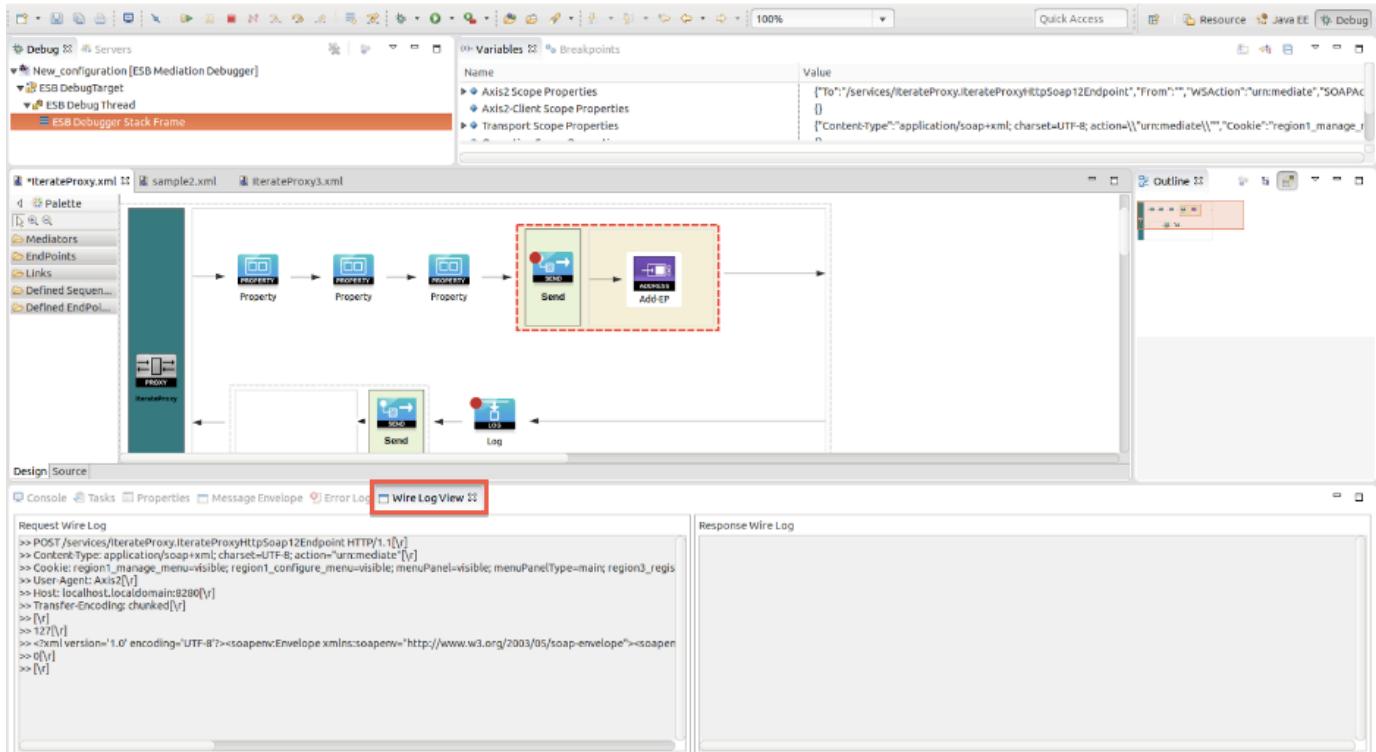
You need to put a debug point to the mediator, to view wire logs of it. When debugging is finished (or while debugging), right click on the mediator, and click **Show WireLogs**, to view wire logs for a specific mediator.

You can only view wire logs for a **whole proxy service, call mediator, send mediator, or other API resources**. However, you cannot view a wire log of a Synapse config (e.g. sequences), because there would not be anything written to wire, when the flow comes to the sequence etc. Hence, you can only view them in wire entry points.



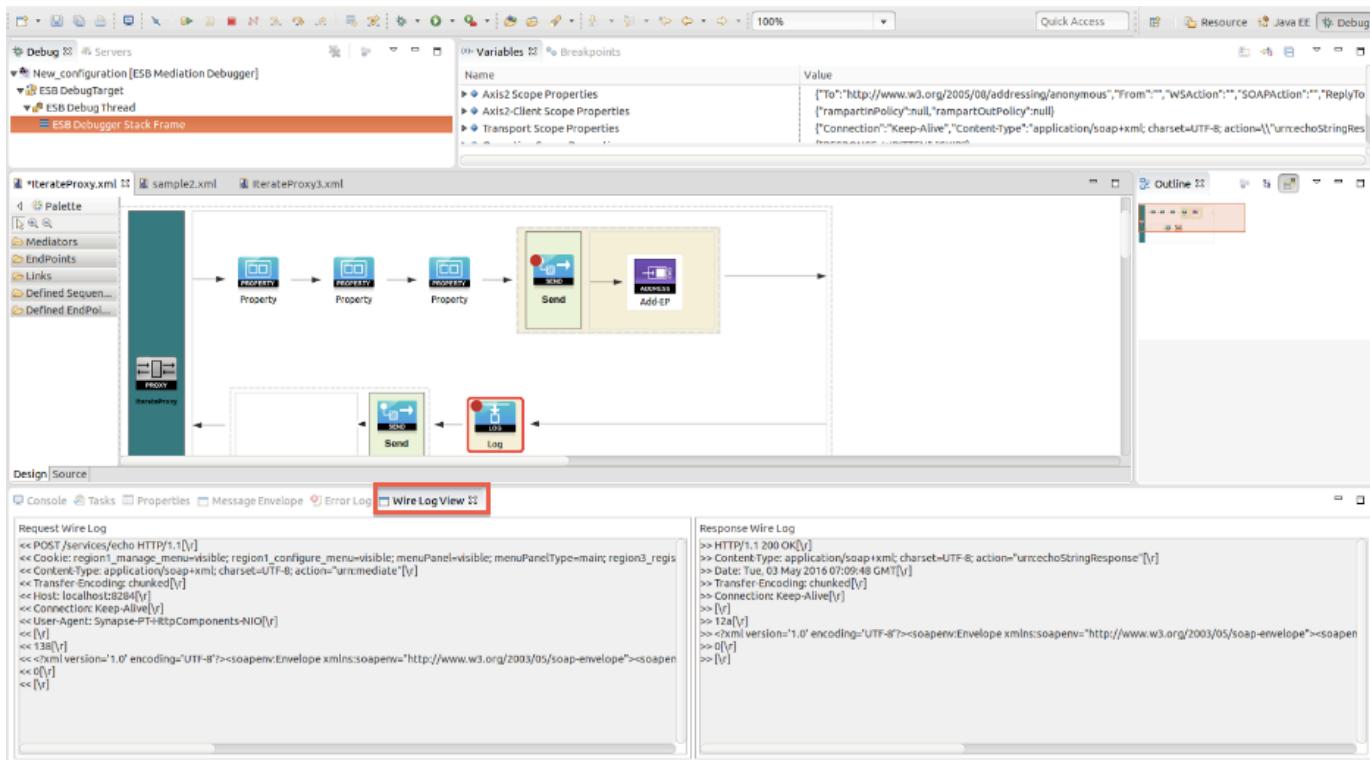
### Viewing wire logs while debugging

If you view wire logs while debugging, you view only the wire logs of mediators, whose execution is already completed as shown in the example below.



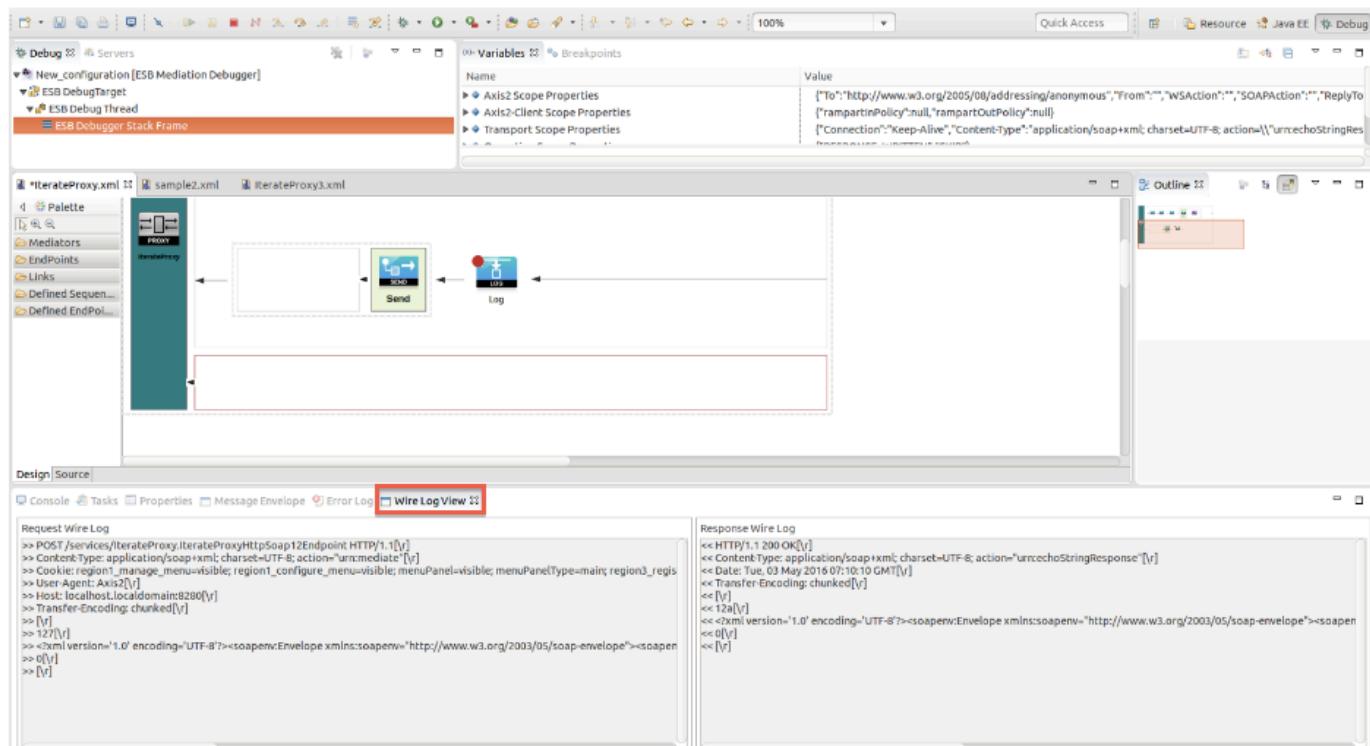
### Viewing wire logs of a mediator after debugging execution of it

When you view wire logs of a mediator (e.g. send mediator) after debugging, you can view the request and response wire logs as shown in the example below.



### Viewing wire logs of a proxy service after debugging

If you view wire logs of a proxy service after debugging finished, you view the request wire log and final response wire log of that proxy as shown in the example below.



## Working with Endpoints

An **endpoint** defines an external destination for an outgoing message through WSO2 ESB. Typically, the endpoint is the address of a proxy service, which acts as the front end to the actual service. For example, the endpoint for the simple stock quote sample is `http://localhost:9000/services/SimpleStockQuoteService`.

For detailed information on each endpoint type available with WSO2 ESB, see [WSO2 ESB Endpoints](#).

### Configuring endpoints

In the XML configuration, the `<endpoint>` element defines an endpoint as follows:

```
<endpoint [name="string"] [key="string"]>
 address-endpoint | default-endpoint | wsdl-endpoint | load-balanced-endpoint |
 fail-over-endpoint
</endpoint>
```

### Using named endpoints

You can use the `name` attribute to create a named endpoint. You can reuse a named endpoint by referencing it in another endpoint using the `key` attribute. For example, if there is an endpoint named `foo`, you can reference the `foo` endpoint in any other endpoint where you want to use `foo`:

```
<endpoint key="foo" />
```

This approach allows you to reuse existing endpoints in multiple places.

### Working with endpoints

You can either use the ESB tooling plug-in to create a new endpoint and to import an existing endpoint, or you can manage endpoints via the ESB Management Console. For detailed information on how to work with endpoints, see the following topics:

- [Working with Endpoints via WSO2 ESB Tooling](#)
- [Working with Endpoints via the Management Console](#)

### Tracing and handling errors

Endpoints have a `trace` attribute, which turns on detailed trace information for messages being sent to the endpoint. These are available in the `trace.log` file, which is configured in the `<PRODUCT_HOME>/repository/conf/log4j.properties` file. Setting the trace log level to `TRACE` logs detailed trace information including message payloads. For more information on endpoint states and handling errors, see [Endpoint Error Handling](#).

## WSO2 ESB Endpoints

Following are the endpoint types available with WSO2 ESB. Click on a required endpoint type to view detailed information on the endpoint.

- [Address Endpoint](#)
- [Dynamic Load-balance Endpoint](#)
- [Failover Group](#)
- [HTTP Endpoint](#)
- [WSDL Endpoint](#)
- [Load-balance Group](#)
- [Indirect and Resolving Endpoints](#)

- Default Endpoint
- Template Endpoint
- Recipient List Endpoint

## Address Endpoint

The **Address endpoint** is an endpoint defined by specifying the EPR (Endpoint Reference) and other attributes of the configuration.

[XML configuration](#) | [UI configuration](#)

### XML configuration

#### Note

You can configure the Address Endpoint using XML. Click on the **Switch to source view** link in the **Address Endpoint** page.

Address Endpoint  Switch to source view

```
<address uri="endpoint address" [format="soap11|soap12|pox|rest|get"]
[optimize="mtom|swa"]
[encoding="charset encoding"]
[statistics="enable|disable"] [trace="enable|disable"]>

<enableSec [policy="key"]/>?
<enableAddressing [version="final|submission"] [separateListener="true|false"]/>?

<timeout>
 <duration>timeout duration in milliseconds</duration>
 <responseAction>discard|fault</responseAction>
</timeout>?

<markForSuspension>
 [<errorCodes>xxx,yyy</errorCodes>]
 <retriesBeforeSuspension>m</retriesBeforeSuspension>
 <retryDelay>d</retryDelay>
</markForSuspension>

<suspendOnFailure>
 [<errorCodes>xxx,yyy</errorCodes>]
 <initialDuration>n</initialDuration>
 <progressionFactor>r</progressionFactor>
 <maximumDuration>l</maximumDuration>
</suspendOnFailure>
</address>
```

### Address endpoint attributes

Attribute	Description
uri	EPR of the target endpoint.
format	Message format for the endpoint.

optimize	Method to optimize the attachments.
encoding	The charset encoding to use for the endpoint.
statistics	This enables statistics for the endpoint.
trace	This enables trace for the endpoint.

### Other elements

QoS for the endpoint

QoS (Quality of Service) aspects such as WS-Security and WS-Addressing may be enabled on messages sent to an endpoint using `enableSec` and `enableAddressing` elements. Optionally, the WS-Security policies could be specified using the `policy` attribute.

QoS configuration

enableSec [policy="key"]	This enables WS-Security for the message which is sent to the endpoint. The optional <code>policy</code> attribute specifies the WS-Security policy.
enableAddressing [version="final   submission"] [seperateListener=" true   false"]	This enables WS-Addressing for the message which is sent to the endpoint. User can specify to have separate listener with version final or submission.

Endpoint timeout

The parameters available to configure an endpoint time out are as follows.

duration	Timeout duration that should elapse before the end point is timed out.
responseAction	<p>This parameter is used to specify the action to perform once an endpoint has timed out. The available options are as follows.</p> <ul style="list-style-type: none"> <li>• <code>discard</code>: If this is selected, the responses which arrive after the endpoint has timed out will be discarded.</li> <li>• <code>fault</code>: If this is selected, a fault is triggered when the endpoint is timed out.</li> </ul> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p>You can specify a value that is 1 millisecond less than the time duration you specify for the endpoint time out for the <code>synapse.timeout_handler_interval</code> property in the <code>&lt;ESB_Home&gt;/repository/conf/synapse.properties</code> file. This would minimise the number of late responses from the backend.</p> </div>

Marking an endpoint for suspension

The `markForSuspension` element contains the following parameters which affect the suspension of a endpoint which would be timed out after a specified time duration.

errorCodes	<p>This parameter is used to specify one or more error codes which can cause the endpoint to be marked for suspension when they are returned by the endpoint. Multiple error codes can be specified separated by commas. See <a href="#">SynapseConstant class</a> for a list of available error codes.</p>
retriesBeforeSuspension	<p>Number of retries before go into suspended state.</p> <p>The number of times the endpoint should be allowed to retry sending the response before it is marked for suspension.</p>

retryDelay	The delay between each try.
------------	-----------------------------

Suspending the endpoint on failure

Leaf endpoints(Address and WSDL) can be suspended if they are detected as failed endpoints. When an endpoint is in suspended state for a specified time duration following a failure, it cannot process any new messages. The following formula determines the wait time before the next attempt.

```
next suspension time period = Max (Initial Suspension duration * (progression factor*
try count *), Maximum Duration)
```

All the variables in the above formula are configuration values used to calculate the try count. Try count is the number of tries carried out after the endpoint is suspended. The increase in the try count causes an increase in the next suspension time period. This time period is bound to a maximum duration.

The parameters available to configure a suspension of an endpoint due to failure are as follows.

Parameter Name	Description
errorCode	A comma separated error code list which can be returned by the endpoint.  This parameter is used to specify one or more error codes which can cause the endpoint to be suspended when they are returned from the endpoint. Multiple error codes can be specified, separated by commas.
initialDuration	The number of milliseconds after which the endpoint should be suspended when it is being suspended for the first time.
progressionFactor	The progression factor for the geometric series. See the above formula for a more detailed description.
maximumDuration	The maximum duration (in milliseconds) to suspend the endpoint.

Following are the sample address URI definition.

Transport	Sample Address
HTTP	http://localhost:9000/services/SimpleStockQuoteService
JMS	jms:/SimpleStockQuoteService? transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory& java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory& java.naming.provider.url=tcp://localhost:61616& transport.jms.DestinationType=topic
Mail	guest@host
VFS	vfs:file:///home/user/directory\ vfs:>file:///home/user/file\ vfs:
FIX	fix://host:port?BeginString=FIX4.4&SenderCompID=WSO2&TargetCompID=APACHE

## UI configuration

The following page is opened by clicking **Address Endpoint** in the **Add Endpoint** tab of the **Manage Endpoints** page.

## Address Endpoint

Address Endpoint Switch to source view

Name *	<input type="text"/>
Address *	<input type="text"/> Test
<input checked="" type="checkbox"/> Show Advanced Options	
<b>Endpoint Properties</b>	
Add Property	
<input type="button" value="Save &amp; Close"/> <input type="button" value="Save in Registry"/> <input type="button" value="Cancel"/>	

The parameters available to configure the endpoint are as follows.

Parameter Name	Description
Name	The unique name of the endpoint.
Address	The URL of the endpoint. You can test the availability of the given URL by clicking <b>Test</b> .
Show Advanced Options	<p>This section is used to enter advanced settings for the endpoint. The advanced options specific for the Address endpoint are as follows.</p> <ul style="list-style-type: none"> <li>• <b>Format</b>- The message format for the endpoint. The available values are as follows.           <ul style="list-style-type: none"> <li>• <b>Leave As-Is</b>: If this is selected, no transformation is done to the outgoing message.</li> <li>• <b>SOAP 1.1</b>: If this is selected the message is transformed to SOAP 1.1.</li> <li>• <b>SOAP 1.2</b>: If this is selected the message is transformed to SOAP 1.2.</li> <li>• <b>Plain Old XML (POX)</b>: If this is selected the message is transformed to plain old XML format.</li> <li>• <b>Representational State Transfer (REST)</b> - If this is selected, the message is transformed to REST.</li> <li>• <b>GET</b>: If this is selected, the message is transformed to a HTTP Get Request.</li> </ul> </li> <li>• <b>Optimize</b>- Optimization for the message, which transfers binary data. The available values are as follows.           <ul style="list-style-type: none"> <li>• <b>Leave As-Is</b> - If this is selected, there will be no special optimization. The original message will be kept.</li> <li>• <b>SwA</b> - If this is selected, the message is optimized as a SwA (SOAP with Attachment) message.</li> <li>• <b>MTOM</b> - If this is selected, the message is optimized using a MTOM (message transmission optimization mechanism).</li> </ul> </li> </ul>
Add Property	<p><b>Note</b></p> <p>The rest of the advanced options are common for Address, WSDL, Default endpoints. See the description of common options in <a href="#">Managing Endpoints</a>.</p> <p>This section is used to add properties to an endpoint.</p>

## Dynamic Load-balance Endpoint

The **Dynamic Load-balance Endpoint** is an [endpoint](#) that distributes its messages (load) among application members by evaluating the load-balancing policy and any other relevant parameters. These application members will be discovered using the `membershipHandler` class, which generally uses a group communication mechanism to discover the application members. The `class` attribute of the `membershipHandler` element should be an implementation of `org.apache.synapse.core.LoadBalanceMembershipHandler`. You can specify `membershipHandler` properties using the `property` elements. The `policy` attribute of the `dynamicLoadbalance` element specifies the load-balancing policy (algorithm) to be used for selecting the next member that will receive the message.

Currently only the `roundRobin` policy is supported.

The `failover` attribute determines if the next member should be selected once the currently selected member has failed and defaults to true.

### Dynamic Load-balance Endpoint Configuration

Currently the Dynamic Load-balance Endpoint does not support configuring it using a UI. However, you can configure it using its XML source code in the **Source View**. Click **Source View** under **Service Bus** in the **Main** menu tab of the ESB management console, to add a Dynamic Load-balance Endpoint. The syntax is as follows.

```
<dynamicLoadBalance [policy="roundRobin"] [failover="true|false"]>
 <membershipHandler class="impl of
 org.apache.synapse.core.LoadBalanceMembershipHandler">
 <property name="name" value="value"/>
 </membershipHandler>
</dynamicLoadBalance>
```

See [Sample 57: Dynamic Load Balancing between Three Nodes](#) for an example.

## Failover Group

A **Failover Group** is a list of leaf endpoints grouped together for the purpose of passing an incoming message from one endpoint to another if a failover occurs. The first endpoint in failover group is considered the primary endpoint. An incoming message is first directed to the primary endpoint, and all other endpoints in the group serve as back-ups. If the primary endpoint fails, the next active endpoint is selected as the primary endpoint, and the failed endpoint is marked as inactive. Thus, failover group ensures that a message is delivered as long as there is at least one active endpoint among the listed endpoints. The ESB switches back to the primary endpoint as soon as it becomes available. This behaviour is known as dynamic failover.

An endpoint failure occurs when an endpoint is unable to invoke a service. Note that an endpoint which responds with an error is not considered a failed endpoint.

---

[XML Configuration](#) | [UI Configuration](#)

---

### XML Configuration

## Note

You can configure the Failover endpoint using XML. Click on **Switch to source view** link in the **Failover Group** page.

```
<failover>
 <endpoint .../>+
</failover>
```

### UI Configuration

The following page is opened by clicking **Failover Group** in the **Add Endpoint** tab of the **Manage Endpoints** page.

#### Failover Group

1. Enter a name for the failover group endpoint, and if you want to add any [properties](#), click **Add Property** and specify the properties.
2. To add a child endpoint to the failover endpoint, click **Add Child**, and then select the required endpoint type from the list.
3. Do the following:
  - a. Enter the basic details for the child endpoint, such as the name and address.
  - b. Click **Show Advanced Options** and specify the [advanced options](#) you want for this endpoint.
  - c. To add [properties](#) to the child endpoint, click **Add Property** and specify the properties.
  - d. Click **Update**.
4. Add more child endpoints as needed, and then save the failover group endpoint.

### HTTP Endpoint

The **HTTP endpoint** allows you to define REST endpoints using [URI templates](#) similar to the REST API. The URI templates allow a RESTful URI to contain variables that can be populated during mediation runtime using property values whose names have the "uri.var." prefix. An HTTP endpoint can also define the particular HTTP method to use in the RESTful invocation.

## XML Configuration | UI Configuration | Example

### XML Configuration

#### Note

You can configure the HTTP endpoint using XML. Click on **Switch to source view** link in the **HTTP Endpoint** page.

Failover Group  [Switch to source view](#)

The syntax is as follows.

```
<http uri-template="URI Template" method="GET|POST|PUSH|PUT|DELETE|OPTIONS|HEAD" />
```

### HTTP Endpoint Attributes

Attribute	Description
uri-template	The <a href="#">URI template</a> that constructs the RESTful endpoint URL at runtime.
method	The HTTP method to use during the invocation.

### UI Configuration

The following page is opened by clicking **HTTP Endpoint** in the **Add Endpoint** tab of the **Manage Endpoints** page.

Home > Manage > Service Bus > Endpoints > HTTP Endpoint

 [Help](#)

### HTTP Endpoint

HTTP Endpoint  [Switch to source view](#)

Name *	<input type="text"/>
URI Template *	<input type="text"/> 
<b>HTTP Options</b>	
HTTP Method	<input type="button" value="GET"/>
<input checked="" type="checkbox"/> <a href="#">Show Advanced Options</a>	
<b>Endpoint Properties</b>	
 <a href="#">Add Property</a>	
<input type="button" value="Save &amp; Close"/> <input type="button" value="Save in Registry"/> <input type="button" value="Cancel"/>	

The parameters to configure an HTTP endpoint are as follows.

Parameter Name	Description
<b>Name</b>	This parameter is used to enter a unique name for the endpoint.
<b>URI Template</b>	<p>The URI template of the endpoint. Insert <code>uri.var.</code> before each variable. Click <b>Test</b> to test the URI.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p>If the endpoint URL is an encoded URL, then you need to add <code>legacy-encoding: w</code> hen defining the uri-template.</p> <p>e.g., <code>uri-template="legacy-encoding:{uri.var.APIurl}"</code></p> </div>
<b>HTTP Method</b>	<p>The HTTP method to use during the invocation of the endpoint. Supported methods are as follows.</p> <ul style="list-style-type: none"> <li>• Get</li> <li>• Post</li> <li>• Push</li> <li>• Put</li> <li>• Delete</li> <li>• Options</li> <li>• Head</li> <li>• Leave-as-is</li> </ul>
<b>Show Advanced Options</b>	Click this link if you want to add advanced options to the endpoint. See Advanced Options for details of common advanced options you can add.

You can create HTTP endpoints by specifying values for the parameters given above.

Alternatively, you can specify one parameter as the HTTP endpoint by using multiple other parameters, and then pass that to define the HTTP endpoint as follows:

```
<property name="uri.var.httpendpointurl" expression="fn:concat($ctx:prefixuri,
$ctx:host, $ctx:port, $ctx:urlparam1, $ctx:urlparam2)" />
<send>
<endpoint>
<http uri-template="
{uri.var.httpendpointurl}
"/>
</endpoint>
</send>
```

## Example

```
<endpoint xmlns="http://ws.apache.org/ns/synapse" name="HTTPEndpoint">
<http
uri-template="http://localhost:8080/{uri.var.servicepath}/restapi/{uri.var.servicename}
/menu?category={uri.var.category}&type={uri.var.pizzaType}" method="GET">
</http>
</endpoint>
```

The URI template variables in this example HTTP endpoint can be populated during mediation as follows:

```
<inSequence>
 <property name="uri.var.servicepath" value="PizzaShopServlet"/>
 <property name="uri.var.servicename" value="PizzaWS"/>
 <property name="uri.var.category" value="pizza"/>
 <property name="uri.var.pizzaType" value="pan"/>
 <send>
 <endpoint key="HTTPEndpoint"/>
 </send>
</inSequence>
```

This configuration will cause the RESTful URL to evaluate to:

`http://localhost:8080/PizzaShopServlet/restapi/PizzaWS/menu?category=pizza&type=pan`

## WSDL Endpoint

The **WSDL Endpoint** is an endpoint definition based on a specified WSDL document.

The WSDL document can be specified in 2 ways:

- As a URI.
- As an inlined definition within the endpoint configuration.

---

[XML Configuration](#) | [UI Configuration](#)

---

### XML Configuration

#### Note

You can configure the WSDL endpoint using XML. Click on the **Switch to source view** link in the **WSDL Endpoint** page.

[WSDL Endpoint](#)  [Switch to source view](#)

The syntax of the endpoint is as follows.

```

<wsdl [uri="wsdl-uri"] service="qname" port/endpoint="qname">
 <wsdl:definition>...</wsdl:definition>?
 <wsdl20:description>...</wsdl20:description>?
 <enableRM [policy="key"]/>?
 <enableSec [policy="key"]/>?
 <enableAddressing/>?

 <timeout>
 <duration>timeout duration in milliseconds</duration>
 <action>discard|fault</action>
 </timeout>?

 <markForSuspension>
 [<errorCodes>xxx,yyy</errorCodes>]
 <retriesBeforeSuspension>m</retriesBeforeSuspension>
 <retryDelay>d</retryDelay>
 </markForSuspension>

 <suspendOnFailure>
 [<errorCodes>xxx,yyy</errorCodes>]
 <initialDuration>n</initialDuration>
 <progressionFactor>r</progressionFactor>
 <maximumDuration>l</maximumDuration>
 </suspendOnFailure>
</wsdl>

```

The service and port name containing the target EPR has to be specified with the service and port (or endpoint) attributes respectively.

enableRM, enableSec, enableAddressing, suspendDurationOnFailure and timeout elements are same as for an [Address Endpoint](#).

## UI Configuration

The following page is opened by clicking **WSDL Endpoint** in the **Add Endpoint** tab of the Manage Endpoints page.

## WSDL Endpoint

The screenshot shows the 'WSDL Endpoint' configuration page. At the top, there's a 'Switch to source view' link. Below are four input fields: 'Name' (mandatory), 'WSDL URI' (mandatory) with a 'Test URI' button, 'Service' (mandatory), and 'Port' (mandatory). A checkbox for 'Show Advanced Options' is checked. Below these are sections for 'Endpoint Properties' and 'Add Property'. At the bottom are three buttons: 'Save & Close', 'Save in Registry', and 'Cancel'.

Parameters available to configure a WSDL endpoint are as follows.

Parameter Name	Description
Name	This parameter is used to enter a unique name for the endpoint.
WSDL URI	The URI of the WSDL. Click <b>Test</b> to test the URI.
Service	The service selected from the available services for the WSDL.
Port	The port selected for the service specified in the <b>Service</b> parameter. In a WSDL, an endpoint is bound to each port inside each service.
Show Advanced Options	Click this link if you want to add advanced options for the endpoint. See <a href="#">Advanced Options</a> for details of the available advanced options.
Add Property	Click this link if you want to add properties to the endpoint. See <a href="#">Properties Reference</a> for details of the available properties.

### Load-balance Group

The **Load-balanced Group** distributes the messages (load) arriving at it among a set of listed endpoints or static members by evaluating the load balancing policy and other relevant parameters.

[XML Configuration](#) | [UI Configuration](#)

### XML Configuration

#### Note

You can configure the Load-balance endpoint using XML. Click on the **Switch to source view** link in the **Load Balance Group** page.

Load Balance Group  Switch to source view

The syntax of the load balance endpoint is as follows.

```
<session type="http|simpleClientSession"/>?
<loadBalance [policy="roundRobin"] [algorithm="impl of
org.apache.synapse.endpoints.algorithms.LoadbalanceAlgorithm"]
 [failover="true|false"]>
 <endpoint .../>+
 <member hostName="host" [httpPort="port"] [httpsPort="port2"]>+
</loadBalance>
```

The Load-balance attributes and elements:

- The `policy` attribute of the load balance element specifies the load balance policy (algorithm) to be used for selecting the target endpoint or static member.

### Tip

Currently only the `roundRobin` policy is supported.

- The `failover` attribute determines if the next endpoint or static member should be selected once the currently selected endpoint or static member has failed, and defaults to true.
- The `loadBalance` element allows to list the set of endpoints or static members among which the load has to be distributed. These endpoints can belong to any endpoint type (see [Address Endpoint](#), [WSDL Endpoint](#), [Default Endpoint](#), [Failover Group](#)). For example, Failover Endpoints can be listed inside the Load-balance endpoint to load balance between failover groups etc.

### Tip

The `loadBalance` element cannot have both `endpoint` and `member` child elements in the same configuration. In the case of the `member` child element, the `hostName`, `httpPort` and/or `httpsPort` attributes could be specified.

- The optional `session` element makes the endpoint a session affinity based load balancing endpoint. If it is specified, sessions are bound to endpoints in the first message and all successive messages for those sessions are directed to their associated endpoints.

### Tip

Currently there are two types of sessions supported in SAL endpoints. Namely HTTP transport based session which identifies the sessions based on HTTP cookies and the client session which identifies the session by looking at a SOAP header sent by the client with the QName  
`"[http://ws.apache.org/ns/synapse]ClientID"`.

- The `failover` attribute mentioned above is not applicable for session affinity based endpoints and it is always considered as set to false. If it is required to have failover behavior in session affinity based load balance endpoints, list failover endpoints as the target endpoints.

## UI Configuration

The following page is opened by clicking **Load-balance Endpoint** in the **Add Endpoint** tab of the **Manage Endpoints** page.

The screenshot shows the 'Load-balance Group' configuration page. At the top, there are navigation links: Home > Manage > Service Bus > Endpoints > Load-balance Group. On the right, there is a 'Help' link. The main area has tabs: 'Switch to source' and 'Switch to source view'. Below these are configuration fields:

- Endpoint Name \***: A text input field.
- Algorithm**: A dropdown menu set to 'Round-robin'.
- Session Management**: A dropdown menu set to 'None'.
- Session Timeout (Mills)**: A text input field containing '0'.

Below the configuration fields is a tree view panel with a single node 'root' and a '+' icon for 'Add Child'.

At the bottom of the page is a 'Load Balance Endpoint Properties' section with a '+ Add Property' button. The footer contains three buttons: 'Save & Close', 'Save in Registry', and 'Cancel'.

The parameters available to configure a load-balance endpoint are as follows.

Parameter Name	Description
<b>Endpoint Name</b>	This parameter is used to enter a unique name for the endpoint.
<b>Algorithm</b>	The algorithm on which the load balancing is based. <b>Round Robin</b> is the default algorithm. You can also load a custom algorithm. Instructions for creating a custom algorithm are included in <a href="#">this article</a> .

<b>Session Management</b>	A session management method from the load balancing group. The possible values are as follows. <ul style="list-style-type: none"> <li>• <b>None</b> - If this is selected, session management is not used.</li> <li>• <b>Transport</b> - If this is selected, session management is done on the transport level using HTTP cookies.</li> <li>• <b>SOAP</b> - If this is selected, session management is done using SOAP sessions.</li> <li>• <b>Client ID</b> - If this is selected, session management is done using an ID sent by the client.</li> </ul>
<b>Session Timeout (Mills)</b>	The number of milliseconds after which the session would time out.
<b>Add Child</b>	Click this link to add a child endpoint. The required endpoint type can be selected from the list and a section displaying parameters relevant to the selected endpoint type will appear. You can add as many endpoints as required to the load-balance group. See the details of child endpoints in <a href="#">Address Endpoint</a> , <a href="#">WSDL Endpoint</a> , and <a href="#">Failover Group</a> .
<b>Add Property</b>	Click this link to add properties to the load-balance endpoint. See <a href="#">Properties Reference</a> for details of the available properties.

## Indirect and Resolving Endpoints

Indirect and Resolving endpoints are endpoint configurations with a key which refers to an existing endpoint.

### Indirect Endpoint

The **Indirect Endpoint** refers to an actual [endpoint](#) by a key. This endpoint fetches the actual endpoint at runtime. Then it delegates the message sending to the actual endpoint. When endpoints are stored in the [registry](#) and referred, this endpoint can be used. The key is a static value for this endpoint.

### XML Configuration

```
<endpoint key="" name="" />
```

### Example

In the following [Send mediator](#) configuration, the `PersonInfoEpr` key refers to a specific endpoint saved in the [registry](#).

```
<send>
<endpoint key="PersonInfoEpr" />
</send>
```

### Resolving Endpoint

The **Resolving Endpoint** refers to an actual endpoint using a dynamic key. The key is an XPath expression.

1. The XPath is evaluated against the current message and key is calculated at run time.
2. Resolving endpoint fetches the actual endpoint using the calculated key.
3. Resolving endpoint delegates the message sending it to the actual endpoint.

When endpoints are stored in the registry and referred, this endpoint can be used.

The XPath expression specified in a Resolving endpoint configuration derives an existing [endpoint](#) rather than the URL of the endpoint to which the message is sent. To derive the endpoint URL to which the message is sent via an XPath expression, use the [Header Mediator](#).

## XML Configuration

```
<endpoint name="" key-expression="" />
```

### Example

In the following [Send mediator](#) configuration, the endpoint to which the message is sent is determined by the `get-property('Mail')` expression.

```
<send>
 <endpoint key-expression="get-property('Mail')"/>
</send>
```

## Default Endpoint

The **Default Endpoint** is an [endpoint](#) defined for adding QoS and other configurations to the endpoint which is resolved from the `To` address of the message context. All the configurations such as message format for the endpoint, the method to optimize attachments, and security policies for the endpoint can be specified as in the [Address Endpoint](#). This endpoint differs from the address endpoint only in the `URI` attribute which will not be present in this endpoint.

[XML Configuration](#) | [UI Configuration](#)

## XML Configuration

### Note

You can configure the Default endpoint using XML. Click **Switch to source view** in the **Default Endpoint** page.

Default Endpoint  [Switch to source view](#)

```

<default [format="soap11|soap12|pox|get"] [optimize="mtom|swa"]
[encoding="charset encoding"]
[statistics="enable|disable"] [trace="enable|disable"]>

<enableRM [policy="key"]/>?
<enableSec [policy="key"]/>?
<enableAddressing [version="final|submission"] [separateListener="true|false"]/>?

<timeout>
 <duration>timeout duration in milliseconds</duration>
 <action>discard|fault</action>
</timeout>?

<markForSuspension>
 [<errorCodes>xxx,yyy</errorCodes>]
 <retriesBeforeSuspension>m</retriesBeforeSuspension>
 <retryDelay>d</retryDelay>
</markForSuspension>

<suspendOnFailure>
 [<errorCodes>xxx,yyy</errorCodes>]
 <initialDuration>n</initialDuration>
 <progressionFactor>r</progressionFactor>
 <maximumDuration>l</maximumDuration>
</suspendOnFailure>
</default>

```

## UI Configuration

To configure a default endpoint in the Management Console, go to the **Main** tab and click **Endpoints** to open the **Manage Endpoints** page. Click on the **Add Endpoint** tab and then click **Default Endpoint**.

The screenshot shows the 'Default Endpoint' configuration dialog. It includes fields for 'Name' (marked with a red asterisk), 'Show Advanced Options' (checked), and 'Add Property'. Buttons at the bottom include 'Save & Close', 'Save in Registry', and 'Cancel'.

Parameters available to configure the default endpoint are as follows.

Parameter Name	Description
Name	This parameter is used to define a unique name for the endpoint.

<b>Show Advanced Options</b>	<p>Click this link if you want to add advanced options for the endpoint. Advanced options specific to the default endpoint are as follows.</p> <ul style="list-style-type: none"> <li>• <b>Format</b>- The message format for the endpoint. The available values are as follows.           <ul style="list-style-type: none"> <li>• <b>Leave As-Is</b>: If this is selected, no transformation is done to the outgoing message.</li> <li>• <b>SOAP 1.1</b>: If this is selected the message is transformed to SOAP 1.1.</li> <li>• <b>SOAP 1.2</b>: If this is selected the message is transformed to SOAP 1.2.</li> <li>• <b>Plain Old XML (POX)</b>: If this is selected the message is transformed to plain old XML format.</li> <li>• <b>Representational State Transfer (REST)</b> - If this is selected, the message is transformed to REST.</li> <li>• <b>GET</b>: If this is selected, the message is transformed to a HTTP Get Request.</li> </ul> </li> <li>• <b>Optimize</b>- Optimization for the message, which transfers binary data. The available values are as follows.           <ul style="list-style-type: none"> <li>• <b>Leave As-Is</b> - If this is selected, there will be no special optimization. The original message will be kept.</li> <li>• <b>SwA</b> - If this is selected, the message is optimized as a SwA (SOAP with Attachment) message.</li> <li>• <b>MTOM</b> - If this is selected, the message is optimized using a MTOM (message transmission optimization mechanism).</li> </ul> </li> </ul>
<b>Add Property</b>	<p>Click this link if you want to add properties to the endpoint. See <a href="#">Properties Reference</a> for further information on the available properties.</p>

## Template Endpoint

Template endpoints are created based on predefined [endpoint templates](#). Parameters of the endpoint template used as the target are copied to the endpoint configuration. However, you can make modifications by removing some of the copied parameters and/or adding new parameters.

A template endpoint can be defined following the instructions below.

**Note**

You can also configure a Template endpoint using XML. Click **Switch to source view** in the **Template Endpoint** page.

Template Endpoint
 Switch to source view

1. Log into the ESB Management Console. In the **Main** tab, click **Endpoints** to open the **Manage Endpoints** page.
2. Click **Template Endpoint** to open the **Template Endpoint** page.

Home > Manage > Service Bus > Endpoints > Template Endpoint Help

## Template Endpoint

**Template Endpoint** Switch to source view

Name (\$name) *	<input type="text"/>
Address (\$uri)	<input type="text"/> <span style="border: 1px solid #ccc; padding: 2px 5px; margin-left: 10px;">Test</span>
Target Template *	<input type="text"/>
Available Templates	<span style="border: 1px solid #ccc; padding: 2px 10px; border-radius: 5px; cursor: pointer;">Select From Templates</span>
<span style="color: green; font-size: 1.2em;">+</span> <a href="#">Add Parameter</a>	
<span style="border: 1px solid #ccc; padding: 2px 10px; border-radius: 5px; margin-right: 10px;">Save &amp; Close</span> <span style="border: 1px solid #ccc; padding: 2px 10px; border-radius: 5px; margin-right: 10px;">Save in Registry</span> <span style="border: 1px solid #ccc; padding: 2px 10px; border-radius: 5px;">Cancel</span>	

Enter the required values for the following parameters.

Parameter Name	Description											
<b>Name</b>	This is a <b>Default Endpoint</b> specific option. It defines the unique name of an endpoint.											
<b>Address</b>	The address of the endpoint (e.g., <a href="http://wso2.com">http://wso2.com</a> ).											
<b>Target Template</b>	The endpoint template which should be used to define the endpoint. This is specified by selecting one of the existing templates from the <b>Available Templates</b> list. The parameters to be included in the endpoint you are defining will be copied from the target template. The values for these parameters can be entered in the <b>Template Endpoint Parameters</b> section shown below. Click <b>Delete Parameter</b> to remove a parameter fetched from the target template. Click <b>Add Parameter</b> to add additional parameters.											
	<b>Template Endpoint Parameters</b> <table border="1" style="width: 100%; border-collapse: collapse; margin-top: 5px;"> <thead> <tr> <th>Name</th> <th>Value</th> <th>Action</th> </tr> </thead> <tbody> <tr> <td>\$name</td> <td><input type="text"/></td> <td><span style="color: orange;">Delete Parameter</span></td> </tr> <tr> <td>\$age</td> <td><input type="text"/></td> <td><span style="color: orange;">Delete Parameter</span></td> </tr> <tr> <td colspan="2" style="text-align: center;"><span style="color: green; font-size: 1.2em;">+</span> <a href="#">Add Parameter</a></td> </tr> </tbody> </table>	Name	Value	Action	\$name	<input type="text"/>	<span style="color: orange;">Delete Parameter</span>	\$age	<input type="text"/>	<span style="color: orange;">Delete Parameter</span>	<span style="color: green; font-size: 1.2em;">+</span> <a href="#">Add Parameter</a>	
Name	Value	Action										
\$name	<input type="text"/>	<span style="color: orange;">Delete Parameter</span>										
\$age	<input type="text"/>	<span style="color: orange;">Delete Parameter</span>										
<span style="color: green; font-size: 1.2em;">+</span> <a href="#">Add Parameter</a>												
<b>Parameters</b>	The parameters of the endpoint. For example, you can add <code>name</code> to add the parameter name, and <code>uri</code> to add the address etc.											

3. Click **Save and Close** to save the endpoint configuration in the Synapse configuration, or click **Save in Registry** to save it in the **Registry**.
4. If you clicked **Save in Registry**, the page will expand to display the **Save as Synapse Registry Entry** section. Select either **Configuration Registry** or **Governance Registry** as relevant to specify the registry in which the endpoint should be saved. Then define a key for the registry entry in the **Key** parameter.
5. Click **Save and Close** to save the configuration.

### Recipient List Endpoint

A Recipient List endpoint can contain multiple child endpoints or member elements. It routes cloned copies of messages to each child recipient.

This will assume that all immediate child endpoints are identical in state (state is replicated) or state is not maintained at those endpoints.

## XML Configuration

You can configure the Recipient List endpoint using XML. Click on the "Switch to source view" link in the "Recipient List Endpoint" page.

```
<recipientlist>
 <endpoint .../>+
</recipientlist>
```

## UI Configuration

- In the "Add Endpoint" list, click "Recipient List Endpoint" (See [Adding an Endpoint](#)). The "Recipient List Endpoint" page appears.

**Recipient List Endpoint**

Switch to source Switch to source view

Endpoint Name \*

root Add Child

**Recipient List Endpoint Properties**

Add Property

Save & Close Save in Registry Cancel

- In the "Endpoint Name" field enter a name for the endpoint.

## Recipient List Endpoint

Switch to source Switch to source view

Endpoint Name \*

root Add Child

Recipient List Endpoint Properties

Add Property

See the detailed information about the Endpoints properties in [Adding an Endpoint](#).

3. Click "Add Endpoint."

## Recipient List Endpoint

Switch to source Switch to source view

Endpoint Name \*

Add Child

---

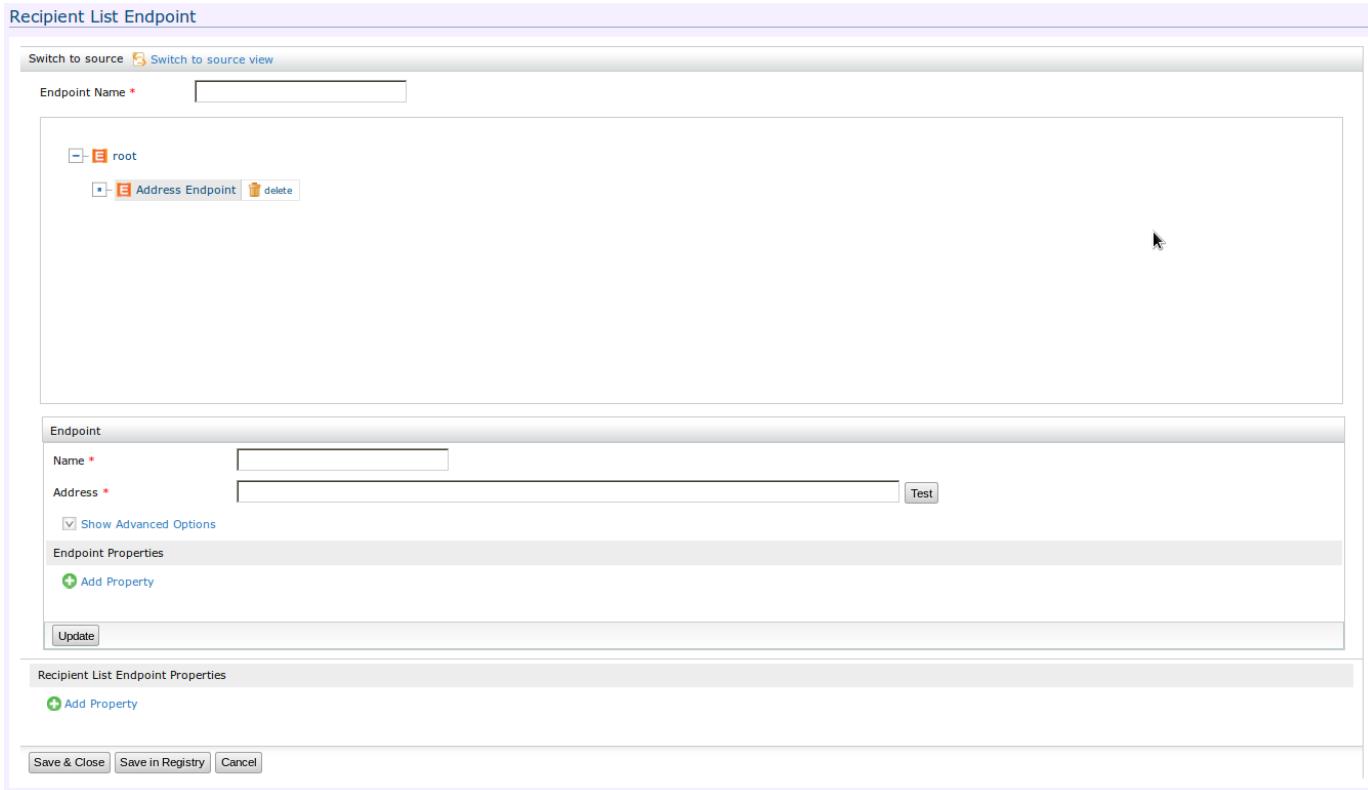
Recipient List Endpoint Properties

Add Property

4. A shortcut menu with the available endpoints appears.



5. Click on the endpoint you want to add as a Child Endpoint. A tab with the fields relevant to the selected endpoint appears.



6. Enter the details (see [Address Endpoint](#), [Default Endpoint](#), [WSDL Endpoint](#)). Click "Save."

**Save & Close** **Save in Registry** **Cancel**

## Working with Endpoints via WSO2 ESB Tooling

You can create a new endpoint or import an existing endpoint from an XML file, such as a Synapse configuration file using WSO2 ESB tooling.

You need to have WSO2 ESB tooling installed to create a new endpoint or to import an existing endpoint via ESB tooling. For instructions on installing WSO2 ESB tooling, see [Installing WSO2 ESB Tooling](#).

When you create an endpoint, you need to give it a name, at which point it appears in the **Defined Endpoint** section of the tool palette in WSO2 ESB tooling. To use a defined endpoint, simply drag and drop the endpoint from the palette to the sequence, proxy service, or other artifact where you want to reference it.

### Creating a new endpoint

Follow these steps to create a new endpoint. Alternatively, you can [import an existing endpoint](#).

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Endpoint** on the **Developer Studio Dashboard**.
3. Select **Create a New Endpoint** and click **Next**.
4. Type a unique name for the endpoint, and then specify the [type of endpoint](#) you are creating.
5. Do one of the following:
  - To save the endpoint in an existing ESB Config project in your workspace, click **Browse** and select that project.
  - To save the endpoint in a new ESB Config project, click **Create new Project** and create the new project.
  - To save the endpoint in a registry resource project, click **Make this a Dynamic Endpoint**, click the **Br**

**browse** button next to **Save endpoint in** and select the registry, and then select the registry (Configuration or Governance) where you want to save the endpoint. Lastly, type the endpoint name in the Registry Path box.

6. In the **Advanced Configuration** section (which changes based on the type of endpoint you are creating), specify any additional options you need for that endpoint type.
7. Click **Finish**. The endpoint is created in the `endpoints` folder under the ESB Config or Registry Resource project you specified, and the endpoint opens in the editor.

## Importing an endpoint

Follow these steps to import an existing endpoint from an XML file such as a Synapse configuration file, into an ESB Config project. Alternatively, you can [create a new endpoint](#).

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Endpoint** on the **Developer Studio Dashboard**.
3. Select **Import Endpoint** and click **Next**.
4. Specify the endpoint file by typing its full path name or clicking **Browse** and navigating to the file.
5. In the **Save Endpoint In** field, specify an existing ESB Config project in your workspace where you want to save the endpoint, or click **Create new Project** to create a new ESB Config project and save the endpoint there.
6. In the **Advanced Configuration** section, select the endpoints you want to import.
7. Click **Finish**. The endpoints you selected are created in the `endpoints` folder under the ESB Config project you specified, and the first endpoint opens in the editor.

## Working with Endpoints via the Management Console

You can add, edit and delete endpoints via the ESB Management Console. You can also enable/disable statistics for endpoints as well as switch on/switch off an endpoint based on your requirement via the Management console.

See the following topics for information on how to manage endpoints via the Management Console:

- [Adding an endpoint](#)
- [Editing an endpoint](#)
- [Deleting an endpoint](#)
- [Enabling/Disabling statistics](#)
- [Switching on/off](#)

### Adding an endpoint

An endpoint can be added as follows.

1. Open the ESB management console.
2. Click **Endpoints** under **Service Bus** in the left navigator to open the **Manage Endpoints** page.
3. Click on the **Add Endpoint** tab.
4. Click on the endpoint type you want to add. Available types are as follows.
  - **Address Endpoint** - Defines the direct URL of the service.
  - **Default Endpoint** - Defines additional configuration for the default target.
  - **Failover Group** - Defines the endpoints that the service will try to connect to in case of a failure. This will take place in a round robin manner.
  - **HTTP Endpoint** - Defines a URI template based REST service endpoint.
  - **Load Balance Endpoint** - Defines groups of endpoints for replicated services. The incoming requests will be directed to these endpoints in a round robin manner. These endpoints automatically handle the fail-over cases as well.
  - **Recipient List Group** - Defines the list of endpoints a message will be routed to.
  - **Template Endpoint** - Allows to create a custom Endpoint template.
  - **WSDL Endpoint** - Defines the WSDL, Service and Port.

Enter the required information for the endpoint depending on the type you selected. Click on the relevant link above to find detailed information on how to configure the endpoint you selected.

5. Click **Show Advanced Options** if you want to configure advanced settings for the endpoint. Available advanced options are as follows.

Field Name	Description
Suspend Error Codes	This parameter allows you to select one or more error codes from the List of Values. If any of the selected errors is received from the endpoint, the endpoint will be suspended.
Initial Duration (in milliseconds))	The time duration for which the endpoint will be suspended, when one or more suspend error codes are received from it for the first time.
Max Duration (Millis)	The maximum time duration for which the endpoint is suspended when suspend error codes are received from it.
Factor	The duration to suspend can vary from the first time suspension to the subsequent time. The factor value decides the suspense duration variance between subsequent suspensions.
On Timeout Error codes	A list of error codes. If these error codes are received from the endpoint, the request will be subjected to a timeout.
Retry	The number of re-tries in case of a timeout, caused by the above listed error codes.
Retry Delay(in milliseconds)	The delay between retries in milliseconds.
Timeout Action	The action to be done at a timeout situation. You can select from: <ul style="list-style-type: none"> <li>• <b>Never Timeout</b></li> <li>• <b>Discard Message</b></li> <li>• <b>Execute Fault Sequence</b></li> </ul>
Timeout Duration (in milliseconds)	The duration in milliseconds before considering a request to be subjected to a time-out.
WS-Addressing	Adds WS-Addressing headers to the endpoint.
Separate Listener	The listener to the response will be a separate transport stream from the caller.
WS-Security	Adds WS-Security features as described in a policy key (referring to a registry location).
Non Retry Error Codes	When a child endpoint of a <a href="#">failover endpoint</a> or <a href="#">load-balance endpoint</a> fails for one of the error codes specified here, the child endpoint will be marked for suspension and the request will not be sent to the next endpoint in the group.
Retry Error Codes	When adding a child endpoint to a <a href="#">failover endpoint</a> or <a href="#">load-balance endpoint</a> , you can specify the error codes that trigger this node to be <a href="#">retried</a> instead of suspended when that error is encountered. This is useful when you know that certain errors are transient and that the node will become available again shortly. Note that if you specify an error code as a Retry code on one node in the group but specify that same code as a Non Retry error code on another node in the group, it will be treated as a Non Retry error code for all nodes in the group.

6. Click **Add Property** if you want to add any properties to an endpoint. The page will expand to display the following parameters.

Endpoint Properties			
<span style="color: green;">+</span> Add Property			
Name	Value	Scope	Action
<input type="text"/>	<input type="text"/>	Synapse	<span style="color: orange;">Delete</span>

Parameter Name	Description
Name	The name of the endpoint property.
Value	The value of the endpoint property.
Scope	The scope of the property. Possible values are as follows. <ul style="list-style-type: none"> <li>• Synapse</li> <li>• Transport</li> <li>• Axis2</li> <li>• axis2-client</li> </ul> See <a href="#">XPath Extension Functions</a> for more information about these scopes.
Action	This parameter allows you to delete a property.

See [Properties Reference](#) for more information on properties that you can add to the endpoint.

- Click **Save & Close** to save the endpoint in the synapse configuration, or click **Save in Registry** if you want to save it as a dynamic endpoint in the [Registry](#).

#### **Editing an endpoint**

After you [add an endpoint](#), you can edit its attributes or semantics that are followed when communicating with that endpoint. An endpoint can be edited as follows.

- Open the ESB management console.
- Click **Endpoints** under **Service Bus** in the left navigator to open the **Manage Endpoints** page.
- Click on the **Defined Endpoints** tab if the endpoint you want to edit is saved in the synapse configuration. Click on the **Dynamic Endpoints** tab if the endpoint you want to edit is saved in the [Registry](#). The existing endpoints saved in the synapse configuration/registry will be displayed.
- Click **Edit** for the required endpoint. Make the required modifications. See [Adding an Endpoint](#) for more information.
- Click **Save & Close** to save the endpoint in the synapse configuration, or click **Save in Registry** if you want to save it as a dynamic endpoint in the [Registry](#).

#### **Deleting an endpoint**

If you no longer need a particular endpoint that you added, you can delete the endpoint as follows.

- Open the ESB management console.
- Click **Endpoints** under **Service Bus** in the left navigator to open the **Manage Endpoints** page.
- Click on the **Defined Endpoints** tab if the endpoint you want to delete is saved in the synapse configuration. Click on the **Dynamic Endpoints** tab if the endpoint you want to delete is saved in the [Registry](#). The existing endpoints saved in the synapse configuration/registry will be displayed.
- Click **Delete** for the required endpoint. Then click **Yes** in the message which appears to confirm whether you want to proceed deleting the endpoint.

#### **Enabling/Disabling statistics**

Statistics can be enabled or disabled as required for [Address endpoints](#) and [WSDL endpoints](#). Note that this action can be carried out only for *defined* endpoints, which are endpoints saved in the Synapse configuration. You cannot enable statistics for *dynamic* endpoints, which are saved in the [Registry](#).

Statistics can be enabled/disabled as follows.

1. Open the ESB management console.
2. Click **Endpoints** under **Service Bus** in the left navigator to open the **Manage Endpoints** page.
3. Click on the **Defined Endpoints** tab. The **Enable Statistics** link will be displayed for endpoints with statistics currently disabled. The **Disable Statistics** link will be displayed for endpoints with statistics currently enabled.
4. If you want to enable statistics for an endpoint, click **Enable Statistics** in the relevant row. If you want to disable statistics for an endpoint, click **Disable Statistics** in the relevant row.

#### **Switching on/off**

Having many active endpoints may result in heavy traffic, and thereby cause performance degradation. In order to prevent that, you can temporarily switch off some endpoints and switch them on again when you need them in an active state. Note that this action can be carried out only for *defined* endpoints, which are endpoints saved in the Synapse configuration. You cannot enable statistics for *dynamic* endpoints, which are saved in the [Registry](#).

An endpoint can be switched on/off as follows.

1. Open the ESB management console.
2. Click **Endpoints** under **Service Bus** in the left navigator to open the **Manage Endpoints** page.
3. Click on the **Defined Endpoints** tab. The **Switch On** link will be displayed for endpoints which are currently switched off. The **Switch Off** link will be displayed for endpoints which are currently switched on.
4. If you want to activate an endpoint, click **Switch On** in the relevant row. If you want to deactivate an endpoint, click **Switch Off** in the relevant row.

#### **Endpoint Error Handling**

This page describes error handling for endpoints. It contains the following sections:

- [Overview](#)
- [Endpoint states](#)
  - Active
  - Timeout
  - Suspended
- [Configuring leaf endpoints](#)
  - "Timeout" settings
  - "MarkForSuspension" settings
  - 'suspendOnFailure' settings
- [Disabling endpoint suspension](#)
- [Configuring retry](#)
- [Configuring the failover endpoint](#)

#### **Overview**

The last step of a message processing inside WSO2 Enterprise Service Bus is to send the message to a service provider (see also [Mediating Messages](#)) by sending the message to a listening service [endpoint](#). During this process, transport errors can occur. For example, the connection might time out, or it might be closed by the actual service. Therefore, endpoint error handling is a key part of any successful ESB deployment.

Messages can fail or be lost due to various reasons in a real TCP network. When an error occurs, if the ESB is not configured to accept the error, it will mark the endpoint as failed, which leads to a message failure. By default, the endpoint is marked as failed for quite a long time, and due to this error, subsequent messages can get lost.

To avoid lost messages, you configure error handling at the endpoint level. You should also run a few long-running load tests to discover errors and fine-tune the endpoint configurations for errors that can occur intermittently due to various reasons.

For information on general error handling and error codes in the ESB, see [Error Handling](#).

#### **Endpoint states**

At any given time, the state of the endpoint can be one of the following:

State	Description
Active	Endpoint is running and handling requests.
Timeout	Endpoint encountered an error but can still send and receive messages. If it continues to encounter errors, it will be suspended.
Suspended	Endpoint encountered errors and cannot send or receive messages. Incoming messages to a suspended endpoint result in a fault.
OFF	Endpoint is not active. To put an endpoint into the OFF state, or to move it from OFF to Active, you must use JMX.

### Active

When WSO2 Enterprise Service Bus starts, endpoints are in the "Active" state and ready to handle messages. If the user does not put the endpoint into the OFF state, it will be in the "Active" state until an error occurs.

The endpoint can be configured to stay in the "Active" state or to go to "Timeout" or "Suspended" based on the error codes you configure for those states. When an error occurs, the endpoint checks to see whether it is a "Timeout" error first, and if not, it checks to see whether it is a "Suspended" error. If the error is not defined for either "Timeout" or "Suspended," the error will be ignored and the endpoint will stay Active.

### Timeout

When an endpoint is in the "Timeout" state, it will continue to attempt to receive messages until one message succeeds or the maximum retry setting has been reached. If the maximum is reached at which point the endpoint is marked as "Suspended." If one message succeeds, the endpoint is marked as "Active."

For example, let's assume the number of retries is set to 3. When an error occurs and the endpoint is set to the "Timeout" state, the ESB can try to send up to three more messages to the endpoint. If the next three messages sent to this endpoint result in an error, the endpoint is put in the "Suspended" state. If one of the messages succeeds before the retry maximum is met, the endpoint will be marked as "Active."

### Suspended

A "Suspended" endpoint cannot send or receive messages. When an endpoint is put into this state, the ESB waits until after an initial duration has elapsed (default is 30 seconds) before attempting to send messages to this endpoint again. If the message succeeds, the endpoint is marked as "Active." If the next message fails, the endpoint is marked as "Suspended" or "Timeout" depending on the error, and the ESB waits before retrying messages using the following formula:

```
Min(current suspension duration * progressionFactor, maximumDuration)
```

You configure the initial suspension duration, progression factor, and maximum duration as part of the [suspendOnFailure settings](#). On each retry, the suspension duration increases, up to the maximum duration.

### Configuring leaf endpoints

The following is the configuration for the [address endpoint](#). Since we all are only interested in error configurations, the same applies for WSDL endpoints as well. The error handling configuration are as follows:

- [Timeout settings](#)
- [MarkForSuspension settings](#)
- [suspendOnFailure settings](#)

```

<address uri="endpoint address" [format="soap11|soap12|pox|get"]
[optimize="mtom|swa"] [encoding="charset encoding"]
[statistics="enable|disable"] [trace="enable|disable"]>
<enableRM [policy="key"]/>?
 <enableSec [policy="key"]/>?
 <enableAddressing [version="final|submission"]>
 [separateListener="true|false"]/>?

 <timeout>
 <duration>timeout duration in seconds</duration>
 <responseAction>discard|fault</responseAction>
 </timeout>?

 <markForSuspension>
 [<errorCodes>xxx,yyy</errorCodes>]
 <retriesBeforeSuspension>m</retriesBeforeSuspension>
 <retryDelay>d</retryDelay>
 </markForSuspension>

 <suspendOnFailure>
 [<errorCodes>xxx,yyy</errorCodes>]
 <initialDuration>n</initialDuration>
 <progressionFactor>r</progressionFactor>
 <maximumDuration>l</maximumDuration>
 </suspendOnFailure>
</address>

```

### **"Timeout" settings**

Name	Values	Default	Description
duration	Miliseconds/ XPATH expression	60000	Connection timeout interval. If the remote endpoint does not respond in this time, it will be marked as "Timeout." This can be defined as a static value or as a <a href="#">dynamic value</a> .
responseAction	discard, fault, none	none	When a response comes to a timed out request, specifies whether to discard it or invoke the fault handler. If none, the endpoint remains in the "Active" state.

### **"MarkForSuspension" settings**

Name	Values	Default	Description
errorCodes	Comma separated list of <a href="#">error codes</a>	101504, 101505	Errors that put the endpoint into the "Timeout" state. If no error codes are specified, the "HTTP Connection Closed" and "HTTP Connection Timeout" errors are considered "Timeout" errors, and all other errors put the endpoint into the "Suspended" state.
retriesBeforeSuspension	Integer	0	In the "Timeout" state this number of requests minus one can be tried and fail before the endpoint is marked as "Suspended". This setting is per endpoint, not per message, so several messages can be tried in parallel and fail and the remaining retries for that endpoint will be reduced.

retryDelay	milliseconds	0	The time to wait between the last retry attempt and the next retry.
------------	--------------	---	---------------------------------------------------------------------

### 'suspendOnFailure' settings

Name	Values	Default	Description
errorCodes	Comma separated list of error codes	All the errors except the errors specified in markForSuspension	Errors that send the endpoint into the "Suspended" state.
initialDuration	milliseconds	30000	After an endpoint gets "Suspended," it will wait for this amount of time before trying to send the messages coming to it. All the messages coming during this time period will result in fault sequence activation.
progressionFactor	Integer	1	The endpoint will try to send the messages after the initialDuration. If it still fails, the next duration is calculated as:  Min(current suspension duration * progressionFactor, maximumDuration)
maximumDuration	milliseconds	Long.MAX_VALUE	Upper bound of retry duration.

### Sample Configuration:

```

<endpoint name="Sample_First" statistics="enable" >
 <address uri="http://localhost/myendpoint" statistics="enable" trace="disable">
 <timeout>
 <duration>60000</duration>
 </timeout>

 <markForSuspension>
 <errorCodes>101504, 101505</errorCodes>
 <retriesBeforeSuspension>3</retriesBeforeSuspension>
 <retryDelay>1</retryDelay>
 </markForSuspension>

 <suspendOnFailure>
 <errorCodes>101500, 101501, 101506, 101507, 101508</errorCodes>
 <initialDuration>1000</initialDuration>
 <progressionFactor>2</progressionFactor>
 <maximumDuration>60000</maximumDuration>
 </suspendOnFailure>
 </address>
</endpoint>

```

In this example, the errors 101504 and 101505 move the endpoint into the "Timeout" state. At that point, three requests can fail for one of these errors before the endpoint is moved into the "Suspended" state. Additionally, errors 101500, 101501, 101506, 101507, and 101508 will put the endpoint directly into the "Suspended" state. The error 101503 is not listed, so if a 101503 error occurs, the endpoint will remain in the "Active" state.

When the endpoint is first suspended, the retry happens after one second. Because the progression factor is 2, the

next suspension duration before retry is two seconds, then four seconds, then eight, and so on until it gets to sixty seconds, which is the maximum duration we have configured. At this point, all subsequent suspension periods will be sixty seconds until the endpoint succeeds and is back in the Active state, at which point the initial duration will be used on subsequent suspensions.

### Sample Configuration for Endpoint Dynamic Timeout:

Let's look at a sample configuration where you have dynamic timeout for the endpoint.

In this, the timeout value is defined using a [Property mediator](#) outside the endpoint configuration. The timeout parameter in the endpoint configuration is then evaluated against an XPATH expression that is used to reference and read the timeout value. Using this timeout values can be configured without having to change the endpoint configuration.

```
<sequence name="dynamic_sequence" xmlns="http://ws.apache.org/ns/synapse">
 <property name="timeout" scope="default" type="INTEGER" value="20000">
 <send>
 <endpoint>
 <address uri="http://localhost/myendpoint">
 <timeout>
 <duration>{get-property('timeout')}</duration>
 <responseAction>discard</responseAction>
 </timeout>
 </endpoint>
 </send>
 </sequence>
```

You also have the option of defining a dynamic timeout for the endpoint as a local entry:

```
<localEntry key="timeout"><![CDATA[20000]]>
 <description/>
</localEntry>
```

For more information about error codes, see [Error Codes](#).

### ***Disabling endpoint suspension***

If you do not want the endpoint to be suspended at all, you can configure the Timeout, MarkForSuspension and suspendOnFailure settings as shown in the following example.

```

<endpoint name="NoSuspendEndpoint">
 <address uri="http://localhost:9000/services/SimpleStockQuoteService">
 <timeout>
 <duration>30000</duration>
 <responseAction>fault</responseAction>
 </timeout>
 <suspendOnFailure>
 <errorCodes>-1</errorCodes>
 <initialDuration>0</initialDuration>
 <progressionFactor>1.0</progressionFactor>
 <maximumDuration>0</maximumDuration>
 </suspendOnFailure>
 <markForSuspension>
 <errorCodes>-1</errorCodes>
 </markForSuspension>
 </address>
</endpoint>

```

### **Configuring retry**

You can configure the ESB to enable or disable retry for an endpoint when a specific error code occurs. For example:

```

<endpoint>
 <address uri="http://localhost:9001/services/LBService1">
 <retryConfig>
 <disabledErrorCodes>101503</disabledErrorCodes>
 </retryConfig>
 </address>
</endpoint>
<endpoint>
 <address uri="http://localhost:9002/services/LBService1">
 <retryConfig>
 <enabledErrorCodes>101503</enabledErrorCodes>
 </retryConfig>
 </address>
</endpoint>

```

In this example, if the error code 101503 occurs when trying to connect to the first endpoint, the endpoint is not retried, whereas in the second endpoint, the endpoint is always retried if error code 101503 occurs. You can specify enabled or disabled error codes (but not both) for a given endpoint.

### **Configuring the failover endpoint**

With leaf endpoints, if an error occurs during a message transmission process, that message will be lost. The failed message will not be retried again. These errors occur very rarely, but still message failures can occur. With some applications these message losses are acceptable, but if even rare message failures are not acceptable, use the fail over endpoint.

Here is the configuration for failover endpoints. At the configuration level, a failover is a logical grouping of one or more leaf endpoints.

```
<failover>
 <endpoint .../>+
</failover>
```

When a message comes to the **Failover** state, it will go through its list of endpoints to pick the first one in **Active** or **Timeout** state. Then it will send the message using that particular endpoint. If an error occurs while sending the message, the failover will go through the endpoint list again from the beginning and will try to send the message using the first endpoint.

Some errors put the endpoint into **Timeout** and some keep the endpoint in the **Active** state. In these cases, the retry can happen using the same endpoint. If the failure occurs with the first endpoint within the failover group and this error does not put the endpoint into **Suspended** state, the retry will happen using the same endpoint.

Failover gives priority to the first endpoint that is not in the **Suspended** state. So it will send the message through the first endpoint in the failover group, as long as it is not suspended. When the first endpoint is suspended, it will send the requests using the second endpoint. When the first endpoint becomes ready to send again, it will try again on the first endpoint, even though the second endpoint is still active.

If there is only one service endpoint and the message failure is not tolerable, failovers are possible with a single endpoint.

A sample failover with one address endpoint:

```
<endpoint name="SampleFailover">
 <failover>
 <endpoint name="Sample_First" statistics="enable" >
 <address uri="http://localhost/myendpoint" statistics="enable"
trace="disable">
 <timeout>
 <duration>60000</duration>
 </timeout>

 <markForSuspension>
 <errorCodes>101504, 101505, 101500</errorCodes>
 <retriesBeforeSuspension>3</retriesBeforeSuspension>
 <retryDelay>1</retryDelay>
 </markForSuspension>

 <suspendOnFailure>
 <initialDuration>1000</initialDuration>
 <progressionFactor>2</progressionFactor>
 <maximumDuration>64000</maximumDuration>
 </suspendOnFailure>
 </address>
 </endpoint>
 </failover>
</endpoints>
```

Here the **Sample\_First** endpoint is marked as **Timeout** if a connection times out, closes, or sends IO errors. For all the other errors, it will be marked as **Suspended**. When this error occurs, the failover will retry using the first non suspended endpoint. In this case, it is the same endpoint (**Sample\_First**). It will retry until the retry count becomes 0. The retry happens in parallel. Since messages come to this endpoint using many threads, the same message may not be retried three times. Another message may fail and can reduce the retry count.

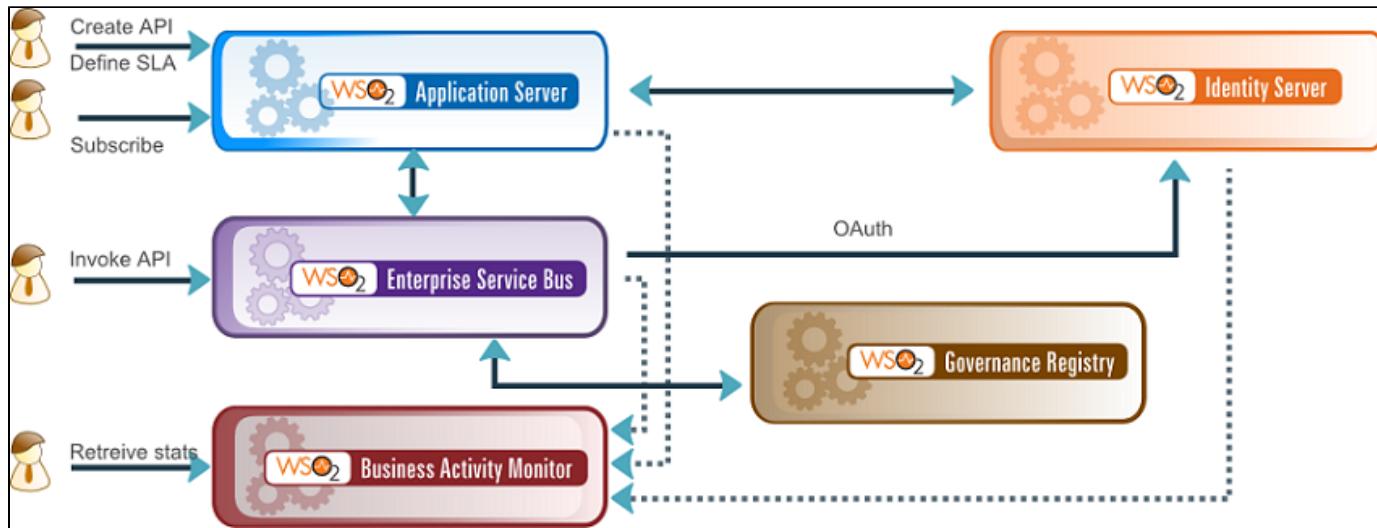
The retry count is per endpoint, not per message.

In this configuration, we assume that these errors are rare and if they happen once in a while, it is OK to retry again. If they happen frequently and continuously, it means that it requires immediate attention to get it back to normal state.

## Configuring Endpoints using REST APIs

In addition to exposing RESTful interfaces and mediating RESTful invocations by mapping REST concepts to SOAP via proxy services, you can use the WSO2 ESB REST API to configure REST endpoints in the ESB by directly specifying HTTP verbs, URL patterns, URI templates, HTTP media types, and other related headers. You can define REST APIs and associated resources in the ESB by combining REST APIs with mediation features provided by the underlying messaging framework.

As depicted in the diagram below, core API management features such as API key management, user management, security, and SLA monitoring are all provided by the WSO2 applications.

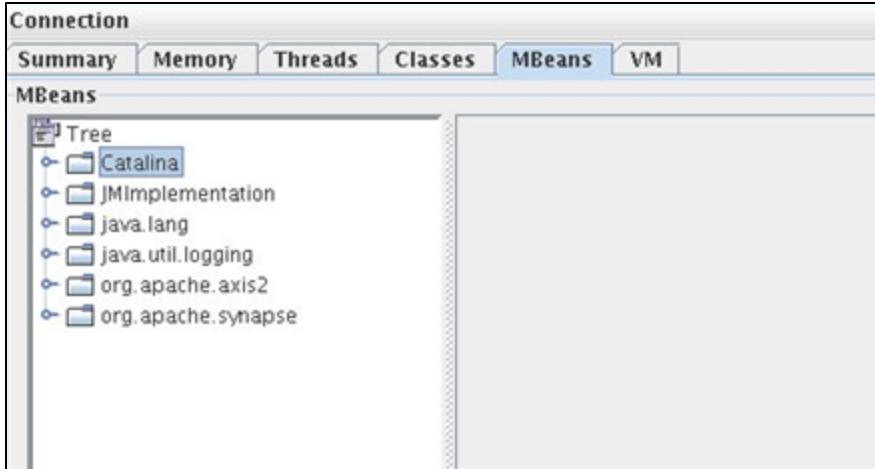


For more information, see [Using REST with APIs](#) and [Sample 800: Introduction to REST API](#).

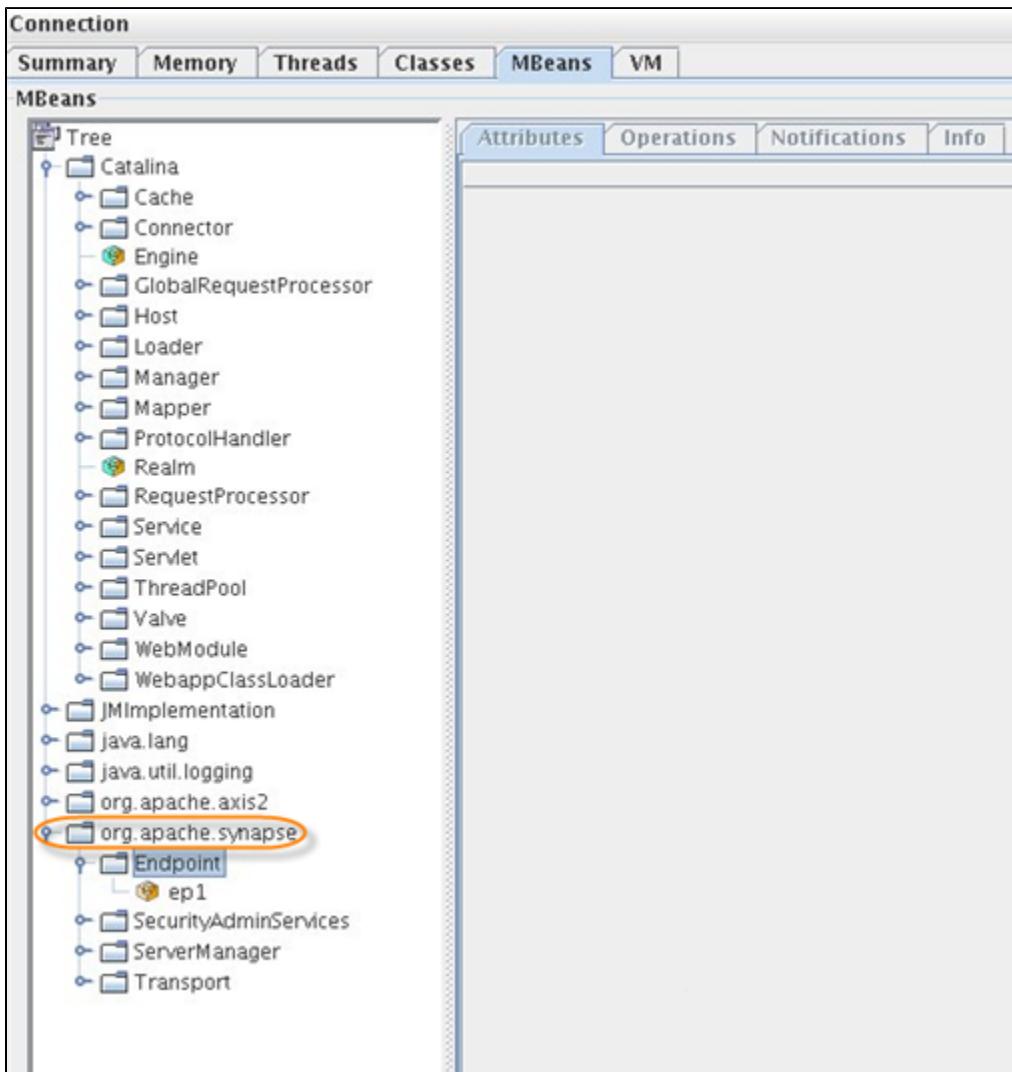
## Using Jconsole to Manage State Transition

The **jconsole** tool is a JMX-compliant graphical tool for monitoring a Java virtual machine. If you have installed java, you will automatically get this tool. We will add an endpoint (name "ep1") and monitor the states transition between states in ep1.

1. Add an address endpoint called "ep1." See [Adding an Endpoint](#).
2. Start jconsole by typing `jconsole` at your terminal, if you encounter any problem, please, consult your operating system guide. You will see the following console once you select MBeans tab.



3. Go to org.apache.synapse menu from left hand and expand it. You will see the following figure with the endpoint "ep1" that we just added.



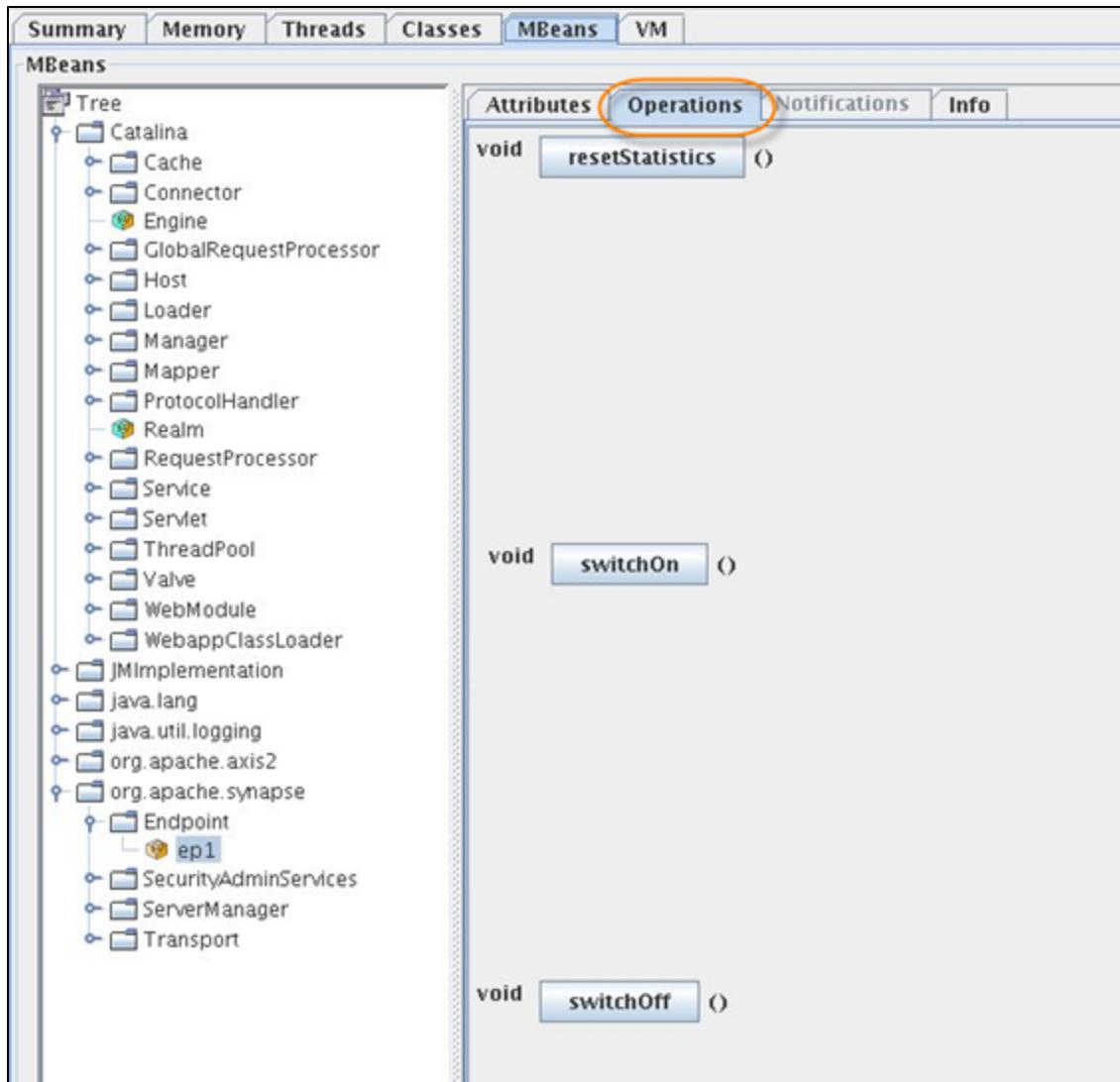
4. Select the "ep1" endpoint and you will see the information related to this endpoint.

## Note

The state is given as active as the first entry and value is set to true. The endpoint is in active state.

Attributes	Operations	Notifications	Info
Name			
Active			true
ActiveChildren			0
AvgSizeReceived			0.0
AvgSizeSent			0.0
BytesReceived			0
BytesSent			0
FaultsReceiving			0
FaultsSending			0
LastResetTime			1245649148751
MaxSizeReceived			0
MaxSizeSent			0
MessagesReceived			0
MessagesSent			0
MetricsWindow			2065870
MinSizeReceived			0
MinSizeSent			0
ReadyChildren			0
ReceivingFaultTable			{}
ResponseCodeTable			{}
SendingFaultTable			{}
Suspended			false
SwitchedOff			false
Timedout			false
TimeoutsReceiving			0
TimeoutsSending			0
TotalChildren			0

5. Go to the "Operations" tab and switch off the endpoint. You will see a message saying the operation was successful.



You will also see a INFO level log message saying that the endpoint was switched off manually.

```
[2009-06-23 12:18:21,149] INFO - EndpointContext Manually switching off endpoint : ep1
```

6. Next again switch back to the "Attributes" tab and see the state of the endpoint after refreshing it. You will see endpoint is inactive and the endpoint is in switch off state. At this point, if you try to use a client which use this endpoint, you will see it fails since the endpoint is down. You can use a similar approach activate the endpoint back.

Attributes	Operations	Notifications	Info
Name			
Active			false
ActiveChildren			0
AvgSizeReceived			0.0
AvgSizeSent			0.0
BytesReceived			0
BytesSent			0
FaultsReceiving			0
FaultsSending			0
LastResetTime			1245649148751
MaxSizeReceived			0
MaxSizeSent			0
MessagesReceived			0
MessagesSent			0
MetricsWindow			2495825
MinSizeReceived			0
MinSizeSent			0
ReadyChildren			0
ReceivingFaultTable			{}
ResponseCodeTable			{}
SendingFaultTable			{}
Suspended			false
SwitchedOff			true
Timedout			false
TimeoutsReceiving			0
TimeoutsSending			0
TotalChildren			0

## Working with Web Services

In **service mediation**, the ESB exposes an [endpoint](#) that accepts messages from clients on behalf of an external Web service. Typically, [proxy services](#) are used to receive and *mediate* these messages before they are proxied to the actual Web service. This allows WSO2 ESB to expose a Web service already available in one [transport](#) over a different transport, or expose a Web service that uses one schema or WSDL as a Web service that uses a different schema or WSDL.

WSO2 ESB provides many tools to manage Web services that are deployed.

See the following topics for more information on working with Web services in WSO2 ESB:

- [Accessing Services](#)
- [Managing Web Services](#)
- [Working with Topics and Events](#)
- [Per-Service Logs in WSO2 ESB](#)

For a service chaining example, see the tutorial [Exposing Several Services as a Single Service](#).

### Accessing Services

You can view all deployed services via the **Deployed Services** screen on the ESB Management Console.

#### To view all Deployed Services

- Click the **Main** tab on the Management Console and then go to **Manage -> Service Bus** and click **List**. The **Deployed Services** screen appears.

The screenshot shows the 'Deployed Services' screen. At the top left, there's an 'Information Bar' containing the text '6 Deployed Service Group(s) 7 Faulty Service Group(s)'. Below it is a 'List Filter' with a dropdown menu set to 'ALL' and a search input field. To the right is a 'Toolbar' with buttons for selecting all services ('Select all in this page | Select none') and deleting selected services ('Delete'). The main area is titled 'The "Services" List' and contains a table with two columns: 'Service Groups' and 'Services'. The table rows are:

Service Groups	Services
<input type="checkbox"/> echo	echo  WSDL1.1  WSDL2.0
<input type="checkbox"/> Event1	Event1  WSDL1.1  WSDL2.0
org.wso2.carbon.sts	wso2carbon-sts  WSDL1.1  WSDL2.0
org.wso2.carbon.xkms	XKMS  WSDL1.1  WSDL2.0
<input type="checkbox"/> Proxy	Proxy  WSDL1.1  WSDL2.0
<input type="checkbox"/> version	version  WSDL1.1  WSDL2.0

The **Deployed Services** screen displays information on all deployed services and also allows you to manage deployed services.

- **Information bar** - Displays the number of deployed service groups and also displays a link to faulty service groups if there are any.

### Note

Click **deployed service group(s)** to have a look at service groups that have been deployed. Click **faulty service group(s)** to have a look at the service groups that were not deployed and delete them if necessary.

- **List filter** - Displays the filter criteria that you can use to search and filter the displayed services. You can search and filter the services by either specifying the service type or by specifying the service name.
- **Toolbar** - Allows you to select all service groups and delete them. (If you need to select individual service groups and delete them, you should select the required check boxes)
- **Services panel**- Displays the following information related to each service.
  - **Check boxes** - Allows you to select one or more services to delete.
  - **Service Name** - Displays the name of a service.
  - **Service Type**- Displays the service type.

.	axis2
.	sts
.	proxy
.	eventing

- **Security** - Displays whether the service is secured or not.
- **WSDL Files** - Allows you to view the WSDL 1.1 and WSDL 2.0 files of the service.
- **Try It** - Allows you to invoke the service and validate the output. For more information, see the [Try It](#) tool.

All the services deployed to WSO2 ESB can be accessed via the `http(s) :<esb>8280/services/` URL by default. If you want to disable this, you need to add the following property to the `<ESB_HOME>/repository/conf/nhttp.properties` file.

```
http.block_service_list=true
```

For more information, see [Managing Web Services](#), [Working with Web Services](#), and [Adding a Proxy Service](#).

## Managing Web Services

WSO2 ESB provides many tools to manage web services that are deployed. When you deploy a single service in a service archive, the archive file name will always be used as the service file name, unless you have a name attributed to the service file. If the name of the service archive file is Test.aar, then the name of the service will be Test. You can view the service dashboard of a particular service via the ESB Management Console.

### To view the Service Dashboard

1. Click the **Main** tab on the Management Console and then go to **Manage -> Service Bus** and click **List**. This will take you to the **Deployed Services** screen.
2. Click the name of the required service to view its dashboard. For more information, see [Accessing Services](#).

The screenshot shows the 'Deployed Services' section of the management console. At the top, it displays '6 active services. 6 deployed service group(s.)'. Below this is a search bar with 'Service Type ALL' and a search icon. Underneath is a toolbar with 'Select all in this page | Select none' and a 'Delete' button. The main area is a table titled 'Services' with columns: a checkbox, the service name, its type (e.g., axis2, proxy, sts), security status (Unsecured), and WSDL versions (WSDL1.1, WSDL2.0). The 'echo' service is highlighted with a red circle around its name in the first column.

Services						
<input type="checkbox"/>	Service Name	Type	Security	WSDL1.1	WSDL2.0	Try this service
<input type="checkbox"/>	echo	axis2	Unsecured	WSDL1.1	WSDL2.0	
<input type="checkbox"/>	Proxy Service	proxy	Unsecured	WSDL1.1	WSDL2.0	
<input type="checkbox"/>	PS2	proxy	Unsecured	WSDL1.1	WSDL2.0	
<input type="checkbox"/>	Version	axis2	Unsecured	WSDL1.1	WSDL2.0	
	wso2carbon-sts	sts	Unsecured	WSDL1.1	WSDL2.0	
	XKMS	axis2	Unsecured	WSDL1.1	WSDL2.0	

You will see the following panels on the service dashboard:

- [Service Details](#)
- [Client Operations](#)
- [Statistics](#)

The following section describes the properties displayed on each panel of the service dashboard.

### Service Details

The **Service Details** panel displays the following details of a service:

- **Service Name** - The name of the service.
- **Service Description** - The description of the service.
- **Service Group Name** - The name of the group that the service belongs to. For more information on service groups, see [Managing Service Groups](#).
- **Deployment Scope** - The service session scope to deploy the service.
- **Service Type** - The service type. For example, Axis2, Proxy, STS.

### Client Operations

The **Client Operations** panel displays the following operations:

- Try this service
- Generate Client
- WSDL 1.1/2.0

The **Client Operations** panel also displays information on the endpoints of a service.

The WSO2 server is set to the following two endpoints by default.

- <https://localhost:8243/services/echo>
- <http://localhost:8280/services/echo>

### Try This Service

When you click **Try this service**, it will direct you to a page that displays all the operations available for the service. For operations that take arguments you will see primitive argument type fields. The values specified in these fields will be passed to the operations. For no-argument operations you will only see the button that has the same name as the operation.

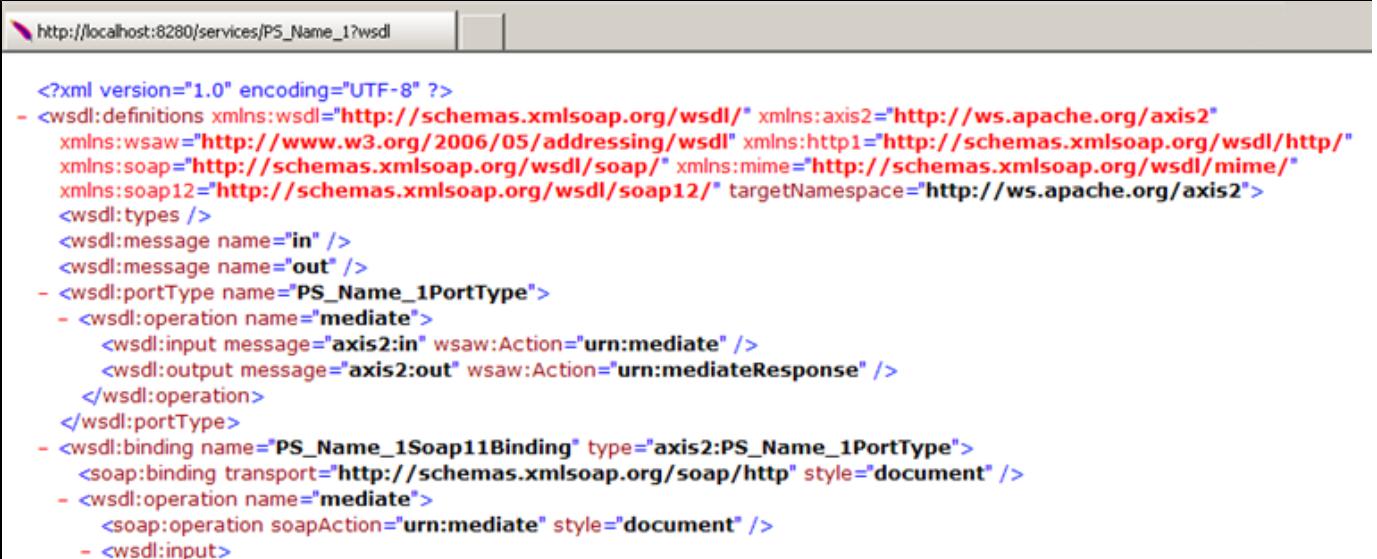
Click **Send** to invoke the service operation. The return value immediately appears in the response text area. For more information, see the [Try It tool](#).

### Generate Client

When you click **Generate Client**, it will open the [WSDL2Java](#) page so that you can easily generate the client for your service. All the required WSDL2 code options are available on the [WSDL2Java](#) page.

### WSDL 1.1 and WSDL 2.0

Click **WSDL 1.1** or **WSDL 2.0** to open the relevant WSDL file of your service.



```

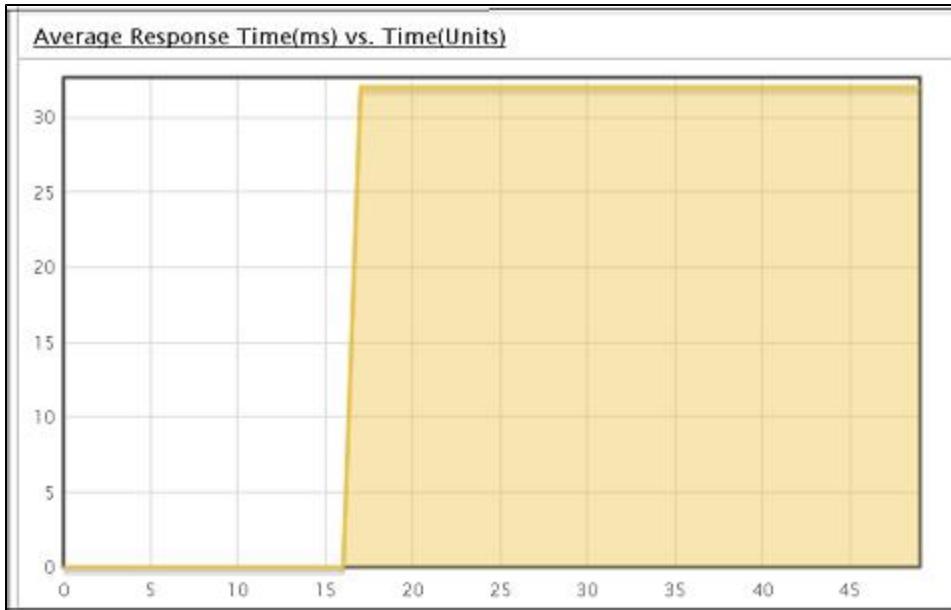
<?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:axis2="http://ws.apache.org/axis2"
 xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl" xmlns:http1="http://schemas.xmlsoap.org/wsdl/http/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
 xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" targetNamespace="http://ws.apache.org/axis2">
 <wsdl:types />
 <wsdl:message name="in" />
 <wsdl:message name="out" />
- <wsdl:portType name="PS_Name_1PortType">
 - <wsdl:operation name="mediate">
 <wsdl:input message="axis2:in" wsaw:Action="urn:mediate" />
 <wsdl:output message="axis2:out" wsaw:Action="urn:mediateResponse" />
 </wsdl:operation>
</wsdl:portType>
- <wsdl:binding name="PS_Name_1Soap11Binding" type="axis2:PS_Name_1PortType">
 <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
 - <wsdl:operation name="mediate">
 <soap:operation soapAction="urn:mediate" style="document" />
 - <wsdl:input>
```

### Statistics

The **Statistics** panel displays the following statistics related to the service:

- **Request Count**
- **Response Count**
- **Fault Count**
- **Maximum Response Time**
- **Minimum Response Time**
- **Average Response Time**

On the **Statistics** panel you will also see a graphical view of the system response time.



## Managing Service Groups

A **Service Group** is a convenient way of deploying multiple services in one service archive file. There is a logical relationship between the services at runtime. The only difference in the `services.xml` for a service group and a single service is its root element. For a service group, the root element is `<serviceGroup>`, and we have multiple `<service>` elements inside the `<serviceGroup>` element.

For example,

```

<serviceGroup>
 <service name=Test1>
 ...
 <service>
 <service name=Test2>
 ...
 </service>
 </serviceGroup>

```

The following sections describe how to manage service groups:

- [Accessing Service Groups](#)
- [Managing Parameters of the Service Group](#)
- [Managing Module Engagements](#)
- [Creating the Archive File](#)

### Accessing Service Groups

Follow the instructions below to manage the parameters of a service group.

1. Click the **Main** tab on Management Console and then go to **Manage -> Service Bus** and click **List**. The **Deployed Services** screen appears.

**Deployed Services**

4 active services. 4 deployed service group(s).

Services						
<input type="checkbox"/>	echo	axis2	Unsecured	WSDL1.1	WSDL2.0	Try this service
<input type="checkbox"/>	Version	axis2	Unsecured	WSDL1.1	WSDL2.0	Try this service
	wso2carbon-sts	sts	Unsecured	WSDL1.1	WSDL2.0	
	XKMS	axis2	Unsecured	WSDL1.1	WSDL2.0	

- Click the **deployed service group(s)** link to access the deployed service groups.

**Deployed Services**

4 active services. **4 deployed service group(s)**.

- In the **Deployed Service Groups** screen that appears, click the required service group to see the relevant **Service Group Dashboard** screen.

Home > Manage > Services > List > Service Group Dashboard

**Service Group Dashboard (echo)**

Service Group Details			
Service Group Name	echo		
<b>Actions</b>			
Parameters			
Modules			
Create Service Archive			
MTOM	False		
<b>Services</b>			
echo	WSDL1.1	WSDL2.0	Try this service

### Managing Parameters of the Service Group

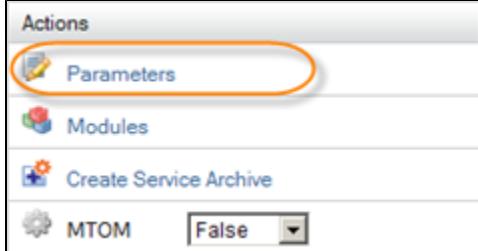
Parameters can be defined inside `services.xml` as an immediate child element of the `service` element. These parameters can be accessed using the message context at runtime or via `AxisService` or `AxisOperation`. A parameter has two attributes:

- name** - The mandatory attribute which specifies the name of a parameter.
- locked** - The optional attribute. The idea of a "locked" attribute is to express whether we allow the parameter value to be overridden by a child node in the hierarchy.

For example, if a parameter was added in the `axis2.xml` file with the `locked` attribute set to True, then if a service tries to add another parameter with the same name, it will give an exception. You can add easily add service group parameters via the Management Console.

## To add a new service group parameter

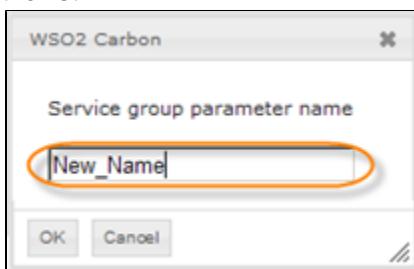
- On the **Actions** panel in the **Service Group Dashboard** screen, click **Parameters**.



- Click **Add New** to create a new service parameter.



- On the window that appears, enter the service group parameter name and click OK to add the parameter to the list.

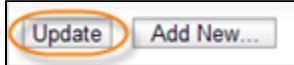


- Enter the value of the parameter.

Service Group Parameters (Service Group: echo)	
Name	Value
Name	<parameter name="Name" locked="false">test_1</parameter>
New_Name	<parameter name="New_Name" locked="false">Enter value here</parameter>

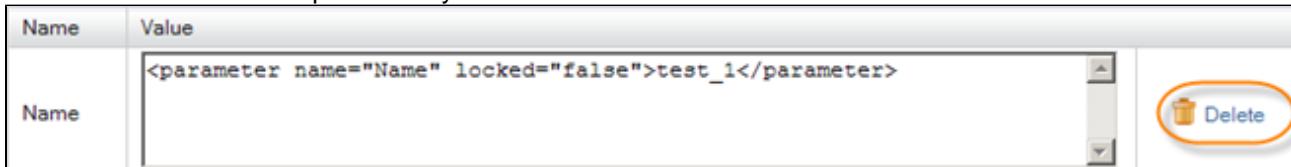
At the bottom of the table are 'Update' and 'Add New...' buttons.

- Click **Update** to save the new parameter in the service group.



## To delete a service group parameter

- On the **Actions** panel in the **Service Group Dashboard** screen, click **Parameters**.
- Click **Delete** based on the parameter you want to delete.



- Confirm that you are sure you want to delete the parameter by clicking **Yes** on the window that appears.

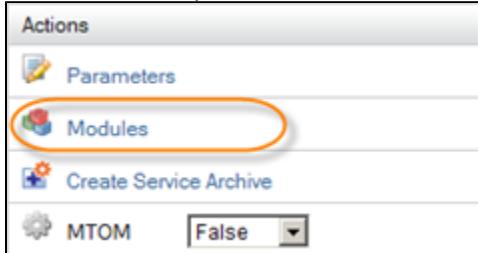


## Managing Module Engagements

There are instances when it is impossible to run the service without engaging the WS-Security module into the service. Engaging a module is just a matter of adding a module tag in the `services.xml` file. If the module is available, it can engage. Else it becomes a faulty service.

### To engage a module into your service group

1. On the **Actions** panel in the **Service Group Dashboard** screen, click **Modules**.



2. Select the required a module from the drop down list. WSO2 ESB provides the following modules that you can engage into your service group:

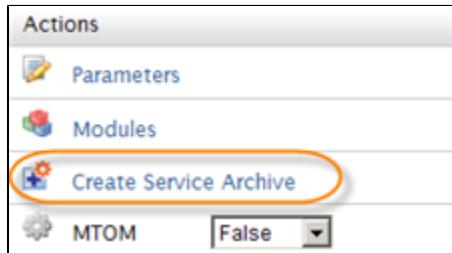
- **rampart-1.61-wso2v14** - Provides the WS-Security and WS-SecureConversation functionalities for Axis2, based on Apache WSS4J, Apache XML-Security and Apache Rahas implementations.
- **relay-4.4.1.20** - Unwraps the binary messages coming from the message relay for admin services.
- **rahas-1.61-wso2v14** - Is used to STS enable a service where it adds the RequestSecurityToken operation to a service that the module is engaged to.

3. Click **Engage**.

For more information on modules, see [Working with Modules](#).

## Creating the Archive File

- On the **Actions** panel in the **Service Group Dashboard** screen, click **Create Service Archive**. This creates the service archive file and downloads it to the Downloads directory on your computer.



## Working with Topics and Events

**Events** are notification messages published to a specific **topic**. Web services can subscribe to a topic in order to receive events that are published to it, allowing web services to publish and subscribe to each others' notifications.

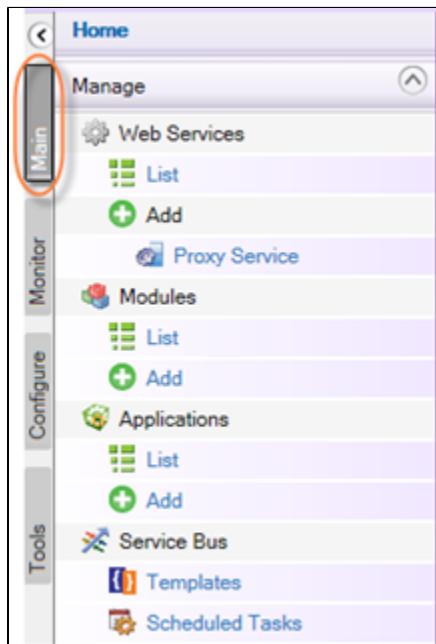
The Topic Browser in the Management Console allows you to create, manage, and subscribe to topics, and the [Event Mediator](#) makes it easy for your [proxy service](#) or [sequence](#) to publish a received request or response to a topic as an event. Eventing in WSO2 ESB is based on the [WS-Eventing Specification](#). The following pages describe how to work with topics in detail:

- Adding a New Topic
- Adding a New Subtopic
- Viewing Topic Details
- Subscribing to a Topic
- Deleting a Topic

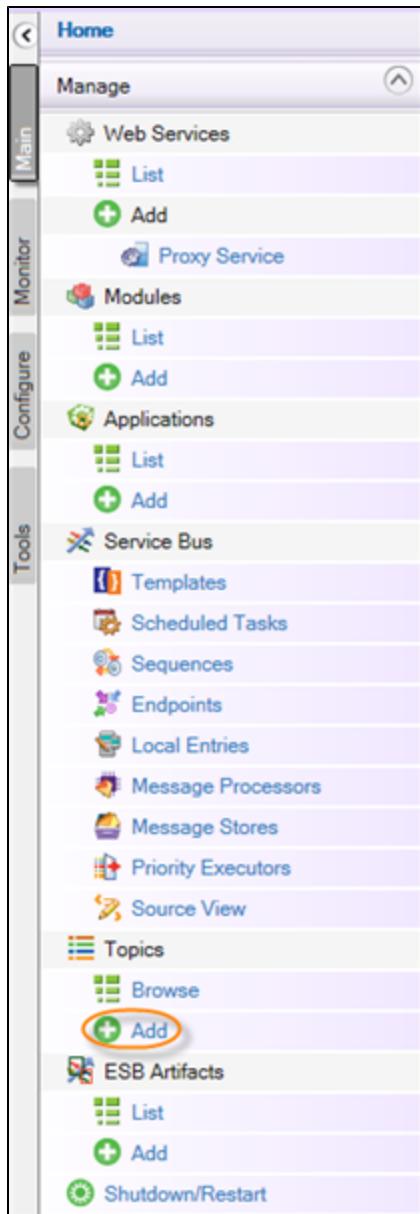
## Adding a New Topic

Follow the instructions below to add a new [topic](#) directly to the root.

1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Click the "Main" button to access the "Manage" menu.



3. Click on "Add," under "Topics", to access the "Add Topic" page.



4. On the "Add Sub Topic" page, enter a name of the new topic in the "Topic" field.

Add Topic		
Add Topic		
Topic	<input type="text"/>	
Permissions		
Role	Subscribe	Publish
everyone	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<input type="button" value="Add Topic"/>		

You can also chose whether to subscribe and publish the topic or not in the corresponding check boxes.

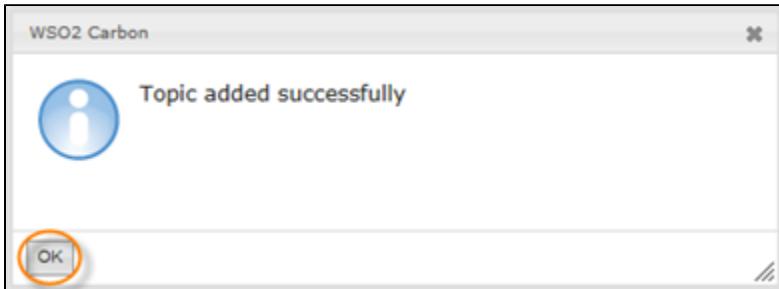
**Add Topic**

Add Topic		
Topic	<input type="text"/>	
Permissions		
Role	Subscribe	Publish
everyone	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<b>Add Topic</b>		

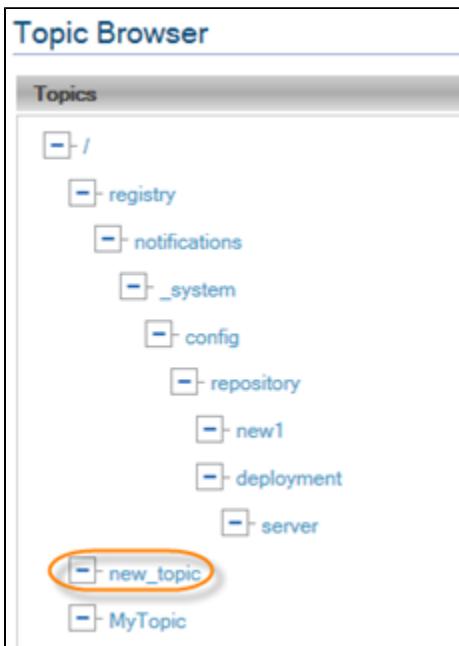
5. Click the "Add Topic" button.



6. The "WSO2 Carbon" window appears. Click "OK."



7. The new topic is shown in the topics tree.



### Note

If you want to add a subtopic under an existing topic, see [Adding a New Subtopic](#).

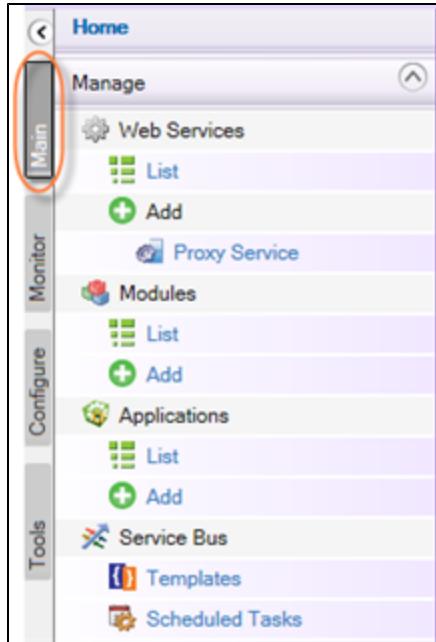
### Adding a New Subtopic

The "Add Subtopic" feature provides user the ability to add a [topic](#) under the existing topic. When adding a new

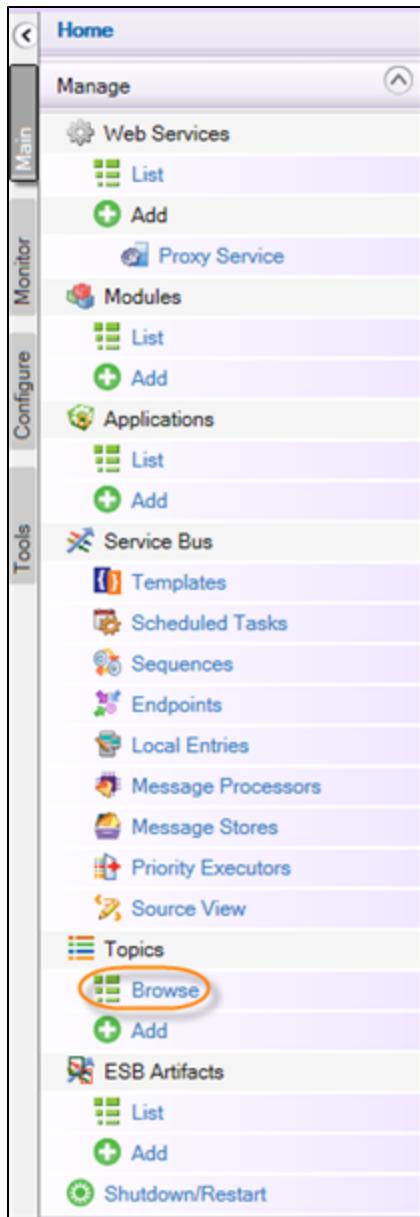
topic under an existing topic, user can provide the name of the new topic and set the permissions for publishing and subscribing for that topic.

Follow the instructions below to add a subtopic.

1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Click the "Main" button to access the "Manage" menu.



3. Click on "Browse," under "Topics", to access the "Topic Browser" page.



4. The "Topic Browser" page appears. Click on the topic you want to add a subtopic to.

5. Click on the "Add Subtopic" link.

The screenshot shows the WSO2 ESB Topic Browser. The left sidebar lists categories like registry, notifications, \_system, config, repository, deployment, and server. Under the config category, there is a subtopic named 'MyTopic'. A context menu is open over the 'config' node, with the 'Add Subtopic' option highlighted by an orange oval.

6. On the "Add Sub Topic" page, enter a name of the new subtopic into the "Topic Name" field.

The screenshot shows the 'Add Sub Topic' configuration page. It includes fields for 'Parent Topic' (registry/notifications/\_system/config), 'Subtopic Details' (Topic Name field highlighted with an orange oval), and 'Permissions' (a table where 'everyone' has checked boxes for 'Subscribe' and 'Publish'). There is also an 'Add Topic' button at the bottom.

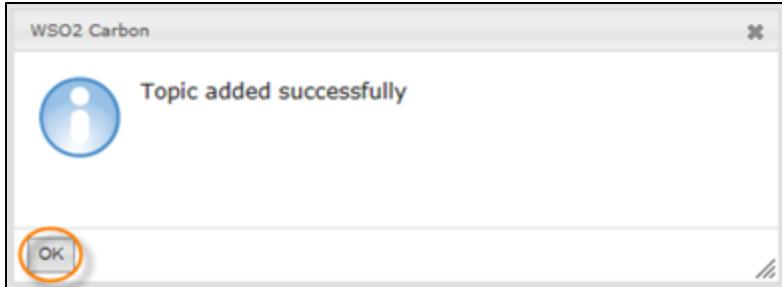
You can also choose whether to subscribe and publish the subtopic or not in the corresponding check-boxes.

The screenshot shows the 'Add Sub Topic' configuration page again. The 'Topic Name' field is empty. In the 'Permissions' section, the 'everyone' row in the table has both 'Subscribe' and 'Publish' checkboxes checked, with orange circles highlighting the checked status.

7. Click the "Add Topic" button.



8. The "WSO2 Carbon" window appears. Click "OK."



9. The new subtopic is shown in the topics tree.

The screenshot shows the 'Topic Browser' interface with a tree view of topics. The root node '/' has several children: 'registry', 'notifications', '\_system', 'config', 'repository', 'deployment', and 'server'. Under 'repository', there is a node 'new1' which is highlighted with a red oval. At the bottom of the tree view, there is a node 'MyTopic'.

### Note

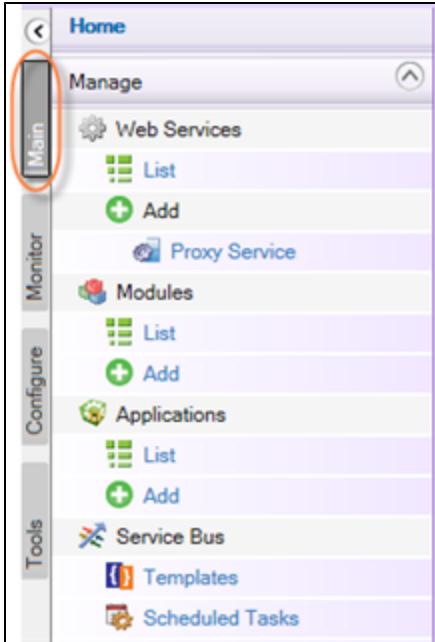
If you want to add a topic directly to the root, see [Adding a New Topic](#).

## Viewing Topic Details

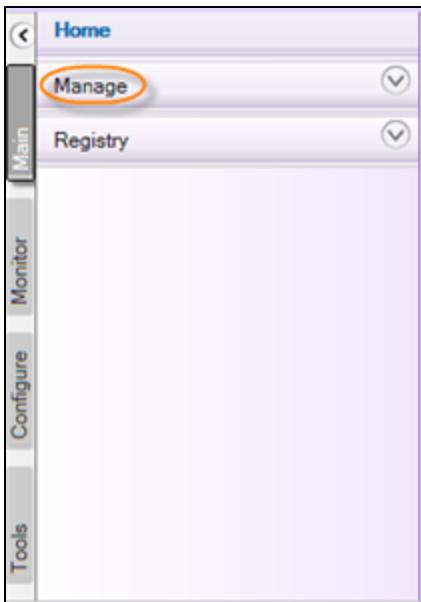
Each [topic](#) has specific details such as permissions, WS subscription details, and JMS subscription details.

Follow the instructions below to view the details of a topic.

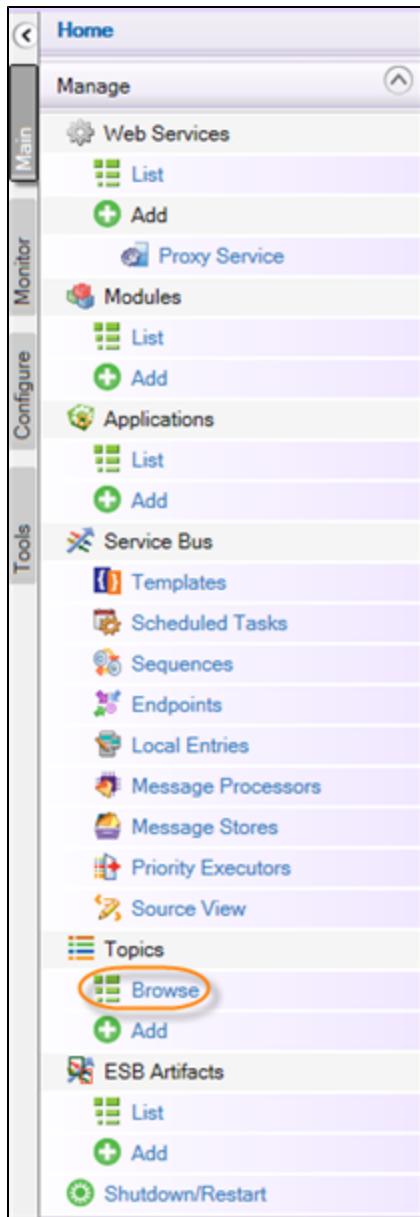
1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Click the "Main" button to access the "Manage" menu.



3. Click on the "Manage" button to access the "Manage" drop-down menu.



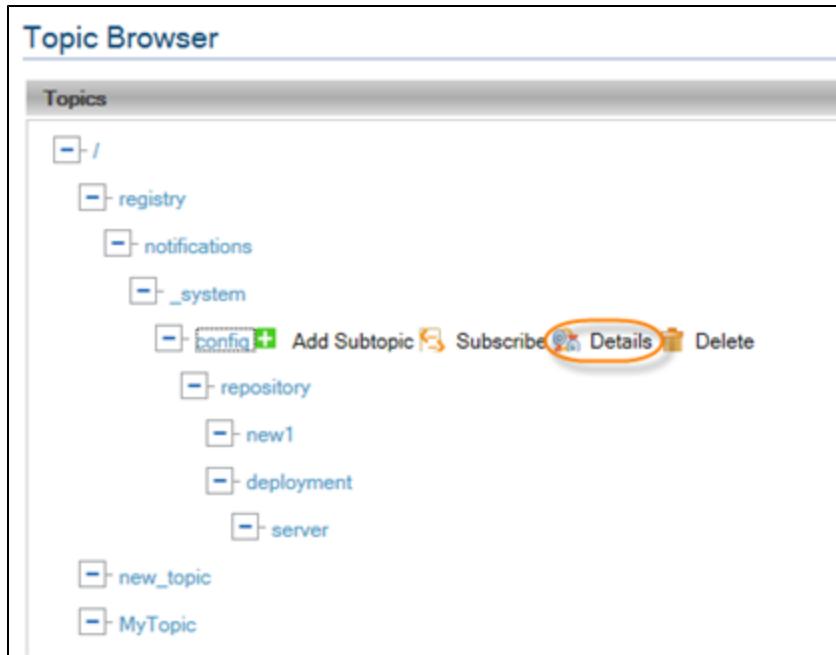
4. Click on "Browse," under "Topics," to access the "Topic Browser" page.



5. The "Topic Browser" page appears. Click on the topic you want to see the details of.

The screenshot shows the Topic Browser interface. At the top, it says 'Topic Browser'. Below that is a 'Topics' section with a tree view. The tree structure includes: /, registry, notifications, \_system, config (which is highlighted with a red oval), repository, deployment, and server. To the right of the tree, there is a context menu with options: Add Subtopic, Subscribe, Details, and Delete.

6. Click on the "Details" link.



7. The "Topic Details" page appears.

**Topic Details**

Topic Name	new_topic
------------	-----------

**Permission Details**

Role	Subscribe	Publish
wso2.anonymous.role	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
everyone	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

**WS Subscription Details**

Topic	Event Sink URL	Mode	Created Time	Expiration Time	Action
http://wso2.org	Topic Only	Fri Jul 29 09:33:30 EEST 2011	Sat Jul 30 00:00:00 EEST 2011		Unsubscribe  Renew

**JMS Subscription Details**

Name	Created Time	Owner
b6c1f5a8bceaa1238e093a7a48e30e1f5bd67be8af2b78b2	2011/07/29 09:31:09	admin
e5c1f5a8bceaa123880a3a7a48e30e1f3ad67be8af2b78b2	2011/07/29 09:33:30	admin

**Publish**

Topic	new_topic
XML Message	<div style="border: 1px solid #ccc; height: 100px; width: 100%;"></div>
<b>Publish</b>	

All the details related with a topic can be viewed on this page. The following details are available:

- **Permission Details** - Allows to view the permissions related with the topic.
- **WS Subscription Details** - Shows all the WS subscriptions for the topic and all its children.
- **JMS Subscription Details** - All the durable and non durable JMS subscriptions are listed here.
- **Publish** - Allows the user to publish a sample XML message to a topic.

#### Permission Details

If you want to change the permission details, proceed to the following steps:

1. Chose the role of the topic and change the ticks on the provided check boxes.

**Permission Details**

Role	Subscribe	Publish
wso2.anonymous.role	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
everyone	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

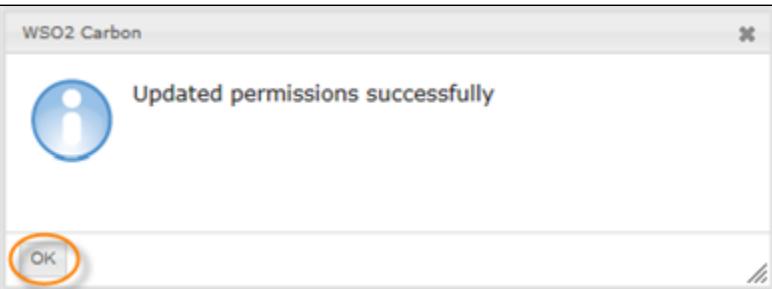
**Update Permissions**

2. Click on the "Update Permissions" button.

### Permission Details

Role	Subscribe	Publish
wso2.anonymous.role	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
everyone	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<a href="#">Update Permissions</a>		

3. The "WSO2 Carbon" window appears. Click "OK."



### WS Subscription Details

Using "Ws Subscription Details" the user can unsubscribe or renew the subscription.

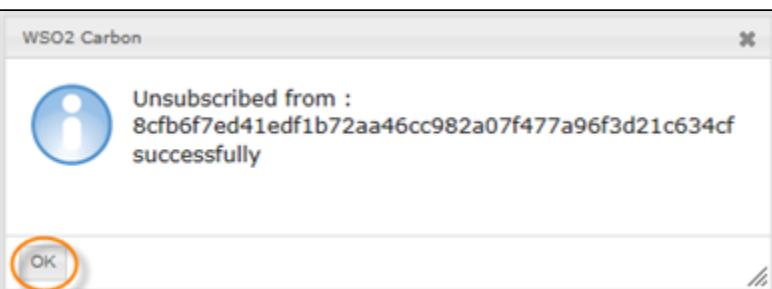
#### Unsubscribing

Use the following instructions to unsubscribe.

1. Choose the necessary subscription and click on the "Unsubscribe" link.

WS Subscription Details					
Topic	Event Sink URL	Mode	Created Time	Expiration Time	Action
http://wso2.org	Topic Only		Fri Jul 29 09:33:30 EEST 2011	Sat Jul 30 00:00:00 EEST 2011	<a href="#">Unsubscribe</a> <a href="#">Renew</a>

2. In the "WSO2 Carbon" window, click "OK" to confirm your request.



#### Renewing the Subscription

Follow the instructions below to unsubscribe.

1. Choose the necessary subscription and click on the "Renew" link.

WS Subscription Details					
Topic	Event Sink URL	Mode	Created Time	Expiration Time	Action
http://wso2.org	Topic Only		Fri Jul 29 09:33:30 EEST 2011	Sat Jul 30 00:00:00 EEST 2011	<a href="#">Unsubscribe</a> <a href="#">Renew</a>

2. The "Renew Subscription page appears," fill in the required fields.

## Renew Subscription

Enter Subscription Details

Topic*	<input type="text" value="/new_topic"/>
Event Sink URL*	<input type="text" value="http://docs.wso2.org"/>
Expiration Time	Date: <input type="text" value="2011/07/30"/> Time: <input type="text" value="HH"/> <input type="text" value="mm"/> <input type="text" value="ss"/>
<input type="button" value="Renew"/>	

### Tip

See the detailed description of the fields in [Subscribing to a Topic](#).

3. Click the "Renew" button.



4. In the "WSO2 Carbon" window, click "OK" to confirm your request.



Publish

Follow the instruction below to publish XML message.

1. Place a XML message in the provided space.

Publish

Topic	<input type="text" value="new_topic"/>
XML Message	<pre>&lt;payment type="check" /&gt;</pre>
<input type="button" value="Publish"/>	

2. Click "Publish."



3. In the "WSO2 Carbon" window, click "OK" to confirm your request.

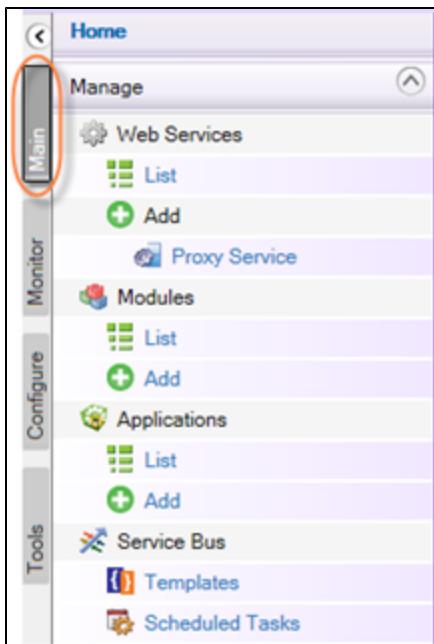


## Subscribing to a Topic

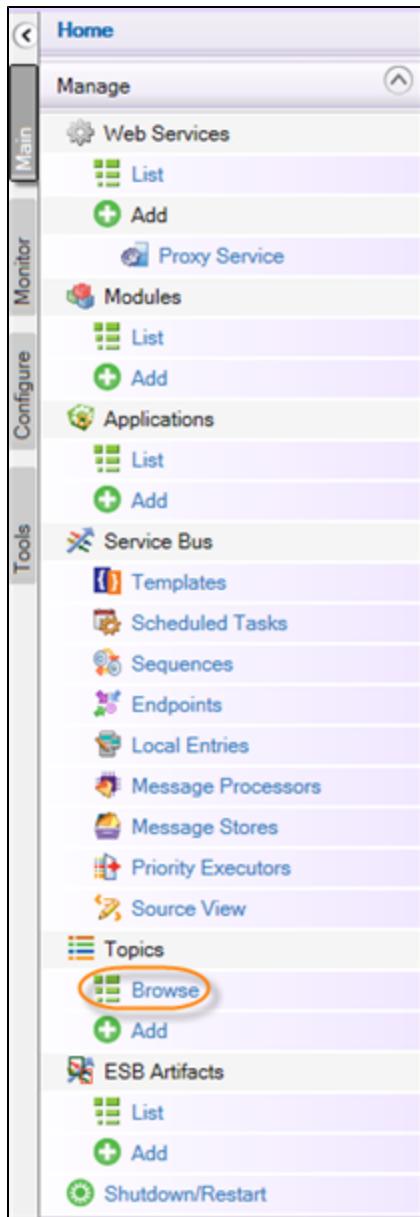
The subscription allows to receive events published to the specified [topic](#) and all its children.

Follow the instructions below to add subscriptions for a particular topic.

1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Click the "Main" button to access the "Manage" menu.



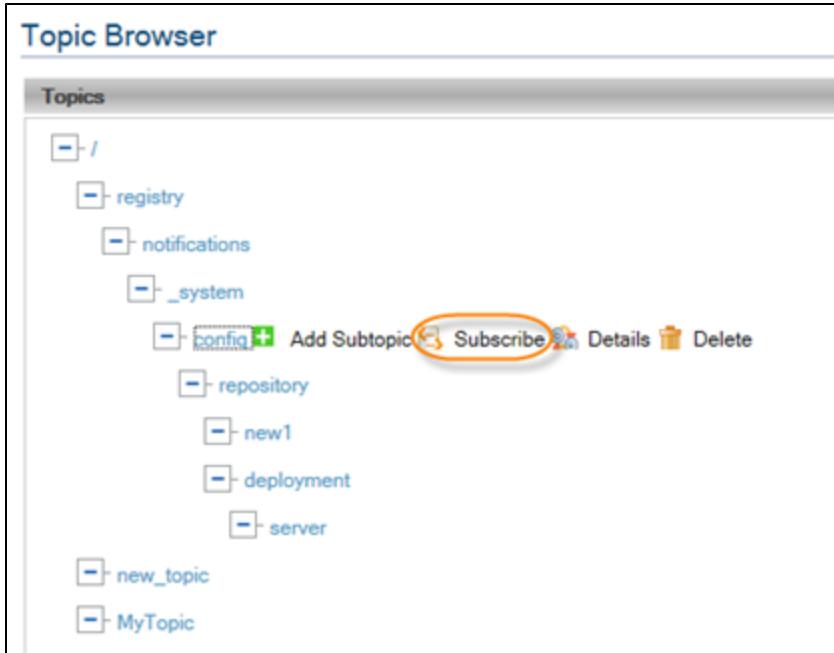
3. Click on "Browse," under "Topics", to access the "Topic Browser" page.



4. The "Topic Browser" page appears. Click on the topic you want to add a subscription to.

The screenshot shows the Topic Browser interface. At the top, it says 'Topic Browser'. Below that is a 'Topics' section with a tree view. The tree starts with a root node, then branches into registry, notifications, \_system, repository, deployment, and server. Under the \_system node, there is a 'config' node. A red circle highlights the 'config' node. To the right of the tree, there are several actions: Add Subtopic, Subscribe, Details, and Delete.

5. Click on the "Subscribe" link.



6. The "Subscribe" page appears. Specify the required details.

The screenshot shows the 'Subscribe' configuration page. The title is 'Subscribe'. The form is titled 'Enter Subscription Details'. It contains the following fields:

- Topic\***: registry/notifications/\_sys
- Subscription Mode :\***: Topic Only (selected from a dropdown menu)
- Event Sink URL\***: (empty input field)
- Date:** (empty input field)
- Time:** (empty input field)
- Expiration Time**: (empty input field) with a calendar icon and time components (hh, mm, ss) set to eg:(15/30/00)

At the bottom are two buttons: 'Subscribe' and 'Cancel'.

The following options are available:

- **Topic** - User does not need to specify the topic here, since its automatically sets up.
- **Subscription Mode**- This is the mode of the subscription and there are tree modes:
  - "**Topic Only**" - The default mode for the subscription. With this mode, user creates the subscription only to the topic. In that mode subscribers only receive events, which are published only to the that topic.
  - "**Topic and Immediate child**" - Allows to receive events published not only the specified topic, but also to the immediate child of that topic.
  - "**Topic and Children**" - Allows to receive events published to the specified topic and all its children
- **Event Sink URL** - This is the URL which the subscriber should provide to receive events published. When events are published to the topic, they are sent to the specified URL here.
- **Expiration Time** - Here user can specify the expiration time of the subscription. This is not a required parameter and if user leave it alone, subscription will never be expired.

7. The "WSO2 Carbon" window appears. Click "OK."



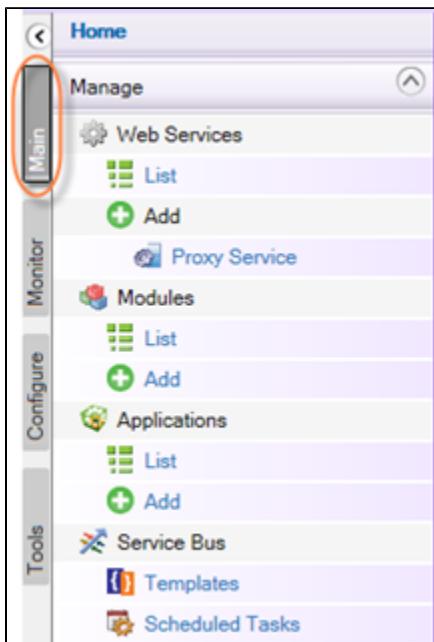
## Deleting a Topic

If you want to delete a [topic](#) from the server, follow the instructions below.

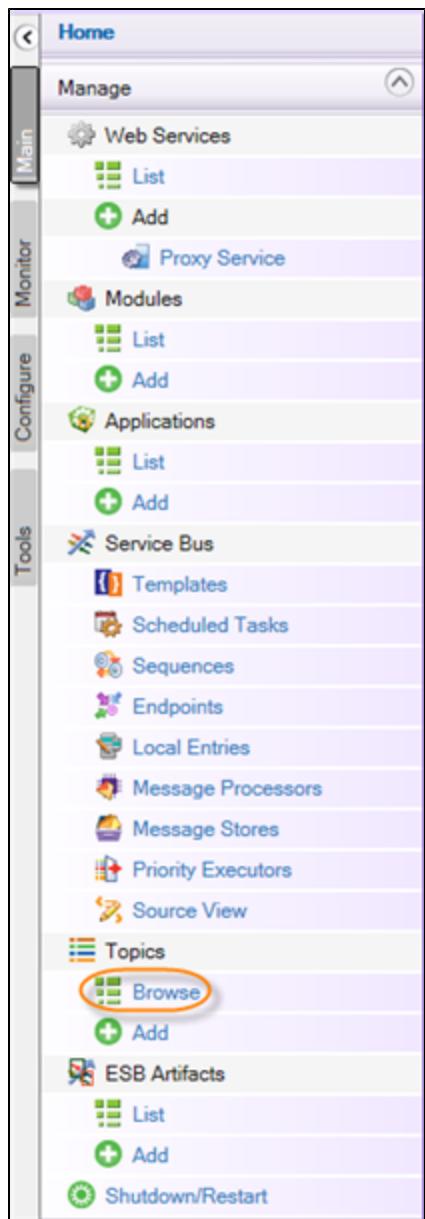
### Note

To delete a topic, subscription count for that topic and its children should be zero. Otherwise it will prompt an error message specifying that there are subscriptions for the topic or its children.

1. Sign in. Enter your user name and password to log on to the ESB Management Console.
2. Click the "Main" button to access the "Manage" menu.



3. Click on "Browse," under "Topics", to access the "Topic Browser" page.



4. The "Topic Browser" page appears. Click on the topic you want to delete.

**Topic Browser**

**Topics**

- /
- registry
- notifications
- \_system
  - config
    - repository
      - new1
    - deployment
    - server
- new\_topic Add Subtopic Subscribe Details Delete
- MyTopic

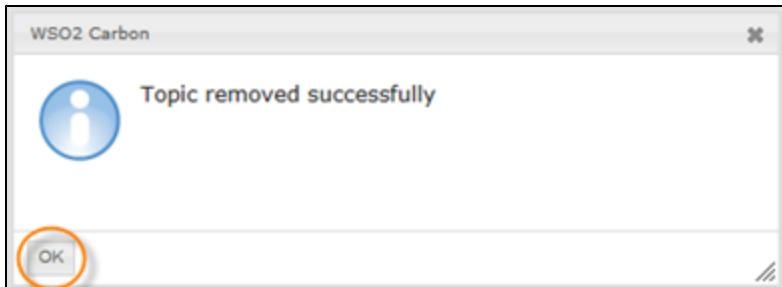
5. Click on the "Delete" link.

**Topic Browser**

**Topics**

- /
- registry
- notifications
- \_system
  - config
    - repository
      - new1
    - deployment
    - server
- new\_topic Add Subtopic Subscribe Details Delete
- MyTopic

6. In the "WSO2 Carbon" window, click "OK."



### Per-Service Logs in WSO2 ESB

The advantage of having per-service log files is that it is very easy to analyze/monitor what went wrong in this

particular Proxy Service by looking at the service log. Enabling this feature will not terminate the `wso2-esb.log` file being keeping the logs about this service, the complete log will contain every log statement including the service logs that you have configured to be logged into a different log file. In other words, the service log is an additional log file, which will contain a copy of the logs to that particular service.

Follow the instructions below to configure the logs of a particular service (to be more specific a Proxy Service) to be logged into a given log file.

1. See [Sample 150 in Proxy Service Samples](#).

It has a Proxy Service named `StockQuoteProxy`.

2. Configure `log4j` to log the service specific logs to a file called `stock-quote-proxy-service.log` in the `logs` directory of the ESB installation directory.

2.1. Open up the `log4j.properties` file found in the `/repository/conf` directory of the WSO2 ESB installation directory using your favorite text editor and add the following section to the end of the file starting in a new line.

```
log4j.category.SERVICE_LOGGER.StockQuoteProxy=DEBUG, SQ_PROXY_APPENDER
log4j.additivity.SERVICE_LOGGER.StockQuoteProxy=false
log4j.appenders.SQ_PROXY_APPENDER=org.apache.log4j.DailyRollingFileAppender
log4j.appenders.SQ_PROXY_APPENDER.File=logs/stock-quote-proxy-service.log
log4j.appenders.SQ_PROXY_APPENDER.datePattern='yyyy-MM-dd-HH-mm
log4j.appenders.SQ_PROXY_APPENDER.layout=org.apache.log4j.PatternLayout
log4j.appenders.SQ_PROXY_APPENDER.layout.ConversionPattern=%d{ISO8601}
\[%X{ip}-%X{host}\] \[%t\] %5p %c{1} %m%n
```

- 2.2. Save the file.

3. Try it out. By default, the configuration does not do any logging at run time, so configure the Proxy Service in-sequence to contain a log mediator to log the message at "Full" log level.

Execute the sample client after starting the ESB with sample 150:

```
$ESB_HOME/bin/wso2esb-samples.sh \-sn 150
```

and the sample axis2 server with the `SimpleStockQuote` service deployed on it as per stated in the sample documentation.

4. Inspect the `logs` directory of the ESB installation directory to see the `stock-quote-proxy-service.log` file.

Further to demonstrate the log file rotation this particular logger was configured to rotate the file in each minute when ever there is a log going into the service log, so if you execute the sample client once again after 1 minute you will be able to see the service log file rotation as well.

You can see the source [here](#).

## Working with Templates

The ESB configuration language is a very powerful and robust way of driving enterprise data/messages through the ESB mediation engine. However, a large number of configuration files in the form of `sequences`, `endpoints`, `proxy services`, and transformations can be required to satisfy all the mediation requirements of your system. To keep your configurations manageable, it's important to avoid scattering configuration files across different locations and to avoid duplicating redundant configurations.

**ESB templates** help minimize this redundancy by creating prototypes that users can use and reuse when needed. This is very much analogous to classes and instances of classes: a template is a class that can be used to wield

instance objects such as templates and endpoints. Thus, ESB templates are an ideal way to improve reusability and readability of ESB configurations/XMLs. Additionally, users can use predefined templates that reflect common enterprise integration patterns for rapid development of ESB message/mediation flows.

ESB templates comes in two different forms.

- **Endpoint Template** - Defines a templated form of an endpoint. An endpoint template can parameterize an endpoint defined within it. You invoke an endpoint template as follows:

```
<endpoint template="name" ...>
<parameter name="name" value="value" />
.....
</endpoint>
```

- **Sequence Template** - Defines a templated form of an ESB sequence. A sequence template can parameterize XPath expressions used within a sequence that is defined inside a template. You invoke a sequence template with the **Call Template mediator** by passing in the parameter values as follows:

```
<call-template target="template" >
<parameter name="name" value="value" />
.....
</call-template>
```

For more information on creating and using these templates, see [Endpoint Template](#) and [Sequence Template](#).

## Endpoint Template

**Endpoint template** is a generalized form of endpoint configuration used in ESB. Unlike [sequence templates](#), endpoint templates are always parametrized using \$ prefixed values (not XPath expressions).

Template endpoint is the artifact that makes a template of endpoint type into a concrete endpoint. In other words, an endpoint template would be useless without a template endpoint referring to it. This is semantically similar to the relationship between a sequence template and the Call Template Mediator.

---

[Configuration](#) | [Syntax](#) | [Template Endpoint Sample](#)

### Configuration

For example, let's say we have two default endpoints with following hypothetical configurations:

```

<endpoint name="ep1">
 <default>
 <suspendOnFailure>
 <errorCodes>10001,10002</errorCodes>
 <progressionFactor>1.0</progressionFactor>
 </suspendOnFailure>
 <markForSuspension>
 <retriesBeforeSuspension>5</retriesBeforeSuspension>
 <retryDelay>0</retryDelay>
 </markForSuspension>
 </default>
</endpoint>

```

```

<endpoint name="ep2">
 <default>
 <suspendOnFailure>
 <errorCodes>10001,10003</errorCodes>
 <progressionFactor>2.0</progressionFactor>
 </suspendOnFailure>
 <markForSuspension>
 <retriesBeforeSuspension>3</retriesBeforeSuspension>
 <retryDelay>0</retryDelay>
 </markForSuspension>
 </default>
</endpoint>

```

We can see that these two endpoints have different set of error codes and different progression factors for suspension. Furthermore, the number of retries is different between them. By defining a endpoint template, these two endpoints can be converged to a generalized form. This is illustrated in the following:

```

<template name="ep_template">
 <parameter name="codes"/>
 <parameter name="factor"/>
 <parameter name="retries"/>
 <endpoint name="$name">
 <default>
 <timeout>
 <duration>$timeout</duration>
 <responseAction>$action</responseAction>
 </timeout>
 <suspendOnFailure>
 <errorCodes>$codes</errorCodes>
 <progressionFactor>$factor</progressionFactor>
 </suspendOnFailure>
 <markForSuspension>
 <retriesBeforeSuspension>$retries</retriesBeforeSuspension>
 <retryDelay>0</retryDelay>
 </markForSuspension>
 </default>
 </endpoint>
</template>

```

**Note**

`$` is used to parametrize configuration and `$name` is an implicit/default parameter.

Since we have a template defined, we can use template endpoints to create two concrete endpoint instances with different parameter values for this scenario. This is shown below.

```
<endpoint name="ep1" template="ep_template">
 <parameter name="timeout" value="{get-property('timeout')}" /> <!-- timeout is
defined using a Property mediator outside the endpoint configuration -->
 <parameter name="codes" value="10001,10002" />
 <parameter name="factor" value="1.0" />
 <parameter name="retries" value="5" />
</endpoint>
```

```
<endpoint name="ep2" template="ep_template">
 <parameter name="codes" value="10001,10003" />
 <parameter name="factor" value="2.0" />
 <parameter name="retries" value="3" />
</endpoint>
```

As with the Call Template Mediator, the above template endpoint will stereotype endpoints with customized configuration parameters. This makes it very easy to understand and maintain certain ESB configurations and improves re-usability in a certain way.

**Syntax**

```
<template name="string">
 <!-- parameters this endpoint template will be supporting -->
 (
 <parameter name="string" />
) *
 <!--this is the in-line endpoint of the template -->
 <endpoint [name="string"] >
 address-endpoint | default-endpoint | wsdl-endpoint | load-balanced-
 endpoint | fail-over-endpoint
 </endpoint>
</template>
```

Similar to sequence templates, endpoint templates are defined as top level element (with the name specified by the name attribute) of ESB configuration. Parameters (for example, `<parameter>`) are the inputs supported by this endpoint template. These endpoint template parameters can be referred by `$` prefixed parameter name. For example, parameter named `foo` can be referred by `$foo`. Most of the parameters of the endpoint definition can be parametrized with `$` prefixed values. This is shown in the following extract:

```

<template name="sample_ep_template">
 <parameter name="foo"/>
 <parameter name="bar"/>
 <default>
 <suspendOnFailure>
 <errorCodes>$foo</errorCodes>
 <progressionFactor>$bar</progressionFactor>
 </suspendOnFailure>
 <markForSuspension>
 <retriesBeforeSuspension>0</retriesBeforeSuspension>
 <retryDelay>0</retryDelay>
 </markForSuspension>
 </default>
</endpoint>
</template>

```

## Note

\$name and \$uri are default parameters that a template can use anywhere within the endpoint template (usually used as parameters for endpoint name and address attributes).

### **Template Endpoint Sample**

```

<endpoint [name="string"] [key="string"] template="string">
 <!-- parameter values will be passed on to a endpoint template -->
 (
 <parameter name="string" value="string" />
) *
</endpoint>

```

Template endpoint defines parameter values that can parametrize endpoint. The template attribute points to a target endpoint template.

## Note

Parameter names has to be exact match to the names specified in target endpoint template.

WSO2 ESB allows [add](#), [delete](#), and [edit](#) endpoint templates.

For more information about templates, see [Working with Templates](#).

### **Sequence Template**

A **Sequence Template** is a parametrized [sequence](#) , providing an abstract or generic form of a sequence defined in the ESB. Parameters of a template are defined in the form of XPath statement/s. Callers can invoke the template by populating the parameters with static values/XPath expressions using the Call Template Mediator, which makes a sequence template into a concrete sequence.

---

[Configuration](#) | [Syntax](#) | [Call Template Mediator](#) | [Syntax](#) | [UI Configuration](#) | [Example](#)

---

### **Configuration**

Let's illustrate the sequence template with a simple example. Suppose we have a sequence that logs the text "hello world" in four different languages.

```

<sequence>
 <switch source="/m0:greeting/m0:lang" xmlns:m0="http://services.samples">
 <case regex="EN">
 <log level="custom">

 <property name="GREETING_MESSAGE" value="HELLO WORLD!!!!!" />

 </log>
 </case>
 <case regex="FR">
 <log level="custom">

 <property name="GREETING_MESSAGE" value="Bonjour tout le monde!!!!!" />

 </log>
 </case>
 <case regex="IT">
 <log level="custom">

 <property name="GREETING_MESSAGE" value="Ciao a tutti!!!!!!" />

 </log>
 </case>
 <case regex="JPN">
 <log level="custom">

 <property name="GREETING_MESSAGE" value="??????!!!!!!" />

 </log>
 </case>
 </switch>
</sequence>

```

Instead of printing our "hello world" message for each and every language inside the sequence, we can create a generalized template of these actions, which will accept any greeting message (from a particular language) and log it on screen. For example, let's create the following template named "HelloWorld\_Logger".

```

<template name="HelloWorld_Logger">
 <parameter name="message" />
 <sequence>
 <log level="custom">
 <property name="GREETING_MESSAGE" expression="$func:message" />
 </log>
 </sequence>
</template>

```

With our "HelloWorld\_Logger" in place, the Call Template mediator can populate this template with actual hello world messages and execute the sequence of actions defined within the template like with any other sequence. This is illustrated below.

```

<sequence>
 <switch source="/m0:greeting/m0:lang" xmlns:m0="http://services.samples">
 <case regex="EN">
 <call-template target="HelloWorld_Logger">
 <with-param name="message" value="HELLO WORLD!!!!!!" />
 </call-template>
 </case>
 <case regex="FR">
 <call-template target="HelloWorld_Logger">
 <with-param name="message" value="Bonjour tout le monde!!!!!!" />
 </call-template>
 </case>
 <case regex="IT">
 <call-template target="HelloWorld_Logger">
 <with-param name="message" value="Ciao a tutti!!!!!!" />
 </call-template>
 </case>
 <case regex="JPN">
 <call-template target="HelloWorld_Logger">
 <with-param name="message" value="??????!!!!!!" />
 </call-template>
 </case>
 </switch>

</sequence>

```

The Call Template mediator points to the same template "HelloWorld\_Logger" and passes different arguments to it. In this way, sequence templates make it easy to stereotype different workflows inside ESB.

## Syntax

```

<template name="string">
 <!-- parameters this sequence template will be supporting -->
 (
 <parameter name="string" />
) *
 <!--this is the in-line sequence of the template -->
 <sequence>
 mediator+
 </sequence>
</template>

```

The sequence template is a top-level element defined by the name attribute in the ESB configuration. Both endpoint and sequence template starts with a template element.

The parameters available to configure the Sequence Template are as follows.

Parameter Name	Description
Name	The name of the Sequence Template
onError	Select the error sequence that needs to be invoked.

Trace Enabled	Whether or not trace is to be enabled for the sequence.
Statistics Enabled	Whether or not statistics is to be enabled for the sequence.
Template Parameters	The input parameter that are supported by this Sequence Template.

Sequence template parameters can be referenced using an XPath expression defined inside the in-line sequence. For example, the parameter named "foo" can be referenced by the Property mediator (defined inside the in-line sequence of the template) in the following ways:

```
<property name="fooValue" expression="$func:foo" />
```

or

```
<property name="fooValue" expression="get-property('foo','func')" />
```

Using function scope or "?func?" in the XPath expression allows us to refer to a particular parameter value passed externally by an invoker such as the Call Template mediator.

### **Call Template Mediator**

The Call Template mediator allows you to construct a sequence by passing values into a sequence template.

This is currently only supported for special types of mediators such as the [Iterator](#) and [Aggregate Mediators](#), where actual XPath operations are performed on a different SOAP message, and not on the message coming into the mediator.

### **Syntax**

```
<call-template target="string">
 <!-- parameter values will be passed on to a sequence template -->
 (
 <!--passing plain static values -->
 <with-param name="string" value="string" /> |
 <!--passing xpath expressions -->
 <with-param name="string" value="{string}" /> |
 <!--passing dynamic xpath expressions where values will be compiled
dynamically-->
 <with-param name="string" value="{{string}}" /> |
) *
 <!--this is the in-line sequence of the template -->
</call-template>
```

You use the `target` attribute to specify the sequence template you want to use. The `<with-param>` element is used to parse parameter values to the target sequence template. The parameter names should be the same as the names specified in target template. The parameter value can contain a string, an XPath expression (passed in with curly braces { }), or a dynamic XPath expression (passed in with double curly braces) of which the values are compiled dynamically.

### **UI Configuration**

The screenshot shows the 'Call-Template Mediator' configuration page. At the top left is the 'Mediator' tab and a 'switch to source view' link. On the right is a 'Help' button. The main area has a title 'Call-Template Mediator'. Below it is a 'Target Template\*' input field with a dropdown arrow. To its right is a 'Available Templates' section with a 'Select From Templates' dropdown. A 'Parameters of the Template mediator' section follows, containing an 'Update' button. At the bottom are 'Save & Close', 'Save', and 'Cancel' buttons.

The parameters available to configure the Call-Template mediator are as follows.

Parameter Name	Description
Target Template	The sequence template to which values should be passed. You can select a template from the Available Templates list

When a target template is selected, the parameter section will be displayed as shown below if the sequence template selected has any parameters. This enables parameter values to be parsed into the sequence template selected.

This screenshot shows the 'Parameters of the Template mediator' configuration screen. It includes a table with columns for Parameter Name, Parameter Type, Value / Expression, and Action (with a delete icon). Below the table is an 'Update' button. At the bottom are 'Save & Close', 'Save', and 'Cancel' buttons.

Parameter Name	Description
Parameter Name	The name of the parameter.
Parameter Type	The type of the parameter. Possible values are as follows. <ul style="list-style-type: none"> <li><b>Value:</b> Select this to define the parameter value as a static value. This value should be entered in the <b>Value / Expression</b> parameter.</li> <li><b>Expression:</b> Select this to define the parameter value as a dynamic value. The XPath expression to calculate the parameter value should be entered in the <b>Value / Expression</b> parameter.</li> </ul>
Value / Expression	The parameter value. This can be a static value, or an XPath expression to calculate a dynamic value depending on the value you selected for the <b>Parameter Type</b> parameter.
Action	Click <b>Delete</b> to delete a parameter.

#### Example

The following four Call Template mediator configurations populate a sequence template named HelloWorld\_Logger with the "hello world" text in four different languages.

```
<call-template target="HelloWorld_Logger">
<with-param name="message" value="HELLO WORLD!!!!!!" />
</call-template>
```

```
<call-template target="HelloWorld_Logger">
<with-param name="message" value="Bonjour tout le monde!!!!!!" />
</call-template>
```

```
<call-template target="HelloWorld_Logger">
<with-param name="message" value="Ciao a tutti!!!!!!" />
</call-template>
```

```
<call-template target="HelloWorld_Logger">
<with-param name="message" value="????????!!!!!!" />
</call-template>
```

The sequence template can be configured as follows to log any greetings message passed to it by the Call Template mediator. Thus, due to the availability of the Call Template mediator, you are not required to have the message entered in all four languages included in the sequence template configuration itself.

```
<template name="HelloWorld_Logger">
<parameter name="message" />
<sequence>
 <log level="custom">
 <property name="GREETING_MESSAGE" expression="$func:message" />
 </log>
</sequence>
</template>
```

See [Sequence Template](#) for a more information about this scenario.

For more information, see the following topics:

- [Adding a New Sequence Template](#)
- [Editing a Sequence Template](#)
- [Deleting a Sequence Template](#)
- [Working with Templates](#)
- [Sample 751: Message Split Aggregate Using Templates](#)

### Working with Templates via WSO2 ESB Tooling

You can create a new template or import an existing template from the file system using WSO2 ESB tooling.

You need to have WSO2 ESB tooling installed to create a new template or to import an existing template via ESB tooling. For instructions on installing WSO2 ESB tooling, see [Installing WSO2 ESB Tooling](#).

## Creating a template

Follow these steps to create a new scheduled task. Alternatively, you can [import an existing template](#).

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Template** on the **Developer Studio Dashboard**.
3. Leave the first option selected and click **Next**.
4. Type a unique name for the template and specify the type of template you are creating.
5. Do one of the following:
  - To save the template in an existing ESB Config project in your workspace, click **Browse** and select that project.
  - To save the template in a new ESB Config project, click **Create new Project** and create the new project.
6. If you specified an address or WSDL endpoint as the template type, enter the URL for the address or the WSDL URI and connection information in the **Advanced Configuration** fields.
7. Click **Finish**. The template is created in the `src/main/synapse-config/templates` folder under the ESB Config project you specified. When prompted, you can open the file in the editor, or you can right-click the template in the project explorer and click **Open With > ESB Editor**. Click its icon in the editor to view its properties.

## Importing a template

Follow these steps to import an existing template into an ESB Config project. Alternatively, you can [create a new template](#).

1. In Eclipse, click the **Developer Studio** menu and then click **Open Dashboard**. This opens the **Developer Studio Dashboard**.
2. Click **Template** on the **Developer Studio Dashboard**.
3. Select **Import a Template** and click **Next**.
4. Specify the XML file that defines the template by typing its full path name or clicking **Browse** and navigating to the file.
5. In the **Save Template In** field, specify an existing ESB Config project in your workspace where you want to save the template, or click **Create new Project** to create a new ESB Config project and save the template configuration there.
6. If there are multiple template definitions in the file, click **Create ESB Artifacts**, and then select the templates you want to import.
7. Click **Finish**. The templates you selected are created in the subfolders of the `src/main/synapse-config/templates` folder under the ESB Config project you specified, and the first template appears in the editor.

## Working with Templates via the Management Console

You can add, edit and delete endpoint templates as well as sequence templates via the ESB Management Console. You can also configuring statistics and tracing for sequence templates via the Management Console.

See the following topics for information on how to work with templates via the Management Console:

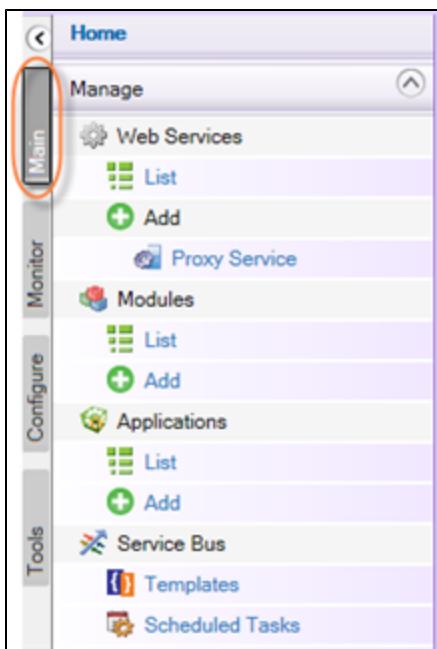
- [Adding a New Endpoint Template](#)
- [Editing an Endpoint Template](#)
- [Deleting an Endpoint Template](#)
- [Adding a New Sequence Template](#)
- [Editing a Sequence Template](#)
- [Deleting a Sequence Template](#)
- [Managing Sequence Templates](#)

### Adding a New Endpoint Template

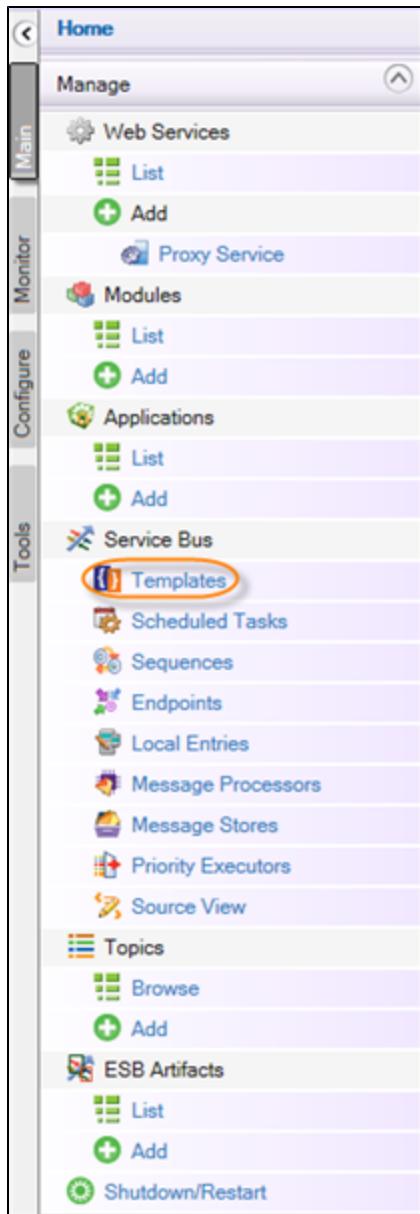
Template endpoint is the artifact that makes a template of endpoint type into a concrete endpoint. In other words, an endpoint template would be useless without a template endpoint referring to it.

Follow the instruction below to add a new [endpoint template](#).

1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Select the "Main" tab to access the "Manage" menu.



3. Click the "Templates" link to access the "Templates" page.



4. On the "Templates" page, click the "Add Endpoint Template" link.

The screenshot shows the 'Templates' page. At the top, there are two 'Add' buttons: 'Add Sequence Template' and 'Add Endpoint Template', with the latter highlighted by a red circle. Below these are four tabs: 'Sequence Templates' (selected), 'Registry Sequence Templates', 'Endpoint Templates', and 'Registry Endpoint Templates'. A section titled 'Available Templates in the Synapse Configuration' lists two entries: 'MyTemplate' and 'sequence\_template'. Each entry has a row of actions: 'Disable Statistics' (with a red circle around it), 'Disable Tracing', 'Edit', and 'Delete'.

Sequence Template Name	Actions
MyTemplate	Disable Statistics  Disable Tracing  Edit  Delete
sequence_template	Enable Statistics  Enable Tracing  Edit  Delete

5. The "Add Endpoint Templates" page appears. Select the endpoint type.

**Add Endpoint Templates**

Endpoint Template Types

Select Endpoint type to Add

 Address Endpoint Template	Defines the direct URL of the service
 Default Endpoint Template	Defines additional configuration for the default target
 WSDL Endpoint Template	Defines the WSDL, Service and Port

Currently the following three types are supported (See their description following the appropriate links):

- **Address Endpoint Template**
- **Default Endpoint Template**
- **WSDL Endpoint Template**

6. Once setting up the options, add necessary parameters and properties. See the detailed information about its fields [below](#).

7. Click the "Save" or the "Save As" button to add the template.

7. The "Save As" button allows to save the template in Governance Registry or Configuration Registry. Choose the necessary one.

Save As Synapse Registry Entry

Save in    Governance Registry     Configuration Registry

8. Click "Save."

---

[Endpoint Template Parameters](#) | [Endpoint Template Property](#) | [XML Configuration of an Endpoint Template](#)

---

## Endpoint Template Parameters

WSO2 ESB allows to add parameters to endpoint templates.

1. Click on "Add Parameter" button, a page will appear with a parameter table.

Parameter	Action
p1	Delete Parameter
p2	Delete Parameter
Add Parameter	

2. Enter the parameter name.

Parameter

The parameter name can be accessed within different attributes of the endpoint template you are currently editing.

Parameters are accessed with \$ prefix. For endpoint templates there are two inbuilt parameters name and uri which can be accessed with \$name and \$uri. The example below shows, how to access parameters uri , p1 and p2 for template attributes address, suspend error codes and suspend duration respectively.

## Endpoint Template Property

You can add property to an endpoint template.

1. Click "Add Property."



2. Fill in the necessary fields.

The following options are available:

- **Name** - The Endpoint properties name.
- **Value** - The Endpoint properties value.
- **Scope**- Can be selected as:
  - **Synapse**
  - **Transport**
  - **Axis2**
  - **axis2-client**
- **Action**
  - **Delete** - Allows to delete a property.

## XML Configuration of an Endpoint Template

1. Click "switch to source view" to view the XML of the particular endpoint template.



2. In the appeared window, you can see the XML of the endpoint template.

**Endpoint Source View**

Address Endpoint [Switch to design view](#)

```

1 <endpoint xmlns="http://ws.apache.org/ns/synapse" name="new">
2 <address uri="$uri" >
3 <suspendOnFailure>
4 <errorCodes>SPL</errorCodes>
5 <initialDuration>$p2</initialDuration>
6 <progressionFactor>1.0</progressionFactor>
7 </suspendOnFailure>
8 <markForSuspension>
9 <retriesBeforeSuspension>0</retriesBeforeSuspension>
10 <retryDelay>0</retryDelay>
11 </markForSuspension>
12 </address>
13 <property name="" value="" scope="default" />
14</endpoint>
15

```

Position: Ln 1, Ch 1 Total: Ln 15, Ch 516

Toggle editor

[Save](#) [Save As](#) [Cancel](#)

If you are familiar with the synapse configuration language, you can edit the XML directly and save the configuration using this view.

### Address Endpoint Template

The Address Endpoint Template has the following options:

- **Template Name** - The unique name for the endpoint template.
- **Name** - A name for the inline endpoint.
- **Address** - The URL of the template endpoint. This can be a parametrized value such as \$uri.

**Address Endpoint Template**

Address Endpoint [Switch to source view](#)

Template Name *	<input type="text"/>
Name *	<input type="text"/>
Address *	<input type="text"/> <a href="#">Test</a>

[Add Parameter](#)

Show Advanced Options

**Endpoint Properties**

[Add Property](#)

[Save](#) [Save As](#) [Cancel](#)

If it is a real endpoint, you can test the availability of the given URL on the fly by clicking "Test."



See the description of the Endpoint's parameters and properties in [Adding a New Endpoint Template](#).

See the Advanced options in [Adding an Endpoint](#).

See also [Address Endpoint](#).

**Default Endpoint Template**

The **Default Endpoint Template** has the following options:

- **Template Name** - The unique name for the endpoint template.
- **Name** - A name for the inline endpoint.

 A screenshot of a web-based configuration dialog titled "Default Endpoint Template". The interface includes fields for "Template Name" and "Name", both marked with red asterisks indicating they are required. There are buttons for "Add Parameter" and "Add Property". A checkbox labeled "Show Advanced Options" is checked. At the bottom, there are "Save", "Save As", and "Cancel" buttons.

See the description of the Endpoint's parameters and properties in [Adding a New Endpoint Template](#).

See the Advanced options in [Adding an Endpoint](#).

See also [Default Endpoint](#).

**HTTP Endpoint Template**

With the **HTTP Endpoint Template** you can define a URI template based REST service endpoint.

The HTTP Endpoint Template has the following options:

- **Template Name** - The unique name for the endpoint template.
- **Name** - A name for the inline endpoint.
- **URI Template** - The URI template of the endpoint. Insert `uri.var.` before each variable. Click **Test** to test the URI.

**HTTP Endpoint**  [Switch to source view](#)

Template Name *	<input type="text"/>
Name *	\$name
URI Template *	<input type="text" value="\$uri"/> <a href="#">Test</a>
<b>OPTIONS</b>	
HTTP Method	<a href="#">Leave-as-is</a> 
<a href="#"> Add Parameter</a> <input checked="" type="checkbox"/> <a href="#">Show Advanced Options</a>	
<b>Endpoint Properties</b>	
<a href="#"> Add Property</a>	
<a href="#">Save &amp; Close</a> <a href="#">Save in Registry</a> <a href="#">Cancel</a>	

For information on configuring advanced options for the endpoint, see [Advanced Options](#).

For descriptions of the parameters and properties of the endpoint, see [Adding a New Endpoint Template](#).

See also [HTTP Endpoint](#).

#### WSDL Endpoint Template

The **WSDL Endpoint Template** has the following options:

- **Template Name** - The unique name for the [endpoint template](#).
- **Name** - A name for the inline endpoint.
- **Specify As** - The method to specify the WSDL. The available values are:
  - **In-lined WSDL** - Paste the WSDL in the text box that appears when this option is selected.
  - **URI** - Activates the WSDL URI field.
- **WSDL URI** - The URI of the WSDL.
- **Service** - The service selected from the available services for the WSDL.
- **Port** - The port selected for the service specified in the above field. In a WSDL an endpoint is bound to each port inside each service.

### WSDL Endpoint Template

WSDL Endpoint [Switch to source view](#)

Template Name *	<input type="text"/>
Name *	<input type="text"/>
Specify As	<input checked="" type="radio"/> In-lined WSDL <input type="radio"/> URI
WSDL URI *	<input type="text"/> <a href="#">Test URI</a>
Service *	<input type="text"/>
Port *	<input type="text"/>
<a href="#">+ Add Parameter</a> <input checked="" type="checkbox"/> Show Advanced Options	
<b>Endpoint Properties</b>	
<a href="#">+ Add Property</a>	
<a href="#">Save</a> <a href="#">Save As</a> <a href="#">Cancel</a>	

See the description of the Endpoint's parameters and properties in [Adding a New Endpoint Template](#).

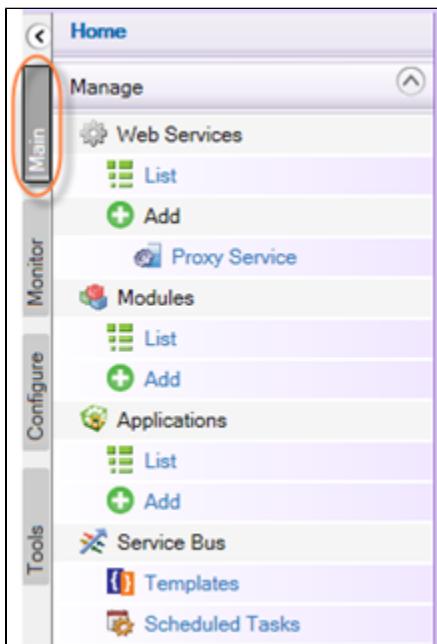
See the Advanced options in [Adding an Endpoint](#).

See also [WSDL Endpoint](#).

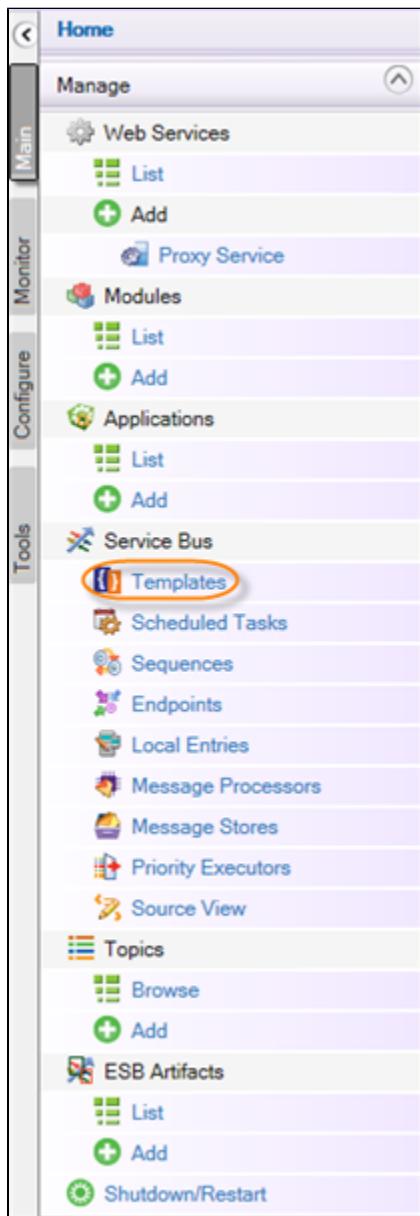
#### Editing an Endpoint Template

Follow the instruction below to edit an endpoint template.

1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Select the "Main" tab to access the "Manage" menu.



3. Click the "Templates" link to access the "Templates" page.



4. On the "Templates" page, choose the "Endpoint Templates" tab to access the available endpoint templates in the synapse configuration, or "Registry Endpoint Templates" to access the templates defined in registry.



or



5. Click the "Edit" link to edit a chosen endpoint template.

Endpoint Template Name	Actions
endpoint_template1	Edit  Delete

6. The "Edit Endpoint Template" page appears. Make the necessary changes.

**Edit Endpoint Template**

Default Endpoint Switch to source view

Template Name *	endpoint_template1
Name *	template
Parameter	Action
name	
uri	

Add Parameter  
 Show Advanced Options

Endpoint Properties  
 Add Property

**Save** **Save As** **Cancel**

See more information in [Adding a New Endpoint Template](#).

7. Click the "Save" or the "Save As" button to save changes.

**Save** **Save As**

8. The "Save As" button allows to save the template in Governance Registry or Configuration Registry. Choose the necessary one.



9. Click "Save."

**Save** **Cancel**

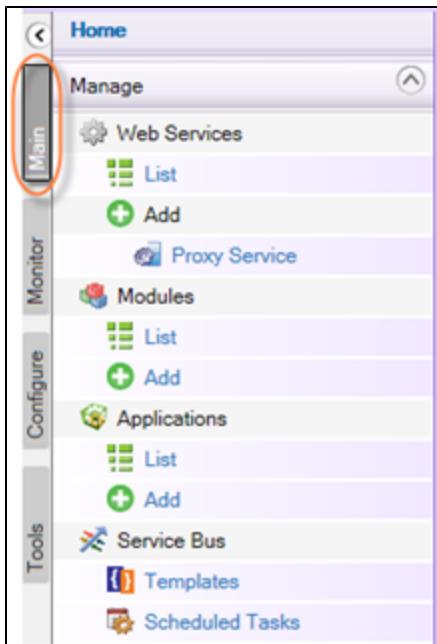
### Note

Statistics/tracing options are not available for endpoints templates.

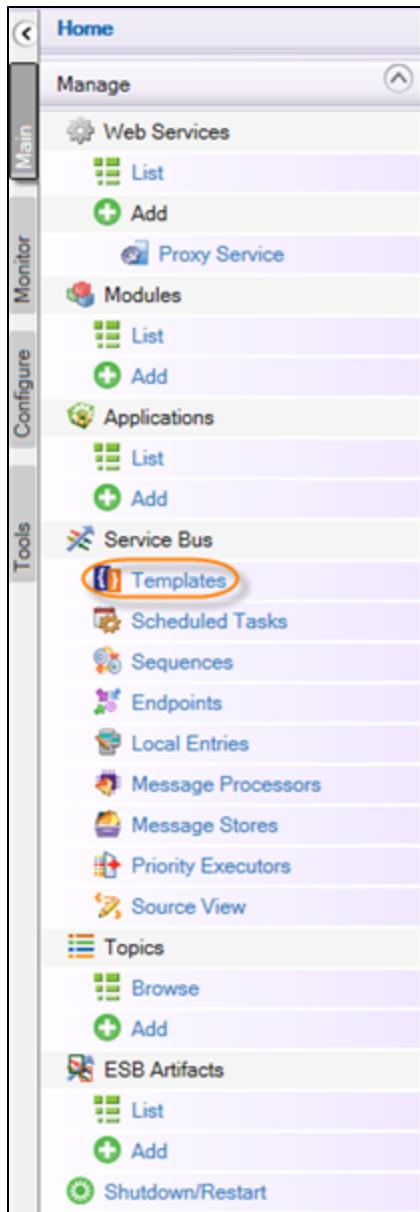
## Deleting an Endpoint Template

If you do not need a particular endpoint template, you can remove it from the system. Follow the instructions below to remove an [endpoint template](#).

1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Select the "Main" tab to access the "Manage" menu.



3. Click the "Templates" link to access the "Templates" page.



4. On the "Templates" page, choose the "Endpoint Templates" tab to access the available endpoint templates in the synapse configuration, or "Registry Endpoint Templates" to access the templates defined in registry.



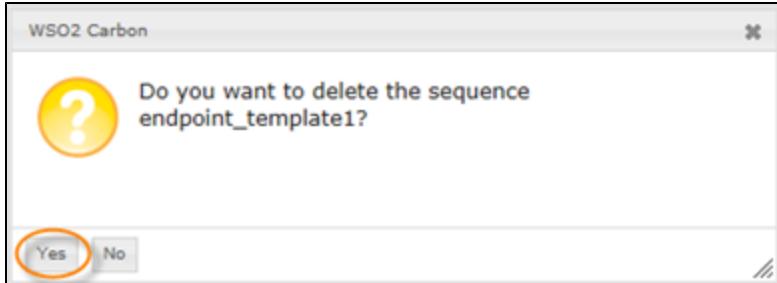
or



5. Click the "Delete" link to delete a chosen endpoint template.

Endpoint Template Name	Actions
endpoint_template1	Edit  Delete

6. The "WSO2 Carbon" window appears. Click "Yes" to confirm your request.



## Note

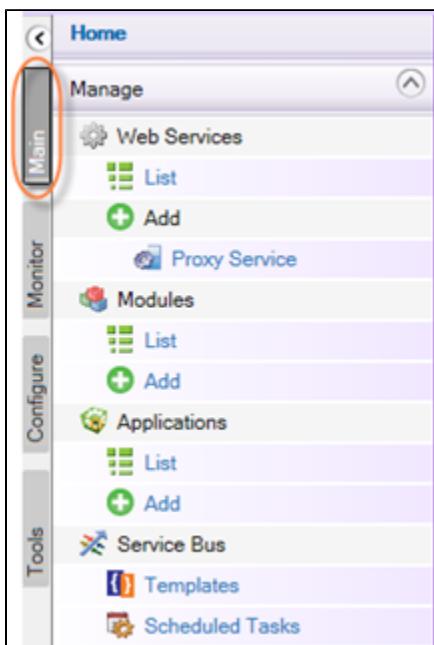
Statistics/tracing options are not available for endpoints templates.

### Adding a New Sequence Template

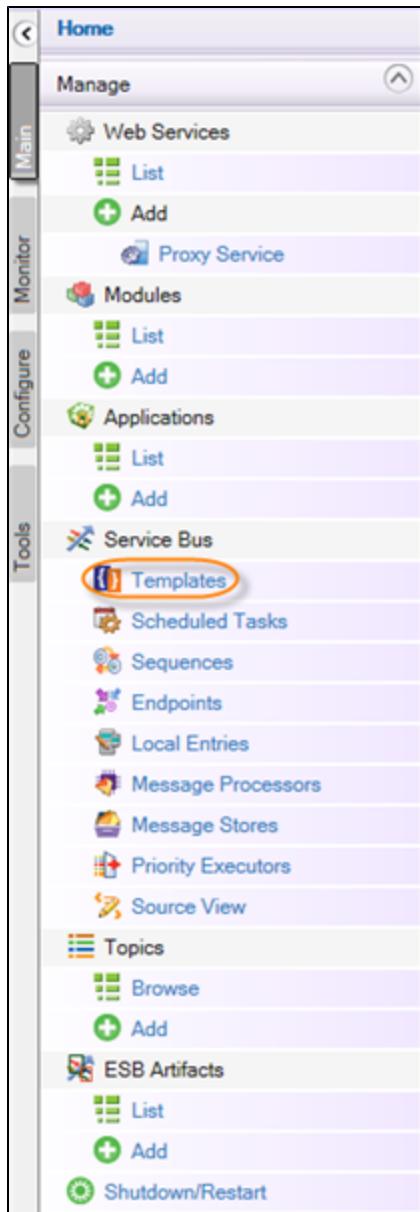
A [sequence template](#) is a parametrized sequence defined in ESB. Parameters of a template are defined in the form of XPath statements. Callers can invoke such a template by populating parameters with static values/XPath expressions.

Follow the instruction below to add a new sequence template.

1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Select the "Main" tab to access the "Manage" menu.



3. Click the "Templates" link to access the "Templates" page.

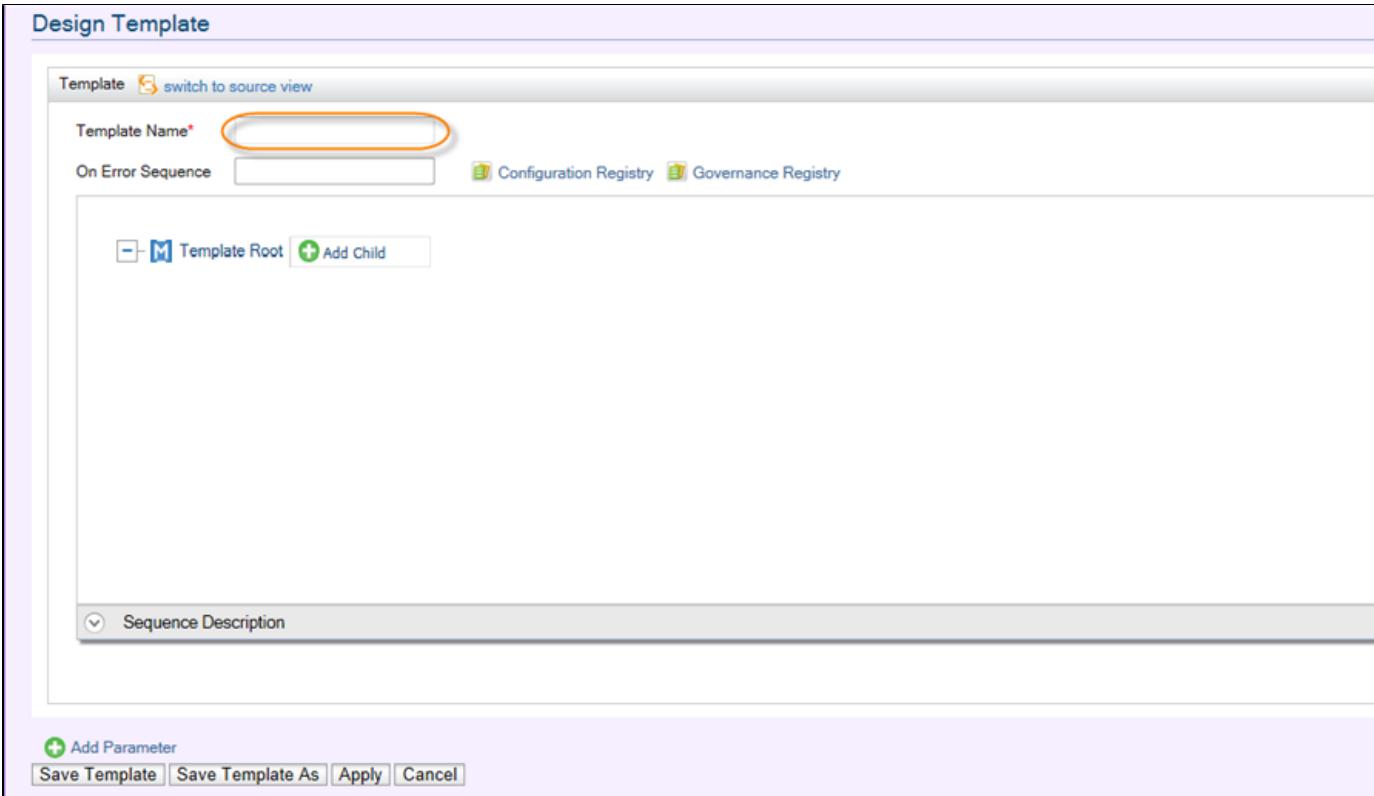


4. On the "Templates" page, click the "Add Sequence Template" link.

The screenshot shows the 'Templates' page. At the top, there are two buttons: '+ Add Sequence Template' (highlighted with a red circle) and '+ Add Endpoint Template'. Below them is a navigation bar with tabs: Sequence Templates (selected), Registry Sequence Templates, Endpoint Templates, and Registry Endpoint Templates. The main content area is titled 'Available Templates in the Synapse Configuration'. It contains a table with two rows:

Sequence Template Name	Actions
MyTemplate	Disable Statistics    Disable Tracing    Edit    Delete
sequence_template	Enable Statistics    Enable Tracing    Edit    Delete

5. The "Design Template" page appears. Enter the template name.



6. Click "Add Child," and select the required nodes for your sequence template.

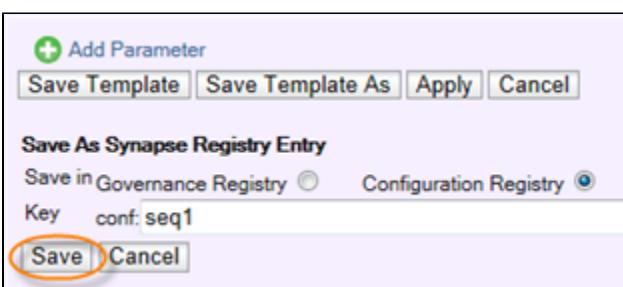


For the detailed information on how to add mediators see [Adding a Mediation Sequence](#).

7. Click the "Save Template" or the "Save Template As" button to add the template.



The "Save Template As" button allows to save the template in Governance Registry or Configuration Registry. Choose the necessary one and click "Save."



### Adding On Error Sequence

ESB executes the specified error handler sequence closest to the point, where the error was encountered.

Follow the instructions below to add an "On error sequence."

1. Click on the "Governance Registry" or the "Configuration Registry" link to select a previously created sequence to be used on error.

**Design Template**

Template switch to source view

Template Name\*

On Error Sequence  Configuration Registry Governance Registry

Template Root Add Child

Sequence Description

Add Parameter Save Template Save Template As Apply Cancel

2. In the displayed Registry window, select a sequence and click on the "OK" button.
3. Click "Save Template," "Save Template As" or "Apply" to add "On Error Sequence."

Add Parameter Save Template Save Template As Apply Cancel

### Adding Parameters to a Sequence Template

1. In case, you want to add parameters to sequence templates, click on the "Add Parameter" link.

Add Parameter Save Template Save Template As Apply Cancel

2. Then a page appear with a parameter table. Specify the parameter name.

Parameter	Action
p1	Delete Parameter
Add Parameter  Save Template  Save Template As  Apply  Cancel	

Such parameter names can be accessed within different child nodes using XPath. The example below shows, how to use this in the **Send Mediator** with XPath.

The screenshot shows the 'Send Mediator' configuration screen. At the top, there's a 'switch to source view' button. Below it, the 'Select Endpoint Type' section has three radio buttons: 'None', 'Define Inline' (selected), and 'Pick From Registry'. A text input field contains '\$func:p1'. To the right of the input field is a 'Namespaces' button. Below the endpoint type section, 'Receiving Sequence Type' is set to 'Default'. A table at the bottom lists a parameter 'p1' with an 'Action' column containing a delete icon and the text 'Delete Parameter'.

## Viewing the XML of a Sequence Template

1. Click "switch to source view" to view the XML of the particular sequence template.



2. In the appeared window, you can see the XML of the sequence template.

The screenshot shows the Synapse XML editor interface. The XML code in the editor is:

```

1 <send xmlns="http://ws.apache.org/ns/synapse">
2 <endpoint key-expression="$func:p1" />
3 </send>
4

```

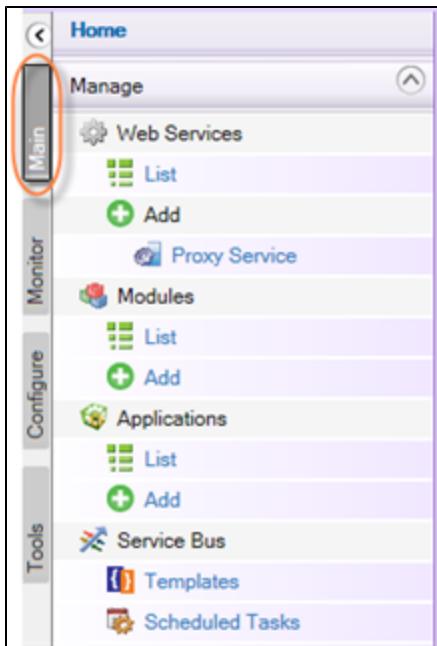
Below the editor, there are status bars for 'Position: Ln 4, Ch 2' and 'Total: Ln 4, Ch 122'. There are also buttons for 'Toggle editor', 'Update', and other toolbar icons.

If you are familiar with the synapse configuration language, you can edit the XML directly and save the config using this view.

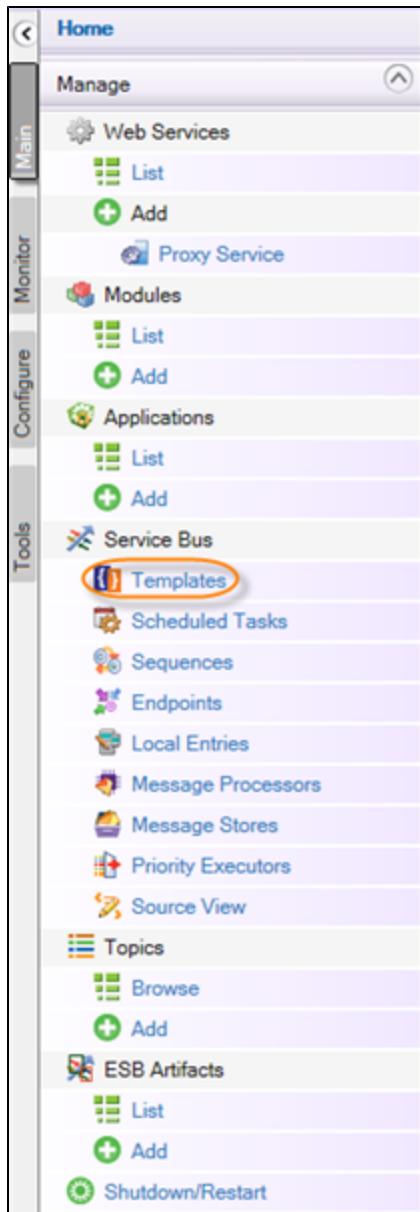
## Editing a Sequence Template

Follow the instructions below to edit a sequence template.

1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Select the "Main" tab to access the "Manage" menu.



3. Click the "Templates" link to access the "Templates" page.



4. On the "Templates" page, choose the "Sequence Templates" tab to access the available endpoint templates in the synapse configuration, or "Registry Sequence Templates" to access the templates defined in registry.

**Sequence Templates**

or

**Registry Sequence Templates**

5. Click the "Edit" link to edit a chosen sequence template.

Sequence Template Name	Actions			
MyTemplate	Disable Statistics	Disable Tracing	Edit (circled in red)	Delete
sequence_template	Enable Statistics	Enable Tracing	Edit	Delete

6. The "Edit Sequence" page appears. Modify the template.

**Edit Sequence**

Sequence switch to source view

Sequence Name\*

On Error Sequence

Configuration Registry Governance Registry

Root Add Child  
 description

Sequence Description

**Save** **Save As** **Apply** **Cancel**

For more information about sequence's options see [Adding a New Sequence Template](#).

7. Click the "Save Template" or the "Save Template As" button to save the alterations.

Add Parameter

**Save Template** **Save Template As** **Apply** **Cancel**

The "Save Template As" button allows to save the template in Governance Registry or Configuration Registry. Choose the necessary one and click "Save."

Add Parameter

**Save Template** **Save Template As** **Apply** **Cancel**

**Save As Synapse Registry Entry**

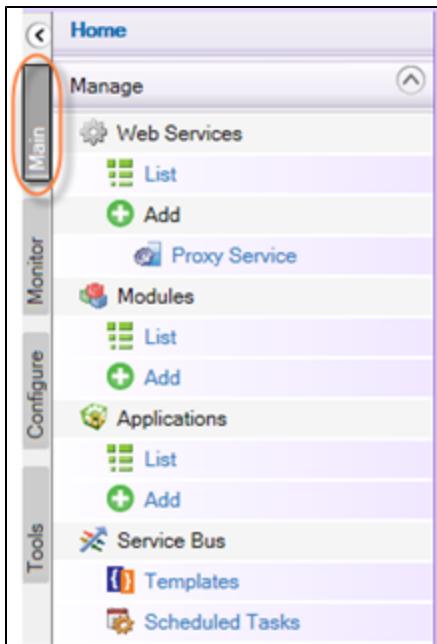
Save in Governance Registry  Configuration Registry   
 Key conf: seq1

**Save** **Cancel**

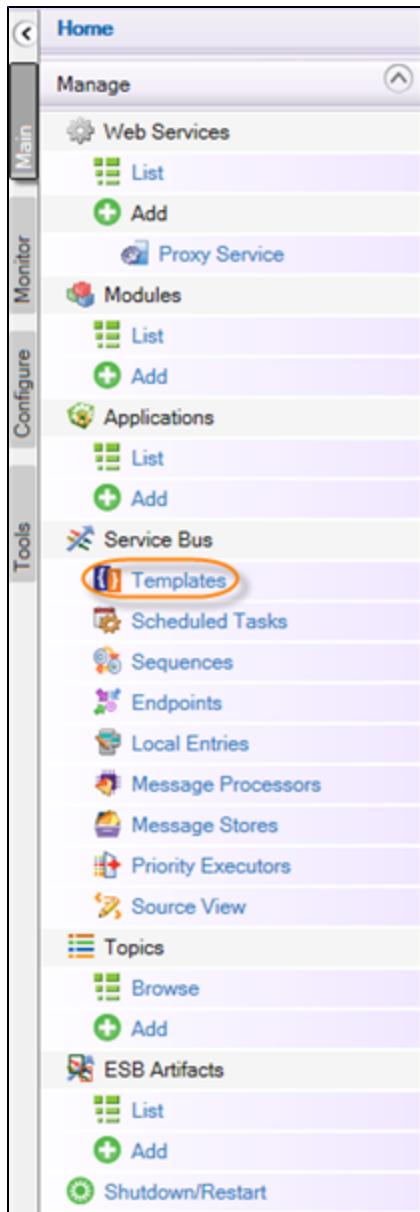
### Deleting a Sequence Template

If you do not need a particular sequence template, you can remove it from the system. Follow the instructions below to do it.

1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Select the "Main" tab to access the "Manage" menu.



3. Click the "Templates" link to access the "Templates" page.



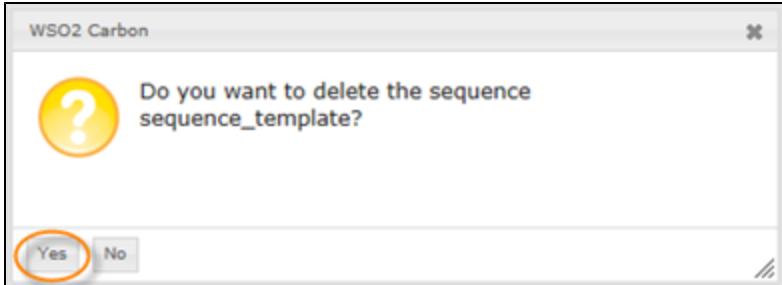
4. On the "Templates" page, choose the "Sequence Templates" tab to access the available endpoint templates in the synapse configuration, or "Registry Sequence Templates" to access the templates defined in registry.

or

5. Click the "Delete" link to delete a sequence template.

Sequence Template Name	Actions			
MyTemplate	Disable Statistics	Disable Tracing	Edit	Delete
sequence_template	Enable Statistics	Enable Tracing	Edit	Delete

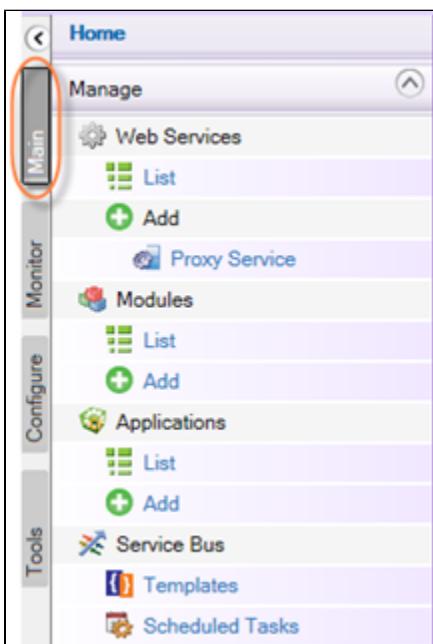
6. The "WSO2 Carbon" window appears. Click "Yes" to confirm your request.



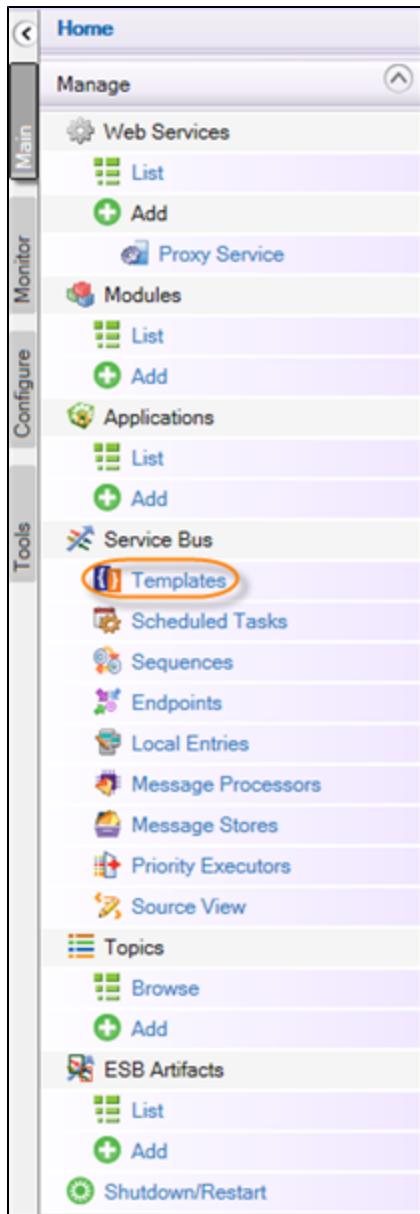
## Managing Sequence Templates

You manage [sequence templates](#), such as configuring statistics and tracing for them, using the ESB Management Console.

1. Sign In. Enter your user name and password to log on to the ESB Management Console.
2. Select the "Main" tab to access the "Manage" menu.



3. Click the "Templates" link to access the "Templates" page.



4. On the "Templates" page, choose the "Sequence Templates" tab.

Templates				
<a href="#">+ Add Sequence Template</a>	<a href="#">+ Add Endpoint Template</a>	<a href="#">Sequence Templates</a>	<a href="#">Registry Sequence Templates</a>	<a href="#">Endpoint Templates</a>
Available Templates in the Synapse Configuration				
Sequence Template Name	Actions			
MyTemplate	<a href="#">Disable Statistics</a>	<a href="#">Disable Tracing</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
sequence_template	<a href="#">Enable Statistics</a>	<a href="#">Enable Tracing</a>	<a href="#">Edit</a>	<a href="#">Delete</a>

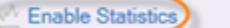
## Note

Defined templates tab shows templates saved in synapse configuration. Registry templates tab shows templates saved in registry.

The appeared page allows to enable statistics and enable tracing for a chosen template.

### Enabling/Disabling Statistics

- To enable statistics for a particular sequence template, click the "Enable Statistics" link.



Enable Statistics

The Enable Statistics option allows to gather information relating to a particular sequence template.

- To disable statistics for a particular sequence template, click the "Disable Statistics" link.



Disable Statistics

### Enabling/Disabling Tracing

- To enable tracing for a particular sequence template, click the "Enable Tracing" link.



Enable Tracing

The Enable Tracing option allows to turn on all trace messages for the corresponding sequence template.

- To disable tracing for a particular sequence template, click the "Disable Tracing" link.



Disable Tracing

### Note

The Enable Statistics and Enable Tracing options are turned off by default because they cause a severe performance degradation. Use these options only in a situation where you have to debug a particular problem. Dynamic sequences does not support Enable Statistics and Enable Tracing.

## Working with Transports

A transport is responsible for carrying messages that are in a specific format. WSO2 ESB supports all the [widely used transports](#) including HTTP/s, JMS, and VFS, and domain-specific transports like FIX. All WSO2 transports are directly or indirectly based on the Apache Axis2 transports framework. This framework provides two main interfaces that each transport implementation must implement.

- org.apache.axis2.transport.TransportListener** - Implementations of this interface specify how incoming messages are received and processed before handing them over to the Axis2 engine for further processing.
- org.apache.axis2.transport.TransportSender** - Implementations of this interface specify how a message can be sent out from the Axis2 engine.

Because each transport implementation has to implement the above two interfaces, each transport generally contains a transport receiver/listener implementation and a transport sender implementation. You configure, enable, and manage transport listeners and senders independently of each other. For example, you could enable just the JMS transport sender without having to enable the JMS transport listener.

For more information on transports, see the following topics:

- [Configuring Transports](#)
- [ESB Transports](#)

For information on configuring transport-level security, go to [Configuring Transport Level Security](#) in the WSO2 Administration Guide.

### Configuring Transports

This page describes how to configure transports in the ESB. It contains the following sections:

- **Globally configuring transports for the ESB**
  - Using the axis2.xml file
  - Configuring servlet transports in standalone mode using the catalina-server.xml file
- Configuring transports at the service level

#### ***Globally configuring transports for the ESB***

#### **Using the axis2.xml file**

WSO2 Carbon and all Carbon-based products ship with a configuration file named `axis2.xml`, located in `<PRODUCT_HOME>/repository/conf/axis2` directory. This is similar to the `axis2.xml` file that ships with Apache Axis2 and Apache Synapse and contains the global configuration for WSO2 Carbon and the Carbon-based products. The `axis2.xml` configuration generally includes configuration details for modules, phases, handlers, global configuration parameters, and transports. The elements `<transportReceiver>` and `<transportSender>` are used to configure transport receivers and senders respectively. Some transports are already configured and enabled by default, including the HTTP and HTTPS transports.

The HTTP and HTTPS transports are required for the ESB management console and other administrative functions. Do not disable these transports.

WSO2 products do not use the HTTP/S servlet transport configurations that are in `axis2.xml` file. Instead, they use Tomcat-level servlet transport configurations in `<PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml` file.

However, products like WSO2 ESB use the HTTP Pass Through transport in `axis2.xml`.

Given below is an example of the JMS transport receiver configuration in the `axis2.xml` file.

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory">
 <parameter
name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFact
ory</parameter>
 <parameter
name="java.naming.provider.url">tcp://localhost:61616</parameter>
 <parameter
name="transport.jms.ConnectionFactoryJNDIName">TopicConnectionFactory</parameter>
 </parameter>

 <parameter name="myQueueConnectionFactory">
 <parameter
name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFact
ory</parameter>
 <parameter
name="java.naming.provider.url">tcp://localhost:61616</parameter>
 <parameter
name="transport.jms.ConnectionFactoryJNDIName">QueueConnectionFactory</parameter>
 </parameter>

 <parameter name="default">
 <parameter
name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFact
ory</parameter>
 <parameter
name="java.naming.provider.url">tcp://localhost:61616</parameter>
 <parameter
name="transport.jms.ConnectionFactoryJNDIName">QueueConnectionFactory</parameter>
 </parameter>
</transportReceiver>
```

<transportReceiver> has the following attributes and elements:

- **name** - A mandatory attribute which indicates a unique name for the transport receiver.
- **class** - A mandatory attribute which indicates the transport receiver implementation class.
- **parameters** - Configuration parameters for the transport receiver, such as the port in the example above. Parameters should be included as child elements of the <transportReceiver> element.

Similarly <transportSender> elements can be used to configure and enable transport senders in WSO2 Carbon. Following is the Pass-through transport sender configuration that comes with WSO2 Carbon by default:

```

<transportSender name="http"
class="org.apache.synapse.transport.passthru.PassThroughHttpSender">
 <parameter name="non-blocking" locked="false">true</parameter>
</transportSender>
```

Simply having <transportReceiver> and <transportSender> elements in the axis2.xml file causes those transports to be loaded and activated during server startup. Therefore, any dependency JARs required by those transport implementations must be included in the server classpath (such as the <ESB\_HOME>/repository/components/dropins directory) to prevent the server from running into exceptions at startup. Additionally, an inaccurate transport configuration (such as a wrong parameter value) might cause the transport not to be enabled properly.

The axis2.xml file will be loaded into memory at server startup only. Therefore, any changes made to the file while the server is running have no effect until you restart the server.

## Configuring servlet transports in standalone mode using the catalina-server.xml file

In addition to configuring transports in axis2.xml, you can globally configure transport receivers using the <PRODUCER\_HOME>/repository/conf/tomcat/catalina-server.xml file when the server is running in standalone mode (that is, the server is not running in a servlet container). By default, you will find the HTTP and HTTPS servlet transports configured in this file. The XML syntax to configure transports in the catalina-server.xml file is similar to the syntax used in the axis2.xml file, except the <transport> element takes the place of the <transportReceiver> element. The default HTTP receiver configuration specified in the catalina-server.xml file is as follows:

```
<transport name="http" class="org.wso2.carbon.server.transports.http.HttpTransport">
 <parameter name="port">9763</parameter>
 <parameter name="maxHttpHeaderSize">8192</parameter>
 <parameter name="maxThreads">150</parameter>
 <parameter name="minSpareThreads">25</parameter>
 <parameter name="maxSpareThreads">75</parameter>
 <parameter name="enableLookups">false</parameter>
 <parameter name="disableUploadTimeout">false</parameter>
 <parameter name="clientAuth">false</parameter>
 <parameter name="maxKeepAliveRequests">100</parameter>
 <parameter name="acceptCount">100</parameter>
 <parameter name="compression">force</parameter>
 <parameter name="compressionMinSize">2048</parameter>
 <parameter name="noCompressionUserAgents">ozilla, traviata</parameter>
 <parameter name="compressableMimeType">

 text/html, text/javascript, application/x-javascript, application/javascript, application/xml, text/css, application/xslt+xml, text/xsl, image/gif, image/jpg, image/jpeg
 </parameter>
</transport>
```

Transport receivers configured in catalina-server.xml will be loaded through a special Transport Manager implementation. Therefore, the classes specified in the <transport> element must implement the interface org.wso2.carbon.server.transports.Transport. Currently, only the default servlet transports can be configured from the catalina-server.xml file.

## Configuring transports at the service level

WSO2 Carbon and Carbon based products allow the user to configure transports at the service level. Transports configured at the service level affect individual services and not all the services deployed in the ESB. Some transport implementations such as JMS and FIX mandate configuring certain transport parameters at the service level.

To configure a transport at the service level, you edit the service's XML file or specify the service-level transport parameters in the proxy service configuration in the management console. For more information, see [Working with Web Services](#).

## ESB Transports

WSO2 ESB supports a variety of transports which makes WSO2 ESB capable of receiving and sending messages over a multitude of transport and application level protocols.

The following transport implementations are supported:

- [HTTP PassThrough Transport](#)

- [HTTP-NIO Transport](#)
- [HTTPS-NIO Transport](#)
- [HTTP Servlet Transport](#)
- [HTTPS Servlet Transport](#)
- [FIX Transport](#)
- [JMS Transport](#)
- [VFS Transport](#)
- [Local Transport](#)
- [MailTo Transport](#)
- [MSMQ Transport](#)
- [RabbitMQ AMQP Transport](#)
- [TCP Transport](#)
- [UDP Transport](#)
- [HL7 Transport](#)
- [Multi-HTTPS Transport](#)
- [MQ Telemetry Transport](#)
- [WebSocket Transport](#)

The links discuss these transport implementations in detail, highlighting the configuration parameters associated with them. For more information on configuring transports, see [Working with Transports](#).

### HTTP PassThrough Transport

**HTTP PassThrough Transport** is a non-blocking HTTP transport implementation based on HTTP Core NIO and specially designed for streaming messages. It is similar to the old message relay transport, but it does not care about the content type and simply streams all received messages through. It also has a simpler and cleaner model for forwarding messages back and forth. It can be used as an alternative to the NHTTP transport.

#### Connection throttling

With the HTTP PassThrough and HTTP NIO transports, you can enable connection throttling to restrict the number of simultaneous open connections. To enable connection throttling, edit the `<PRODUCT_HOME>/repository/conf/nhttp.properties` (for the HTTP NIO transport) or `<PRODUCT_HOME>/repository/conf/passthru-httpp.properties` (for the PassThrough transport) and add the following line:

```
max_open_connections = 2
```

This will restrict simultaneous open incoming connections to 2. To disable throttling, delete the `max_open_connections` setting or set it to -1.

Connection throttling is never exact. For example, setting this property to 2 will result in roughly two simultaneous open connections at any given time.

### HTTP-NIO Transport

**HTTP-NIO transport** is a module of the Apache Synapse project. The two classes that implement the receiver and sender APIs are `org.apache.synapse.transport.nhttp.HttpCoreNIOListener` and `org.apache.synapse.transport.nhttp.HttpCoreNIOListener` respectively. These classes are available in the JAR file named `synapse-nhttp-transport.jar`. The transport implementation is based on Apache HTTP Core - NIO and uses a configurable pool of non-blocking worker threads to grab incoming HTTP messages off the wire.

#### Transport receiver parameters

In transport parameter tables, literals displayed in italic mode under the "Possible Values" column should be considered as fixed literal constant values. Those values can be directly put in transport configurations.

Parameter Name	Description	Required	Possible Values	Default Value
port	The port on which this transport receiver should listen for incoming messages.	No	A positive integer less than 65535	8280
non-blocking	Setting this parameter to true is vital for a number of scenarios to work properly.	Yes	<i>true/false</i>	<i>true</i>
bind-address	The address of the interface to which the transport listener should bind.	No	A host name or an IP address	127.0.0.1
hostname	The host name of the server to be displayed in service EPRs, WSDLs etc. This parameter takes effect only when the WSDLEPRPrefix parameter is not set.	No	A host name or an IP address	localhost
WSDLEPRPrefix	A URL prefix which will be added to all service EPRs and EPRs in WSDLs etc.	No	A URL of the form <protocol>://<hostname>:<port>/	

### Transport sender parameters

Parameter Name	Description	Required	Possible Values	Default Value
http.proxyHost	If the outgoing messages should be sent through an HTTP proxy server, use this parameter to specify the target proxy.	No	A host name or an IP address	
http.proxyPort	The port through which the target proxy accepts HTTP traffic.	No	A positive integer less than 65535	
http.nonProxyHosts	The list of hosts to which the HTTP traffic should be sent directly without going through the proxy.	No	A list of host names or IP addresses separated by ' '	
non-blocking	Setting this parameter to true is vital for a number of scenarios to work properly.	Yes	<i>true/false</i>	<i>true</i>

For more information, see [Working with Transports](#).

### Connection throttling

With the HTTP PassThrough and HTTP NIO transports, you can enable connection throttling to restrict the number of simultaneous open connections. To enable connection throttling, edit the `<PRODUCT_HOME>/repository/conf/nhttp.properties` (for the HTTP NIO transport) or `<PRODUCT_HOME>/repository/conf/passthru-htt.p.properties` (for the PassThrough transport) and add the following line:

```
max_open_connections = 2
```

This will restrict simultaneous open incoming connections to 2. To disable throttling, delete the `max_open_connections` setting or set it to -1.

Connection throttling is never exact. For example, setting this property to 2 will result in roughly two simultaneous open connections at any given time.

## HTTPS-NIO Transport

HTTPS-NIO transport is also a module that comes from the Apache Synapse code base. The receiver class is named as follows:

```
org.apache.synapse.transport.nhttp.HttpCoreNIOSSLListener
```

The sender class is named as follows:

```
org.apache.synapse.transport.nhttp.HttpCoreNIOSSLSender
```

These classes are available in the `synapse-nhttp-transport.jar` bundle. These classes simply extend the **HTTP-NIO** implementation by adding SSL support to it. Therefore, they support all the configuration parameters supported by the **HTTP-NIO** receiver and sender plus the parameters in the following table. The sender can also verify certification revocation. For general information on transports, see [Working with Transports](#).

Transport Parameters (Common to both receiver and the sender):

Parameter Name	Description	Required	Possible Values
keystore	The default keystore to be used by the receiver or the sender should be specified here along with its related parameters as an XML fragment. The path to the keystore file, its type and the passwords to access the keystore should be stated in the XML. The keystore would be used by the transport to initialize a set of key managers.	Yes	<parameter name="keystore"> <KeyStore> <Location>lib/identity.jks</Location> <Type>JKS</Type> <Password>password</Password> <KeyPassword>password</KeyPassword> </KeyStore> </parameter>
truststore	The default trust store to be used by the receiver or the sender should be specified here along with its related parameters as an XML fragment. The location of the trust store file, its type and the password should be stated in the XML body. The truststore is used by the transport to initialize a set of trust managers.	Yes	<parameter name="truststore"> <TrustStore> <Location>lib/identity.jks</Location> <Type>JKS</Type> <Password>password</Password> </TrustStore> </parameter>

customSSLProfiles	The HTTPS NIO transport supports the concept of custom SSL profiles. An SSL profile is a user defined keystore-truststore pair. Such an SSL profile can be associated with one or more target servers. For example, when the HTTPS sender connects to a target server, it will use the SSL profile associated with the target server. If no custom SSL profiles are configured for the target server, the default keystore-truststore pair will be used. Using this feature the NIO HTTPS sender can connect to different target servers using different certificates and identities. The given example only contains a single SSL profile, but there can be as many profiles as required. Each profile must be associated with at least one target server. If a profile should be associated with multiple target servers, the server list should be specified as a comma-separated list. A target server is identified by a host-port pair.	No	<parameter name="customSSLProfiles"><profile><servers>www.test.org:80, www.test2.com:9763</servers><KeyStore><Location>/path/to/identity/store</Location><Type>JKS</Type><Password>password</Password><KeyPassword>password</KeyPassword></KeyStore><TrustStore><Location>/path/to/trust/store</Location><Type>JKS</Type><Password>password</Password></TrustStore></profile></parameter>
-------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Verifying certificate revocation

The HTTPS-NIO transport sender can verify with the certificate authority whether the certificate is still trusted before it completes the SSL connection. If the certificate authority has revoked the certificate, the connection will not be completed. To enable this feature, add the `CertificateRevocationVerifier` parameter to the sender as follows:

```
<transportSender name="https"
class="org.apache.synapse.transport.nhttp.HttpCoreNIOSSLSSender">
...
<parameter name="CertificateRevocationVerifier">true</parameter>
</transportSender>
```

When this parameter is set to true, Synapse attempts to use Online Certificate Status Protocol (OCSP) to perform the verification with the certificate authority at the handshake phase of SSL protocol. If OCSP is not supported by the certificate authority, Synapse uses Certified Revocation Lists (CRL) instead. The verification process checks all the certificates in the certificate chain.

The response from the certificate authority includes the verification and the duration for which the verification is valid. To prevent the performance overhead of continuous HTTP calls, this verification response is cached for the duration specified by the certificate authority so that subsequent verification calls are not needed until the response has expired. There are two Least Recently Used (LRU) in-memory caches for OCSP and CRL, which are automatically managed by a dedicated CacheManager thread for each cache. These CacheManagers update expired cache entries and maintain the LRU cache replacement policy.

## HTTP Servlet Transport

The transport receiver implementation of the HTTP transport is available in the Carbon core component. The transport sender implementation comes from the Tomcat http connector. This transport is shipped with WSO2 Carbon and all WSO2 Carbon-based products, which use this transport as the default transport, except WSO2 ESB. By default, we use non-blocking Tomcat Java connector, `org.apache.coyote.http11.Http11NioProtocol`.

- This is a non-blocking HTTP transport implementation, which means that I/O the threads does not get blocked while received messages are processed.
- Although the `axis2.xml` file contains configurations for HTTP/S transports by default, they are not used by WSO2 products. Instead, the products use the HTTP/S transport configurations in Tomcat-level; therefore, changing the HTTP/S configurations in the `axis2.xml` file has no effect.

In the transport parameter tables, the literals displayed in italics under the "Possible Values" column should be considered as fixed literal constant values. Those values can be directly put into the transport configurations.

## Transport Connector Parameters

Parameter Name	Description	Required	Possible Values	Default Value
port	The port number on which this transport receiver should listen for incoming messages.	Yes	A positive integer less than 65535	
proxyPort	When used, this transport listener will accept messages arriving through a HTTP proxy server which listens on the specified proxy port. Apache mod_proxy should be enabled in the proxy server. All the WSDLs generated will contain the proxy port value as the listener port.	No	A positive integer less than 65535	

HTTP servlet transport should be configured in the `<PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml` file. The transport class that should be specified in the `catalina-server.xml` file is as follows:

```
<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"/>
```

Parameter Name	Description	Possible Values	Default Value
port	The port over which this transport receiver listens for incoming messages.	A positive integer less than 65535	9763 for HTTP Connector 9443 for HTTPS Connector
redirectPort	If this Connector is supporting non-SSL requests, and a request is received for which a matching <security-constraint> requires SSL transport, Catalina will automatically redirect the request to the port number specified here.	A positive integer less than 65535	9443

bindOnInit	Controls when the socket used by the connector is bound. By default it is bound when the connector is initiated and unbound when the connector is destroyed. If set to <code>false</code> , the socket will be bound when the connector is started and unbound when it is stopped.		<code>false</code>
proxyPort	When used, this transport listener will accept messages arriving through a HTTP proxy server which listens on the specified proxy port. Apache <code>mod_proxy</code> should be enabled on the proxy server. All the WSDLs generated will contain the proxy port value as the listener port.	A positive integer less than 65535	
maxHttpHeaderSize	The maximum size of the HTTP request and response header in bytes.	A positive integer	4096
acceptorThreadCount	The number of threads to be used to accept connections. Increase this value on a multi CPU machine, although you would never really need more than 2. Also, with a lot of non keep alive connections, you might want to increase this value as well.		2
maxThreads	The maximum number of worker threads created by the receiver to handle incoming requests. This parameter largely determines the number of concurrent connections that can be handled by the transport.	A positive integer	40
minSpareThreads	The minimum number of threads always kept running. If not specified, the default will be used.		50
enableLookups	Use this parameter to enable DNS lookups in order to return the actual host name of the remote client. Disabling DNS lookups at transport level generally improves performance. By default, DNS lookups are disabled.	<code>true, false</code>	<code>false</code>
disableUploadTimeout	This flag allows the servlet container to use a different, longer connection timeout while a servlet is being executed, which in the end allows either the servlet a longer amount of time to complete its execution, or a longer timeout during data upload.	<code>true, false</code>	<code>true</code>
connectionUploadTimeout	Specifies the timeout, in milliseconds, to use while a data upload is in progress. This only takes effect if <code>disableUploadTimeout</code> is set to <code>false</code> .		
clientAuth	Set to <code>true</code> if you want the SSL stack to require a valid certificate chain from the client before accepting a connection. Set to <code>want</code> if you want the SSL stack to request a client Certificate, but not fail if one is not present. A <code>false</code> value (which is the default) will not require a certificate chain unless the client requests a resource protected by a security constraint that uses <code>CLIENT-CERT</code> authentication.	<code>true, false, want</code>	<code>false</code>

maxKeepAliveRequests	The maximum number of HTTP requests which can be pipelined until the connection is closed by the server. Setting this attribute to 1 will disable HTTP/1.0 keep-alive, as well as HTTP/1.1 keep-alive and pipelining. Setting this to -1 will allow an unlimited amount of pipelined or keep-alive HTTP requests.	-1 or any positive integer	100
acceptCount	The maximum queue length for incoming connection requests when all possible request processing threads are in use. Any requests received when the queue is full will be refused.	A positive integer	10
server	Overrides the Server header for the http response. If set, the value for this attribute overrides the Tomcat default and any Server header set by a web application. If not set, any value specified by the application is used.	Any string	WSO2 Carbon Server
compression	<p>The <b>Connector</b> may use HTTP/1.1 GZIP compression in an attempt to save server bandwidth.</p> <p>The acceptable values for the parameter is "off" (disable compression), "on" (allow compression, which causes text data to be compressed), "force" (forces compression in all cases), or a numerical integer value (which is equivalent to "on", but specifies the minimum amount of data before the output is compressed). If the content-length is not known and compression is set to "on" or more aggressive, the output will also be compressed. If not specified, this attribute is set to "off".</p>	on, off, force	off
compressionMinSize	If <b>compression</b> is set to "on" then this attribute may be used to specify the minimum amount of data before the output is compressed.	A positive integer	2048
noCompressionUserAgents	Indicate a list of regular expressions matching user-agents of HTTP clients for which compression should not be used, because these clients, although they do advertise support for the feature, have a broken implementation.	A comma separated list of regular expressions	empty string
compressableMimeType	Use this parameter to indicate a list of MIME types for which HTTP compression may be used.	A comma separated list of valid mime types	text/html, text/xml, text/plain
URIEncoding	This specifies the character encoding used to decode the URI bytes, after %xx decoding the URL.	URI encoding Character set name	ISO-8859-1

This servlet transport implementation can be further tuned up using the following parameters.

This is only a subset of all the supported parameters. The servlet HTTP transport uses the [org.apache.catalina.connector.Connector](#) implementation from Apache Tomcat. So the servlet HTTP transport actually accepts any parameter accepted by the connector implementation. For a complete list of supported parameters, see [Apache Tomcat's connector configuration reference](#).

## Transport Sender Parameters

Parameter Name	Description	Required	Possible Values	Default Value
PROTOCOL	The version of HTTP protocol to be used for outgoing messages.	No	<i>HTTP/1.0, HTTP/1.1</i>	HTTP/1.1
Transfer-Encoding	Effective only when the HTTP version is 1.1 (i.e. the value of the PROTOCOL parameter should be <i>HTTP/1.1</i> ). Use this parameter to enable chunking support for the transport sender.	No	<i>chunked</i>	Not Chunked
SocketTimeout	The socket timeout value in milliseconds, for out bound connections.	No	A positive integer	60000 ms
ConnectionTimeout	The connection timeout value in milliseconds, for out bound connections.	No	A positive integer	60000 ms
OmitSOAP12Action	Set this parameter to "true" if you need to disable the soapaction for SOAP 1.2 messages.	No	<i>true, false</i>	false

## HTTPS Servlet Transport

Similar to the HTTP transport, the HTTPS transport consists of a receiver implementation which comes from the Carbon core component and a sender implementation which comes from the Apache Axis2 transport module. In fact, this transport uses exactly the same transport sender implementation as the HTTP transport. So the two classes that should be specified in the configuration are `org.wso2.carbon.core.transports.http.HttpsTransportListener` and `org.apache.axis2.transport.http.CommonsHTTPTransportSender` for the receiver and sender in the specified order. The configuration parameters associated with the receiver and the sender are the same as in HTTP transport. This is also a blocking transport implementation.

However, when using the following class as the receiver implementation, we need to specify the servlet HTTPS transport configuration in the transport's XML file.

- `org.wso2.carbon.core.transports.http.HttpsTransportListener`

The class that should be specified as the transport implementation is `org.wso2.carbon.server.transports.http.HttpsTransport`. In addition to the configuration parameters supported by the HTTP servlet transport, HTTPS servlet transport supports the following configuration parameters:

In transport parameter tables, literals displayed in italic mode under the "Possible Values" column should be considered as fixed literal constant values. Those values can be directly put in transport configurations.

Parameter Name	Description	Required	Possible Values	Default Value
sslProtocol	Transport level security protocol to be used.	No	<i>TLS, SSL</i>	TLS
keystore	Path to the keystore which should be used for encryption/decryption.	Yes	A valid file path to a keystore file	
keypass	Password to access the specified keystore.	Yes	A valid password	

Similar to the servlet HTTP transport, this transport is also based on Apache Tomcat's connector implementation. Please refer Tomcat connector configuration reference for a complete list of supported parameters.

Although the `axis2.xml` file contains configurations for HTTP/S transports by default, they are not used by

WSO2 products. Instead, the products use the HTTP/S transport configurations in Tomcat-level; therefore, changing the HTTP/S configurations in the `axis2.xml` file has no effect.

## FIX Transport

**FIX (Financial Information eXchang) transport** implementation is a module developed under the Apache Synapse project. This transport is mainly used with WSO2 ESB in conjunction with proxy services. The following class acts as the transport receiver:

- `org.apache.synapse.transport.fix.FIXTransportListener`

The `org.apache.synapse.transport.fix.FIXTransportSender` acts as the transport sender implementation. These classes can be found in the `synapse-fix-transport.jar` file. The transport implementation is based on Quickfix/J open source FIX engine and hence the following additional dependencies are required to enable the FIX transport.

- `mina-core.jar`
- `quickfixj-core.jar`
- `quickfixj-msg-fix40.jar`
- `quickfixj-msg-fix41.jar`
- `quickfixj-msg-fix42.jar`
- `quickfixj-msg-fix43.jar`
- `quickfixj-msg-fix44.jar`
- `slf4j-api.jar`
- `slf4j-log4j12.jar`

This transport supports JMX ([New in version 4.0](#)).

Download Quickfix/J from [here](#) and in the distribution archive you will find all the dependencies listed above. Also please refer to Quickfix/J documentation on configuring FIX acceptors and initiators.

FIX transport does not support any global parameters. All the FIX configuration parameters should be specified at service level.

QuickFix 4J configuration parameters can be found [here](#)

### Service Level FIX Parameters

#### Tip

In transport parameter tables, literals displayed in italic mode under the "Possible Values" column should be considered as fixed literal constant values. Those values can be directly put in transport configurations.

Parameter Name	Description	Required	Possible Values	Default Value
<code>transport.fix.AcceptorConfigURL</code>	URL to the Quickfix/J acceptor configuration file (see notes below).	Required for receiving messages over FIX	A valid URL	
<code>transport.fix.InitiatorConfigURL</code>	URL to the Quickfix/J initiator configuration file (see notes below).	Required for sending messages over FIX	A valid URL	

transport.fix.AcceptorLogFactory	Log factory implementation to be used for the FIX acceptor (Determines how logging is done at the acceptor level).	No	<i>console, file, jdbc</i>	Logging disabled
transport.fix.InitiatorLogFactory	Log factory implementation to be used for the FIX acceptor (Determines how logging is done at the acceptor level).	No	<i>console, file, jdbc</i>	Logging disabled
transport.fix.AcceptorMessageStore	Message store mechanism to be used with the acceptor (Determines how the FIX message store is maintained).	No	<i>memory, file, sleepycat, jdbc</i>	memory
transport.fix.InitiatorMessageStore	Message store mechanism to be used with the initiator (Determines how the FIX message store is maintained).	No	<i>memory, file, sleepycat, jdbc</i>	memory
transport.fix.ResponseDeliverToCompID	If the response FIX messages should be delivered to a location different from the location the request was originated use this property to set the DeliverToCompID field of the FIX messages.	No		
transport.fix.ResponseDeliverToSubID	If the response FIX messages should be delivered to a location different from the location the request was originated use this property to set the DeliverToSubID field of the FIX messages.	No		

transport.fix.ResponseDeliverToLocationID	If the response FIX messages should be delivered to a location different from the location the request was originated use this property to set the DeliverToLocationID field of the FIX messages.	No		
transport.fix.SendAllToInSequence	By default, all received FIX messages (including responses) will be directed to the in sequence of the proxy service.  Use this property to override that behavior.	No	<i>true, false</i>	true
transport.fix.BeginStringValidation	Whether the transport should validate BeginString values when forwrding FIX messages across sessions.	No	<i>true, false</i>	true
transport.fix.DropExtraResponses	In situation where the FIX recipient sends multiple responses per request use this parameter to drop excessive responses and use only the first one.	No	<i>true, false</i>	false

For more information, see the following topics:

- [Working with Transports](#)
- [Setting Up the ESB Samples](#)
- [Sample 257: Proxy Services with the FIX Transport](#)
- [Sample 258: Switching from HTTP to FIX](#)
- [Sample 259: Switch from FIX to HTTP](#)
- [Sample 260: Switch from FIX to AMQP](#)
- [Sample 261: Switching between FIX Versions](#)
- [Sample 262: CBR of FIX Messages](#)

## JMS Transport

WSO2 ESB's Java Message Service (JMS) transport allows you to easily send and receive messages to queues and topics of any JMS service that implements the JMS specification.

The JMS transport implementation comes from the WS-Commons Transports project, and it makes use of JNDI to connect to various JMS brokers. As a result, WSO2 ESB can work with any JMS broker that offers JNDI support. All the relevant classes are packed into the `axis2-transport-jms-<version>.jar` and the classes `org.apache.axis2.transport.jms.JMSListener` and `org.apache.axis2.transport.jms.JMSSender` act as the transport receiver and sender respectively.

The JMS transport implementation requires an active JMS server instance to be able to receive and send messages. We recommend using [WSO2 Message Broker](#) or Apache ActiveMQ, but other implementations such as Apache Qpid and Tibco are also supported. For information on how to configure the JMS transport with the most common broker servers that can be integrated with WSO2 ESB, see [Configuring the JMS Transport](#).

### JMS connection factory parameters

Configuration parameters for the JMS receiver and the sender are XML fragments that represent JMS connection factories. Following is a typical JMS configuration that uses [WSO2 MB](#) as the message broker:

```
<parameter name="myTopicConnectionFactory">
 <parameter
 name="java.naming.factory.initial">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 <parameter
 name="java.naming.provider.url">repository/conf/jndi.properties</parameter>
 <parameter
 name="transport.jms.ConnectionFactoryJNDIName">TopicConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType">topic</parameter>
 </parameter>
```

This is a bare minimal JMS connection factory configuration that consists of four connection factory parameters.

The following table describes each JMS connection factory parameter in detail:

### Tip

In transport parameter tables, literals displayed in italic mode under the **Possible Values** column should be considered as fixed literal constant values. Those values can be directly used in transport configurations.

Parameter Name	Description
<code>java.naming.factory.initial</code>	JNDI initial context factory class. The class must implement the <code>java.naming.spi.InitialContextFactory</code> interface.
<code>java.naming.provider.url</code>	URL of the JNDI provider.
<code>java.naming.security.principal</code>	JNDI Username.
<code>java.naming.security.credentials</code>	JNDI password.
<code>transport.Transactionality</code>	Preferred mode of transactionality.
<p><b>N</b></p> <p>In WSO2 ESB, JMS transactions only work either the Callout mediator or the Call mediator in blocking mode.</p>	
<code>transport.UserTxnJNDIName</code>	JNDI name to be used to require user transaction

<code>transport.CacheUserTxn</code>	Whether caching for user transactions should be enabled or not.
<code>transport.jms.SessionTransacted</code>	Whether the JMS session should be transacted or not.
<code>transport.jms.SessionAcknowledgement</code>	JMS session acknowledgment mode.
<code>transport.jms.ConnectionFactoryJNDIName</code>	The JNDI name of the connection factory.
<code>transport.jms.ConnectionFactoryType</code>	Type of the connection factory.
<code>transport.jms.JMSSpecVersion</code>	JMS API version.
<code>transport.jms.UserName</code>	The JMS connection username.
<code>transport.jms.Password</code>	The JMS connection password.
<code>transport.jms.Destination</code>	The JNDI name of the destination.
<code>transport.jms.DestinationType</code>	Type of the destination.
<code>transport.jms.DefaultReplyDestination</code>	JNDI name of the default reply destination.
<code>transport.jms.DefaultReplyDestinationType</code>	Type of the reply destination.
<code>transport.jms.MessageSelector</code>	Message selector implementation.
<code>transport.jms.SubscriptionDurable</code>	Whether the connection factory is subscription durable or not.
<code>transport.jms.DurableSubscriberClientID</code>	The ClientId parameter when using durable subscriptions
<code>transport.jms.DurableSubscriberName</code>	The name of the durable subscriber.
<code>transport.jms.PubSubNoLocal</code>	Whether the messages should be published by the connection they were received.

<code>transport.jms.CacheLevel</code>	The cache level, with which JMS objects should be cached at start up. You can configure this in the <OME>/repository/conf/axis2/axis2.xml if the ESB acts as a producer. Else, you can configure a proxy service parameter, if the ESB acts as a consumer. Following are the possible values for this parameter and the description of each:
	<ul style="list-style-type: none"> <li>• <b>none</b> - None of the JMS objects will be cached.</li> <li>• <b>connection</b> - JMS connection objects will be cached.</li> <li>• <b>session</b> - JMS connection and session objects will be cached.</li> <li>• <b>consumer</b> - JMS connection, session and consumer objects will be cached.</li> <li>• <b>producer</b> - JMS connection, session and producer objects will be cached.</li> <li>• <b>auto</b> - An appropriate cache level will be used based on the transaction strategy.</li> </ul>
<code>transport.jms.ReceiveTimeout</code>	Time to wait for a JMS message during polling. Set this parameter value to a negative integer to wait indefinitely. Set to zero to prevent waiting.
<code>transport.jms.ConcurrentConsumers</code>	Number of concurrent threads to be started to consume messages when polling.
<code>transport.jms.MaxConcurrentConsumers</code>	Maximum number of concurrent threads to use during polling.
<code>transport.jms.IdleTaskLimit</code>	The number of idle runs per thread before it dies off.
<code>transport.jms.MaxMessagesPerTask</code>	The maximum number of successful message reception per thread.
<code>transport.jms.InitialReconnectDuration</code>	Initial reconnection attempts duration in milliseconds.
<code>transport.jms.ReconnectProgressFactor</code>	Factor by which the reconnection duration will be increased.
<code>transport.jms.MaxReconnectDuration</code>	Maximum reconnection duration in milliseconds.
<code>transport.jms.ReconnectInterval</code>	Reconnection interval in milliseconds.
<code>transport.jms.MaxJMSConnections</code>	Maximum cached JMS connections in the producer level.
<code>transport.jms.MaxConsumeErrorRetriesBeforeDelay</code>	Number of retries on consume errors before sleep kicks in.
<code>transport.jms.ConsumeErrorDelay</code>	Sleep delay when a consume error is encountered (in milliseconds).
<code>transport.jms.ConsumeErrorProgression</code>	Factor by which the consume error retry sleep will be increased.

JMS transport implementation has some parameters that should be configured at service level. For example, parameters that should be configured in the service XML files of individual services.

## Service level JMS configuration parameters

Following are some of the common parameters you can configure at the service level.

Parameter Name	Description	Required	Possible Values
transport.jms.ConnectionFactory	Name of the JMS connection factory the service should use.	No	A name of an already defined connection factory.
transport.jms.PublishEPR	JMS EPR to be published in the WSDL.	No	A JMS EPR
transport.jms.ContentType	Specifies how the transport listener should determine the content type of received messages.	No	A simple string value, in which case the received messages always have the specified content type. For more information, see <a href="#">http://wso2.org/jms.html#Service_configuration</a> .
transport.jms.MessagePropertyHyphens	Specifies the action to be taken when there are JMS Message property names that contain hyphens.	No	none - No action will be taken. This is the default. replace - Transport headers with hyphens will be replaced by them as JMS message properties, and the original hyphens will be reintroduced on message delivery. delete - Transport headers with hyphens will be deleted.

For more information, see [Java Message Service \(JMS\) Support](#).

For information on how to tune the JMS transport for better performance, see [Tuning the JMS Transport](#).

### Configuring the JMS Transport

WSO2 ESB can work with any JMS broker that offers JNDI support.

For information on how to configure the most common broker servers that can be integrated with WSO2 ESB, see the following topics:

- [Configure with WSO2 Message Broker](#)
- [Configure with ActiveMQ](#)
- [Configure with IBM WebSphere MQ](#)
- [Configure with IBM WebSphere Application Server](#)
- [Configure with JBossMQ](#)
- [Configure with MSMQ](#)
- [Configure with Tibco EMS](#)
- [Configure with SwiftMQ](#)
- [Configure with WebLogic](#)
- [Configure with HornetQ](#)

- Configure with Multiple Brokers

### **Configure with WSO2 Message Broker**

This page walks you through the steps to follow when configuring WSO2 ESB's JMS transport with WSO2 Message Broker (WSO2 MB). Click the required tab for step by step instructions based on the version of WSO2 MB that you are going to use.

Configure with WSO2 Message Broker 3.0.0 | Configure with WSO2 Message Broker 2.2.0 or lower

Follow the steps below to configure the ESB's JMS transport with WSO2 MB 3.0.0

#### Setting up WSO2 MB

1. Download and install WSO2 MB. For instructions on how to download and install WSO2 MB, see [Getting Started with WSO2 MB](#).

The unzipped WSO2 MB distribution folder will be referred to as <MB\_HOME> throughout the documentation.

#### **Note**

It is not possible to start multiple WSO2 products with their default configurations simultaneously in the same environment. Since all WSO2 products use the same port in their default configuration, there will be port conflicts. Therefore, to avoid port conflicts, apply a port offset in the <MB\_HOME>/repository/conf/carbon.xml file by changing the offset value to 1. For example,

```
<Ports>
 <!-- Ports offset. This entry will set the value of the ports defined
 below to
 the define value + Offset.
 e.g. Offset=2 and HTTPS port=9443 will set the effective HTTPS port to
 9445
 -->
 <Offset>1</Offset>
```

2. Open a command prompt (or a shell in Linux) and go to the <MB\_HOME>/bin directory.
3. Start the Message Broker by executing sh wso2server.sh (on Linux/OS X) or wso2server.bat (on Windows).

#### Setting up WSO2 ESB

1. If you have not already done so, see [Getting Started with WSO2 ESB](#) for details on installing and running WSO2 ESB.
2. To enable the JMS transport of WSO2 ESB to communicate with the Message Broker, edit the <ESB\_HOME>/repository/conf/axis2/axis2.xml file, find the commented <transport receiver> block and uncomment it as shown below.

```

<!--Uncomment this and configure as appropriate for JMS transport support with
WSO2 MB 2.x.x -->
<transportReceiver name="jms"
class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 >
 <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">TopicConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 </parameter>

 <parameter name="myQueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 >
 <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 >
 <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>
</transportReceiver>

```

Also, uncomment the following `<transport sender>` block for JMS in the same file:

```

<!-- uncomment this and configure to use connection pools for sending messages>
<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender" />

```

For more information on the JMS configuration parameters used in the code segments above, see [JMS Connection Factory Parameters](#).

3. Copy the following JAR files from the `<MB_HOME>/client-lib` directory to the `<ESB_HOME>/repository/components/lib` directory.
  - `andes-client-3.0.1.jar`

- geronimo-jms\_1.1\_spec-1.1.0.wso2v1.jar
- org.wso2.securevault-1.0.0-wso2v2.jar

4. Open <ESB\_HOME>/repository/conf/jndi.properties file and make a reference to the running Message Broker as specified below:

- Use **carbon** as the virtual host.
- Define a queue named JMSMS.
- Comment out the topic, since it is not required in this scenario. However, in order to avoid getting the javax.naming.NameNotFoundException:TopicConnectionFactory exception during server startup, make a reference to the Message Broker from the TopicConnectionFactory as well.

For example:

```
register some connection factories
connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.QueueConnectionFactory =
amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5673'
connectionfactory.TopicConnectionFactory =
amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5673'
register some queues in JNDI using the form
queue.[jndiName] = [physicalName]
queue.JMSMS=JMSMS
queue.StockQuotesQueue = StockQuotesQueue
```

5. Ensure that WSO2 Message Broker is running, and then open a command prompt (or a shell in Linux) and go to the <ESB\_HOME>\bin directory.
6. Start the ESB server by executing sh wso2server.sh (on Linux/OS X) or wso2server.bat (on Windows).

Now you have instances of both WSO2 Message Broker and WSO2 ESB configured and running.

Follow the steps below to configure the ESB's JMS transport with WSO2 MB 2.2.0 or lower

Setting up WSO2 MB

1. Download and install WSO2 MB. For instructions on how to download and install WSO2 MB, see [Getting Started with WSO2 MB](#).

The unzipped WSO2 MB distribution folder will be referred to as <MB\_HOME> throughout the documentation.

## Note

It is not possible to start multiple WSO2 products with their default configurations simultaneously in the same environment. Since all WSO2 products use the same port in their default configuration, there will be port conflicts. Therefore, to avoid port conflicts, apply a port offset in the <MB\_HOME>/repository/conf/carbon.xml file by changing the offset value to 1. For example,

```

<Ports>
 <!-- Ports offset. This entry will set the value of the ports defined
 below to
 the define value + Offset.
 e.g. Offset=2 and HTTPS port=9443 will set the effective HTTPS port to
 9445
 -->
 <Offset>1</Offset>

```

- WSO2 MB uses a Cassandra server that is bundled with it by default for storage. However, in a production setup, using an external Cassandra server that is capable of handling large volumes of queues is recommended. When an external Cassandra server is used, point the Message Broker to it by editing the value of the `<connectionString>` in `<MB_HOME>/repository/conf/advanced/andes-virtualhosts.xml` accordingly. For information on setting up WSO2 MB with an external server, see the [Clustering and Deployment Guide](#).

```

<store>
 <class>org.wso2.andes.server.store.CassandraMessageStore</class>
 <username>admin</username>
 <password>admin</password>
 <cluster>ClusterOne</cluster>

 <idGenerator>org.wso2.andes.server.cluster.coordination.TimeStampBasedMessageIdGe
 nerator</idGenerator>
 <connectionString>localhost:9161</connectionString>
</store>

```

- The default message batch size for browser subscriptions of Message Broker is 200. If needed, you can increase it to a larger value by setting the following property in the `<MB_HOME>/repository/conf/advanced/qpid-config.xml` file.

```

<messageBatchSizeForBrowserSubscriptions>100000</messageBatchSizeForBrowserSubscr
 iptions>

```

- Open a command prompt (or a shell in Linux) and go to the `<MB_HOME>\bin` directory.
- Start the Message Broker by executing `sh wso2server.sh` (on Linux/OS X) or `wso2server.bat` (on Windows).

## Setting up WSO2 ESB

- If you have not already done so, see [Installation Guide](#) for details on installing and running WSO2 ESB.
- To enable the JMS transport of WSO2 ESB to communicate with the Message Broker, edit the `<ESB_HOME>/repository/conf/axis2/axis2.xml` file, find the commented `<transport receiver>` block that relates to configuring with WSO2 MB and uncomment it as shown below.

```

<!--Uncomment this and configure as appropriate for JMS transport support with
WSO2 MB 2.x.x -->
<transportReceiver name="jms"
class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 >
 <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">TopicConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 </parameter>

 <parameter name="myQueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 >
 <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 >
 <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>
</transportReceiver>

```

Also, uncomment the following `<transport sender>` block for JMS in the same file:

```

<!-- uncomment this and configure to use connection pools for sending messages>
<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender" />

```

For details on the JMS configuration parameters used in the code segments above, see [JMS Connection Factory Parameters](#).

3. Copy the following JAR files from the `<MB_HOME>/client-lib` folder to `<ESB_HOME>/repository/components/lib` folder. They are client libraries required from Message Broker to ESB.

- andes-client-0.13.wso2v10
- geronimo-jms\_1.1\_spec-1.1.0.wso2v1

4. Open <ESB\_HOME>/repository/conf/jndi.properties file and make a reference to the running Message Broker as specified below:

- Use **carbon** as the virtual host.
- Define a queue named JMSMS.
- Comment out the topic, since it is not required in this scenario. However, in order to avoid getting the javax.naming.NameNotFoundException:TopicConnectionFactory exception during server startup, make a reference to the Message Broker from the TopicConnectionFactory as well.

For example:

```
register some connection factories
connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.QueueConnectionFactory =
amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5673'
connectionfactory.TopicConnectionFactory =
amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5673'
```

5. Ensure that WSO2 Message Broker is running, and then open a command prompt (or a shell in Linux) and go to the <ESB\_HOME>\bin directory.
6. Start the ESB server by executing sh wso2server.sh (on Linux/OS X) or wso2server.bat (on Windows).

Now you have instances of both WSO2 Message Broker and WSO2 ESB configured and running.

### **Configure with ActiveMQ**

This section describes how to configure the WSO2 ESB's JMS transport with ActiveMQ. The following topics are covered:

- Setting up WSO2 ESB and ActiveMQ
- Connecting multiple ActiveMQ brokers
- Configuring Redelivery in ActiveMQ Queues

Setting up WSO2 ESB and ActiveMQ

Follow the instructions below to set up and configure.

1. Download, set up and start [Apache ActiveMQ](#).
2. Follow the [Installation Guide](#) and set up WSO2 ESB.

ActiveMQ should be up and running before starting the ESB.

3. Copy the following client libraries from the <AMQ\_HOME>/lib directory to the <ESB\_HOME>/repository/components/lib directory.

#### **ActiveMQ 5.8.0 and above**

- activemq-broker-5.8.0.jar
- activemq-client-5.8.0.jar

- activemq-kahadb-store-5.8.0.jar
- geronimo-jms\_1.1\_spec-1.1.1.jar
- geronimo-j2ee-management\_1.1\_spec-1.0.1.jar
- geronimo-jta\_1.0.1B\_spec-1.0.1.jar
- hawtbuf-1.9.jar
- Slf4j-api-1.6.6.jar
- activeio-core-3.1.4.jar (available in <AMQ\_HOME>/lib/optional folder)

#### Earlier version of ActiveMQ

- activemq-core-5.5.1.jar
- geronimo-j2ee-management\_1.0\_spec-1.0.jar
- geronimo-jms\_1.1\_spec-1.1.1.jar

4. Next, configure transport listeners and senders in ESB.

If you are using ActiveMQ version 5.12.0 or later, you need to start WSO2 ESB server with the following command:

```
wso2server.sh -Dorg.apache.activemq.SERIALIZABLE_PACKAGES="*"
```

Setting up the JMS listener

To enable the JMS transport listener, un-comment the following listener configuration related to ActiveMQ in <ESB\_HOME>/repository/conf/axis2/axis2.xml file.

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">TopicConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 </parameter>

 <parameter name="myQueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>
 </parameter>
 </parameter>
 </transportReceiver>

```

#### Setting up the JMS sender

To enable the JMS transport sender, un-comment the following configuration in <ESB\_HOME>/repository/conf/axis2/axis2.xml file.

```
<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender" />
```

For details on the JMS configuration parameters used in the code segments above, see [JMS Connection Factory Parameters](#).

The above configurations do not address the problem of transient failures of ActiveMQ message broker. Let's say for some reason ActiveMQ goes down and comes back up after a while. ESB won't reconnect to ActiveMQ but instead it'll throw some errors as requests are sent to ESB till it is restarted. In order to tackle this issue one needs to add the following configuration in the place of the *java.naming.provider.url*,

**failover:tcp://localhost:61616**

This will simply make sure re-connection takes place when ActiveMQ is up and running. Failover prefix is

associated with the Failover Transport of ActiveMQ. For further information about this please try [The Failover Transport](#).

You now have instances of ActiveMQ and WSO2 ESB configured, up and running. Next, refer to section [JMS Usecases](#) for implementation details of various JMS use cases.

Connecting multiple ActiveMQ brokers

The WSO2 ESB can be configured as explained in the following procedure in order to work with two ActiveMQ brokers. In this example, port 61616 is used for one ActiveMQ instance and port 61617 is used for the other.

1. Configure the ESB to work with one ActiveMQ broker as described above.
2. Start another ActiveMQ instance.
3. Add another transport receiver to the <ESB\_Home>/repository/conf/axis2/axis2.xml file as follows. Note that the name of the transport receiver is different to that of the transport receiver already entered. The port specified is 61617.

```

<transportReceiver name="jms1"
class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61617</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">TopicConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 </parameter>

 <parameter name="myQueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61617</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61617</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>
</transportReceiver>
```

4. Set up the JMS sender as explained in [Setting up the JMS sender](#).

Configuring Redelivery in ActiveMQ Queues

When WSO2 ESB is configured to consume messages from an ActiveMQ queue, you have the option to configure

message re-delivery. This is useful when messages are unable to be processed by ESB due to failures.

1. Enable the JMS listener as explained in [Setting up the JMS listener](#).
2. Add the following JMS parameters into the proxy service configuration in WSO2 ESB:

```
<parameter name="redeliveryPolicy.maximumRedeliveries">1</parameter>
<parameter name="transport.jms.DestinationType">queue</parameter>
<parameter name="transport.jms.SessionTransacted">true</parameter>
<parameter name="transport.jms.Destination">JMSToHTTPStockQuoteProxy</parameter>
<parameter name="redeliveryPolicy.redeliveryDelay">2000</parameter>
<parameter name="transport.jms.CacheLevel">consumer</parameter>
```

### Parameters

- `redeliveryPolicy.maximumRedeliveries`: Maximum number of retries for delivering the message. If set to -1 ActiveMQ will retry indefinitely.
- `transport.jms.SessionTransacted`: When set to `true`, this enables the JMS session transaction for the proxy service.
- `redeliveryPolicy.redeliveryDelay`: Delay time in milliseconds between retries.
- `transport.jms.CacheLevel`: This needs to be set to `consumer` for the ActiveMQ redelivery mechanism to work.

3. Add the following line in your fault sequence:

```
<property name="SET_ROLLBACK_ONLY" value="true" scope="axis2"/>
```

## SET\_ROLLBACK\_ONLY

This parameter must be defined for ActiveMQ to redeliver the message.

When ESB is unable to deliver a message to the back-end service due to an error, it will be routed to the fault sequence in the ESB configuration. When "SET\_ROLLBACK\_ONLY" property is set in the fault sequence, ESB informs ActiveMQ to redeliver the message.

Below is a sample proxy service configuration:

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="JMSToHTTPStockQuoteProxy"
 transports="jms"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
 <target>
 <inSequence>
 <property name="transactionID"
 expression="get-property('MessageID')"
 scope="default"/>
 <property name="sourceMessageID"
 expression="get-property('MessageID')"
 scope="default"/>
 <property name="proxyMessageID"
 expression="get-property('MessageID')"
 scope="default"/>
 <log level="full">
 <property name="transactionID"
 expression="get-property('transactionID')"/>
 <property name="sourceMessageID"
 expression="get-property('sourceMessageID')"/>
 <property name="MessageID" expression="get-property('proxyMessageID')"/>
 </log>
 <property name="SET_ROLLBACK_ONLY" value="true" scope="axis2"/>
 <drop/>
 </inSequence>
 <faultSequence="jms_fault">
 </target>
 <parameter name="redeliveryPolicy.maximumRedeliveries">1</parameter>
 <parameter name="transport.jms.DestinationType">queue</parameter>
 <parameter name="transport.jms.SessionTransacted">true</parameter>
 <parameter name="transport.jms.Destination">JMSToHTTPStockQuoteProxy</parameter>
 <parameter name="redeliveryPolicy.redeliveryDelay">2000</parameter>
 <parameter name="transport.jms.CacheLevel">consumer</parameter>
 <description/>
 </proxy>

```

### Configure with IBM WebSphere MQ

The WSO2 JMS transport can be configured with IBM® WebSphere® MQ. The following topics cover the configuration steps.

- Prerequisites
- Creating queue manager, queue and channel in IBM WebSphere MQ
- Generating the .bindings file
- Configuring WSO2 ESB JMS transport
- Copying IBM Websphere MQ libraries
- Deploying JMS listener proxy service
- Testing the proxy service
- Sample Scenarios

The configuration steps below are for a Windows environment. Similar steps apply for other environments such as OSX and Linux.

## Prerequisites

- WSO2 ESB is installed. To test the samples, you must also have Apache Ant installed. For details, see [Installation Prerequisites](#).
- WebSphere MQ is installed and the latest fix pack applied (see the [IBM documentation](#)). The fix pack can be obtained from <http://www-01.ibm.com/software/integration/wmq>.

Creating queue manager, queue and channel in IBM WebSphere MQ

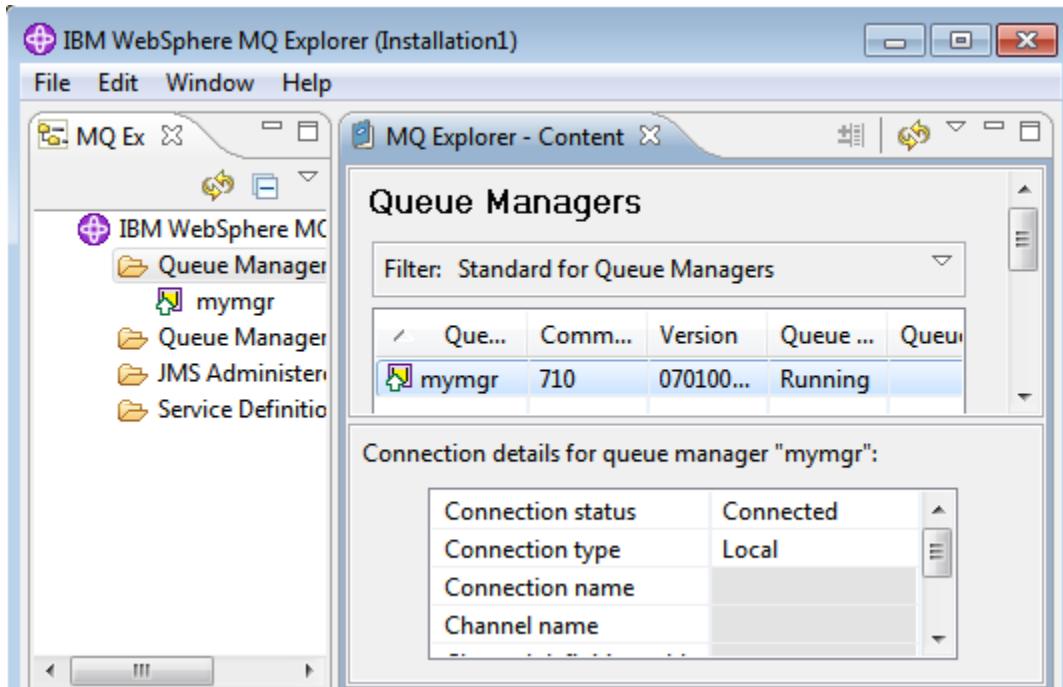
1. Start IBM WebSphere MQ Explorer as an administrator. If you are not running on an administrator account, right-click on the IBM WebSphere MQ icon/menu item and then click **Run as Administrator**.
2. Right-click on **Queue Managers**, move the cursor to **New** and then click **Queue Manager** to open the **Create Queue Manager** wizard. Enter `ESBQManager` as the queue manager name. Make sure you select **make this the default queue manager** check box. Leave the default values unchanged in the other fields. Click **Next** to move to the next page.
3. Click **Next** in the page for entering data and log values without changing any default values.
4. In the page for entering configuration options, select the following. Then click **Next**.

Field Name	Value
<b>Start queue manager after it has been created</b> check box	If this is selected, the queue manager will start running immediately after it is created.
<b>Automatic</b> field	If this is selected, the queue manager is automatically started when the machine starts up.

5. Configure the following in the page for entering listener options.

Field Name	Value
<b>Create listener configured for TCP/IP</b> check box	Select this check box to create the listener.
<b>Listen on port number</b> field	Enter the number of the port where you want to set the listener. In this example, the port number will be 1414.

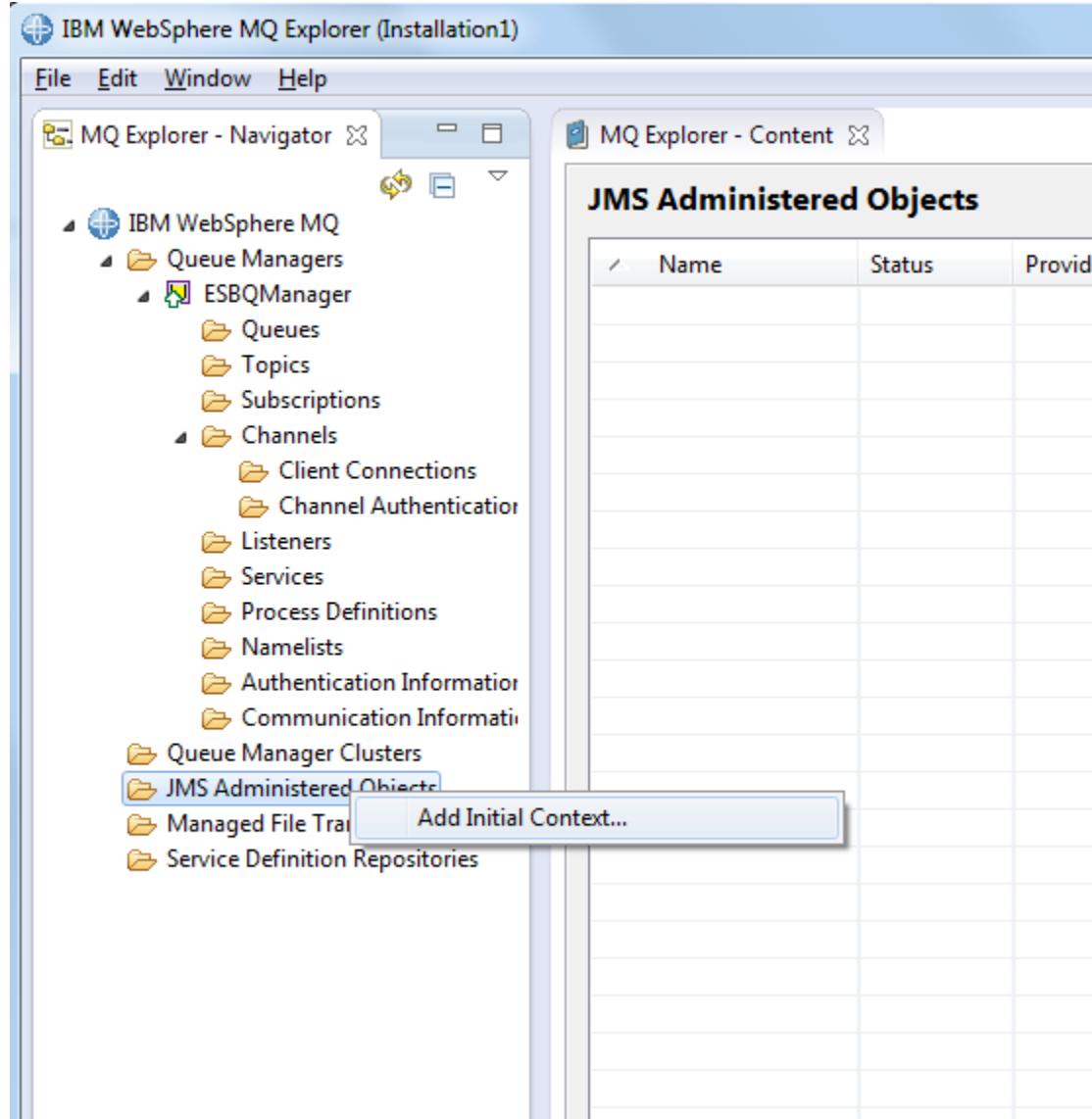
6. Click **Next** and then click **Finish** to save the configuration. The queue manager will be created as shown below.



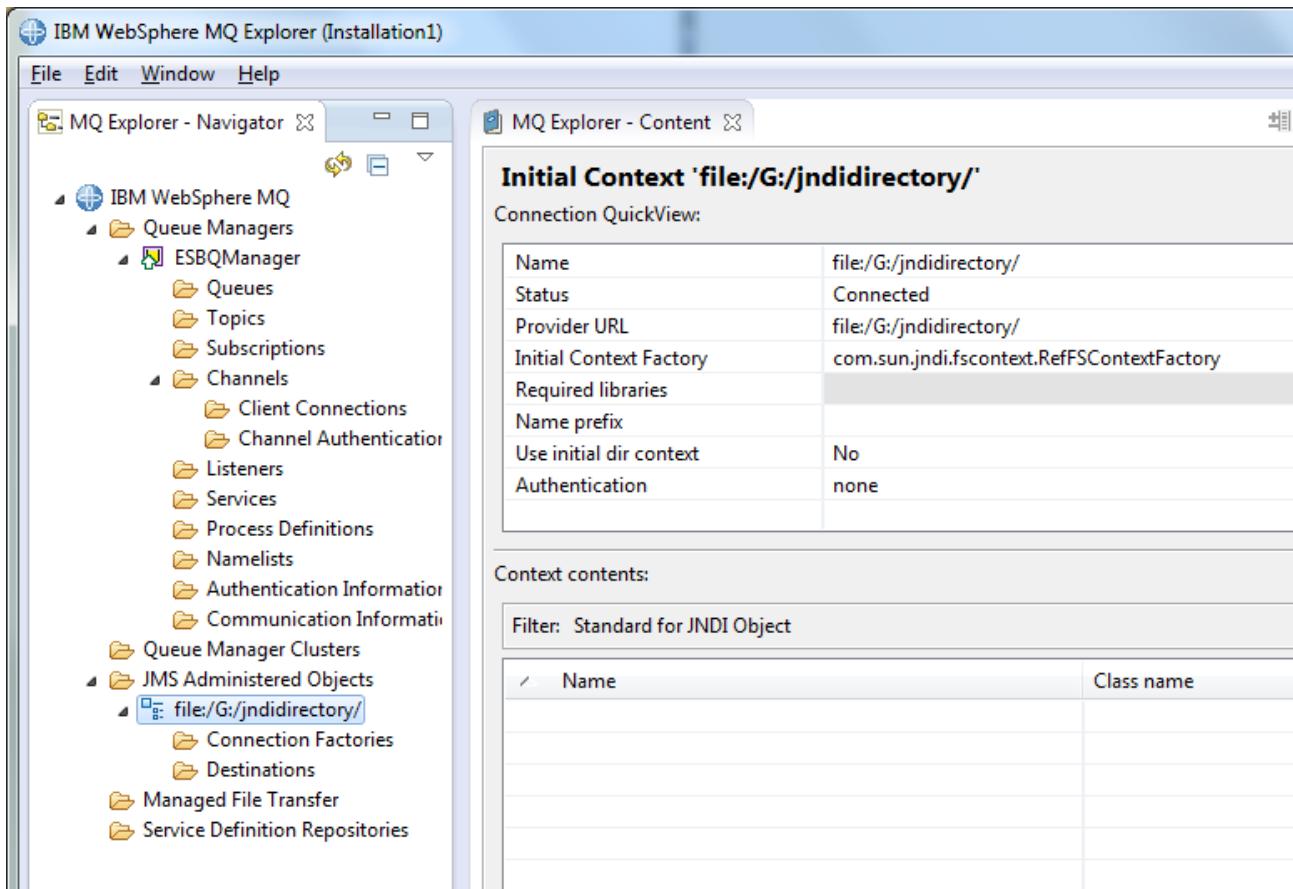
7. Expand the navigation tree of the `ESBQManager` queue manager in the navigation tree. Right-click on **Queues**, move the cursor to **New** and then click **Local Queue** to open the **Create a Local Queue** wizard. Enter the local queue name as `LocalQueue1` and complete running the wizard. Leave the default values of all other fields unchanged, and click **Finish** to save the local queue.
8. Right-click on **Channels**, move the cursor to **New**, and then click **Server-connection Channel** to open the **Create a Server-connection Channel** wizard. Enter `myChannel` as the channel name and click **Next**. Make sure that the value for the **Transmission Protocol** is **TCP**. Leave the default values unchanged for the rest of the fields, and click **Finish** to save the channel.

Generating the .bindings file

1. Create a directory in which the `.bindings` file can be saved in any location of your computer. In this example, a directory named `jndidirectory` will be created in the `G` folder.
2. Go to IBM Websphere MQ, and right-click on **JMS Administered Objects**, and then click **Add Initial Context**.



3. Select the **File system** option in the **Connection Details** wizard. Enter `file:G/jndidirectory` in the **Context nickname** field. Leave the default values unchanged for other fields and complete running the wizard. The new file initial context will be displayed in the left navigator under **JMS Administered Objects** as shown below.



4. Click on the file initial context (named `file:/G:/jndidirectory` in this example) in the navigator to expand it. Right-click on **Connection Factories**, move the cursor to **New**, and then click **Connection Factory**. Enter the name of the connection factory as `MyQueueConnectionFactory`. Select `Queue Connection Factory` as the connection factory type. Select `MQClient` as the transport. Leave the default values unchanged for other fields and complete running the wizard.
5. Right-click on the newly connected connection factory in the left navigator, and click **Properties**. Click **Connection**. Then browse and select `ESBQManager` for the **Base queue manager** field. You can change the host and port name for the connection factory if required. No changes will be made in this example since default values are used. Leave the default values unchanged for other fields and click **OK**.
6. Right-click **Destination** under **JMS Administered Objects** in the left navigator. Move the cursor to **New** and then click **Destination** to open the **New Destination** wizard. In order to map the destination to the local queue you created in step 7 of the [Creating queue manager, queue and channel in IBM WebSphere MQ](#) section, enter the same queue name (`LocalQueue1` in this example) in the **Name** field. Select `Queue` for the **Type** field. Select `ESBQManager` as the queue manager and `LocalQueue1` as the queue in the wizard. Leave the default values unchanged for other fields and complete running the wizard.

The .bindings file will be created in the location you specified (`file:/G:/jndidirectory` in this example) after you carry out the above steps.

In order to connect to the queue, you need to configure channel authentication. Run the following two commands to disable channel authentication for the ease of use. Alternatively, you can configure the authentication for the MQ server.

Note that you have to run the command prompt as a admin user and run these commands.

```
runmqsc ESBQManager
```

```
ALTER QMGR CHLAUTH(DISABLED)
```

```
REFRESH SECURITY TYPE(CONNAUTH)
```

The following will be displayed in the command prompt.

```
C:\Windows\system32>runmqsc ESBQManager
5724-H72 (C) Copyright IBM Corp. 1994, 2014.
Starting MQSC for queue manager ESBQManager.

ALTER QMGR CHLAUTH(DISABLED)
 1 : ALTER QMGR CHLAUTH(DISABLED)
AMQ8005: WebSphere MQ queue manager changed.
REFRESH SECURITY TYPE(CONNAUTH)
 2 : REFRESH SECURITY TYPE(CONNAUTH)
AMQ8560: WebSphere MQ security cache refreshed.
```

Configuring WSO2 ESB JMS transport

1. Add the following transport receiver to the <ESB\_HOME>/repository/conf/axis2/axis2.xml file.

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/G:/jndidirectory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MyQueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.UserName" locked="false">nandika</parameter>
 <parameter name="transport.jms.Password" locked="false">password</parameter>
 </parameter>

 <parameter name="myQueueConnectionFactory1" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/G:/jndidirectory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MyQueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.UserName" locked="false">nandika</parameter>
 <parameter name="transport.jms.Password" locked="false">password</parameter>
 </parameter>
</transportReceiver>

```

2. Add the following transport sender to the <ESB\_HOME>/repository/conf/axis2/axis2.xml file.

```

<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender">
 <parameter name="default" locked="false">
 <parameter name="vender.class.loader.enabled">false</parameter>
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/G:/jndidirectory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MyQueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.UserName" locked="false">nandika</parameter>
 <parameter name="transport.jms.Password" locked="false">password</parameter>
 </parameter>

 <parameter name="myQueueConnectionFactory1" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/G:/jndidirectory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MyQueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.UserName" locked="false">nandika</parameter>
 <parameter name="transport.jms.Password" locked="false">password</parameter>
 </parameter>
 </parameter>
 </transportSender>

```

## Note

If you use the default configuration of the IBM MQ queue manager, you need to provide username and password client authentication. The username and password that you need to provide here is the username and password that you provide to log on to your operating system.

The `vender.class.loader.enabled` parameter in the above configuration should be added only when you use IBM Websphere MQ as the JMS broker.

WSO2 uses some external class loader mechanisms for some external products such as QPID and AMQP due to the limitation of serializing the JMSObject message. However, it is not required to use this mechanism for IBM Websphere MQ. By adding the `vender.class.loader.enabled` parameter, you can skip the external class loader for IBM Websphere MQ.

This property can also be included in a proxy service, REST API, message store, JMS receiver or the Synapse configuration depending on the use case.

### Copying IBM Websphere MQ libraries

Follow the instructions below to build and install IBM WebSphere MQ client JAR files to WSO2 ESB.

These instructions are tested on IBM WebSphere MQ version 8.0.0.4. However, you can follow them for other versions appropriately.

1. Create a new directory named `wmq-client`, and then create another new directory named `lib` inside it.
2. Copy the following JAR files from the `<IBM_MQ_HOME>/java/lib/` directory to the `wmq-client/lib/` directory.

`<IBM_MQ_HOME>` refers to the IBM WebSphere MQ installation directory.

- `com.ibm.mq.allclient.jar`
- `fscontext.jar`
- `jms.jar`
- `providerutil.jar`

3. Create a `POM.xml` file inside the `wmq-client/` directory and add all the required dependencies as shown in the example below.

You need to change the values of the `<version>` and `<systemPath>` properties accordingly.

```

<?xml version="1.0"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>wmq-client</groupId>
 <artifactId>wmq-client</artifactId>
 <version>8.0.0.4</version>
 <packaging>bundle</packaging>
 <dependencies>
 <dependency>
 <groupId>com.ibm</groupId>
 <artifactId>fscontext</artifactId>
 <version>8.0.0.4</version>
 <scope>system</scope>
 <systemPath>${basedir}/lib/fscontext.jar</systemPath>
 </dependency>
 <dependency>
 <groupId>com.ibm</groupId>
 <artifactId>providerutil</artifactId>
 <version>8.0.0.4</version>
 <scope>system</scope>
 <systemPath>${basedir}/lib/providerutil.jar</systemPath>
 </dependency>
 <dependency>
 <groupId>com.ibm</groupId>
 <artifactId>allclient</artifactId>
 <version>8.0.0.4</version>
 <scope>system</scope>
 <systemPath>${basedir}/lib/com.ibm.mq.allclient.jar</systemPath>
 </dependency>
 <dependency>
 <groupId>javax.jms</groupId>
 <artifactId>jms</artifactId>
 <version>1.1</version>
 <scope>system</scope>
 <systemPath>${basedir}/lib/jms.jar</systemPath>
 </dependency>
 </dependencies>

```

```
<build>
 <plugins>
 <plugin>
 <groupId>org.apache.felix</groupId>
 <artifactId>maven-bundle-plugin</artifactId>
 <version>2.3.4</version>
 <extensions>true</extensions>
 <configuration>
 <instructions>

<Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
 <Bundle-Name>${project.artifactId}</Bundle-Name>

<Export-Package>*;-split-package:=merge-first</Export-Package>
 <Private-Package/>
 <Import-Package/>

<Embed-Dependency>*;scope=system;inline=true</Embed-Dependency>
 <DynamicImport-Package>*</DynamicImport-Package>
 </instructions>
 </configuration>
 </plugin>
```

```

</plugins>
</build>
</project>

```

4. Navigate to the `wmq-client` directory using your Command Line Interface (CLI), and execute the following command, to build the project: `mvn clean install`
5. Stop the WSO2 ESB server, if it is already running.
6. Remove any existing IBM MQ client JAR files from the `<ESB_HOME>/repository/components/dropins/` directory and the `<ESB_HOME>/repository/components/lib/` directory.
7. Copy the `<wmq-client>/target/wmq-client-8.0.0.4.jar` file to the `<ESB_Home>/repository/components/dropins/` directory.
8. Copy the `<wmq-client>/target/jta.jar` file to the `<ESB_HOME>/repository/components/lib/` directory.
9. Remove following line from the `<ESB_HOME>/repository/conf/etc/launch.ini` file: `jaxam.jms,\`
10. **Regenerate .bindings file with the Provider Version : 8 property** (if you already generated one before), and replace the existing `.bindings` file (if you have one) with the new `.bindings` file you generated.
11. Start the WSO2 ESB server.

Deploying JMS listener proxy service

In this section, the following simple proxy service is deployed to listen to the `LocalQueue1` queue. When a message is published in this queue, the proxy service would pull the message out of the queue and log it. See [Working with Proxy Services](#) for detailed instructions to create a proxy service.

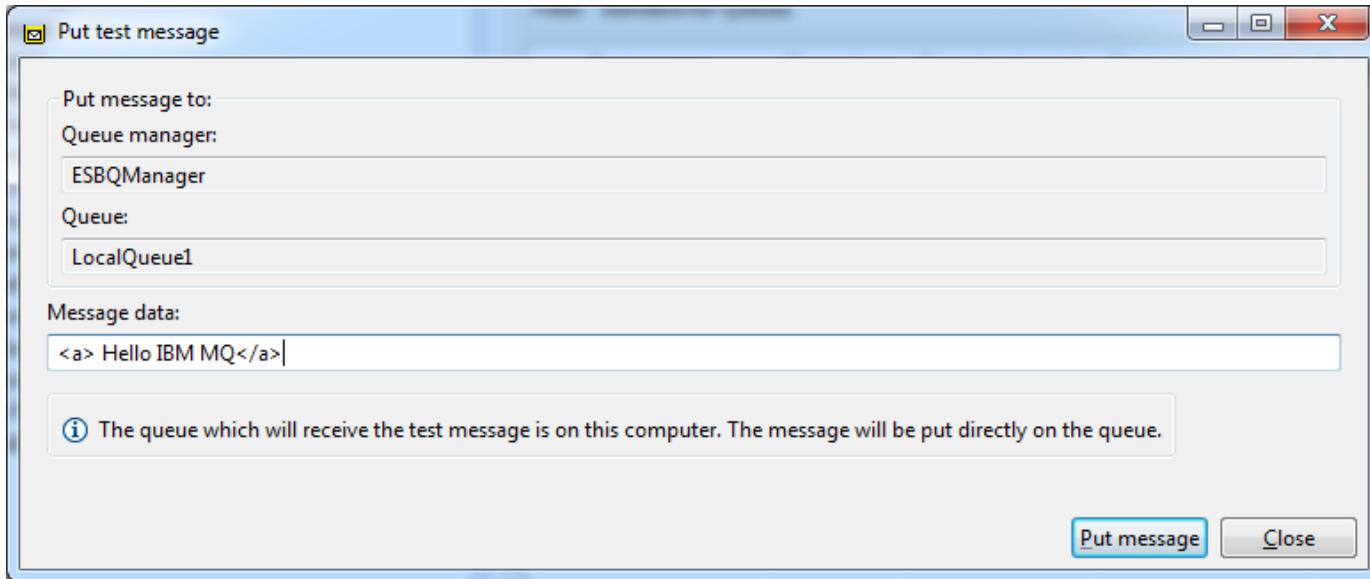
```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="MyJMSProxy"
 transports="jms"
 startOnLoad="true"
 trace="disable">
 <description/>
 <target>
 <inSequence>
 <log level="full"/>
 <drop/>
 </inSequence>
 </target>
 <parameter name="transport.jms.Destination">LocalQueue1</parameter>
</proxy>

```

Testing the proxy service

Open IBM Websphere MQ and publish a message to `LocalQueue1`.



The message will be logged in the ESB Management console as well as the log file.

#### Sample Scenarios

This section describes how to configure the following sample scenarios using the JMS transport, WebSphere MQ, and WSO2 ESB:

- Queue Scenario 1: JMS Client -> Queue -> ESB -> Axis2server
- Queue Scenario 2: JMS Client -> ESB -> Queue -> Axis2server
- Topic Scenario 1: JMS Client -> Topic -> ESB -> Axis2server
- Topic Scenario 2: JMS Client -> ESB -> Topic -> Axis2server

In scenarios where the client places the message directly on the queue or topic, and the message is then picked up by the ESB, you configure the non-default connection factories in <ESB\_HOME>\repository\conf\axis2\axis2.xml and comment them out in the <ESB\_HOME>\samples\axis2Server\repository\conf\axis2.xml file. In scenarios where the client sends the message to the ESB first, and the ESB places the message on the queue or topic, you configure the non-default connection factories in <ESB\_HOME>\samples\axis2Server\repository\conf\axis2.xml and comment them out in <ESB\_HOME>\repository\conf\axis2\axis2.xml.

#### Queue Scenario 1: Client to Queue to ESB

In this scenario, the JMS client places an order on the JMS\_QUEUE queue. The ESB listens on this queue, gets the message, and sends it to the back-end server to process.

1. In <ESB\_HOME>\repository\conf\axis2\axis2.xml, comment out the myTopicConnectionFactory parameter and uncomment the SQProxyCF parameter. It should look as shown below.

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <!--parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter-->

 <parameter name="SQProxyCF" locked="false">
 <parameter
name="java.naming.factory.initial">com.sun.jndi.fscontext.RefFSContextFactory</pa
rameter>
 <parameter
name="java.naming.provider.url">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter>

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
</transportReceiver>

```

2. If you are using version 6.0 of IBM WebSphere MQ, add the following parameter to axis2.xml to ensure that JMS Spec version 1.1.2b is used instead of version 1.1:

```
<parameter name="transport.jms.JMSSpecVersion">1.0.2b</parameter>
```

3. Start the ESB with the **Sample 250** configuration by running the following command.

```
wso2esb-samples.bat -sn 250
```

4. Log in to the server management console at: <https://localhost:9443/carbon/>.

5. Click **Services -> list -> StockQuoteProxy -> edit (Specific Configuration)**

6. Add a service parameter as follows and save it.

```
name = transport.jms.ConnectionFactory value = SQProxyCF
```

7. Go to the <ESB\_HOME>/samples/axis2Client directory and build it using the ant command.

8. Go to the <ESB\_HOME>/samples/axis2Client/src/samples/userguide directory, open the GenericJMSClient.java source file, and make the following changes in the code:

- a. Set the jms\_dest property default value to JMS\_QUEUE (line 45)
  - b. Set the java.naming.provider.url to file:/C:/JNDI-Directory (line 82)
  - c. Set the java.naming.factory.initial to com.sun.jndi.fscontext.RefFSContextFactory (line 85)
  - d. Set the lookup key to MQ\_JMS\_MANAGER (line 89)
9. Configure the proxy configuration so that it appears as follows.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="https http jms" startOnLoad="true"
trace="disable">
 <target>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 <inSequence>
 <property name="OUT_ONLY" value="true" />
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 <parameter name="transport.jms.ContentType">
 <rules>
 <jmsProperty>contentType</jmsProperty>
 <default>application/xml</default>
 </rules>
 </parameter>
 <parameter name="transport.jms.ConnectionFactory">SQProxyCF</parameter>
 </proxy>
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default "fault" sequence"/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
 </sequence>
 <sequence name="main">
 <log/>
 <drop/>
 </sequence>
</definitions>

```

10. Configure <ESB\_HOME>\samples\axis2Server\repository\conf\axis2.xml so that it looks as follows.

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <!--parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter-->

 <!--parameter name="SQProxyCF" locked="false">
 <parameter
name="java.naming.factory.initial">com.sun.jndi.fscontext.RefFSContextFactory</pa
rameter>
 <parameter
name="java.naming.provider.url">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter-->

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/C/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter>
 </parameter>
 </transportReceiver>

```

11. Start the axis2 Server with the following command.

axis2Server.bat

12. Send the request from the JMS client, and the sample Axis2 server console will print a message.

ant jmsclient -Djms\_type=pox -Djms\_dest=JMS\_QUEUE -Djms\_payload=MSFT

Queue Scenario 2: Client to ESB to Queue

In this scenario, the JMS client places an order to the ESB, which then places it on the queue. The back-end server listens on this queue, and then gets the message and processes the request.

1. In <ESB\_HOME>\repository\conf\axis2\axis2.xml, comment out both the myTopicConnectionFactory parameter and the SQProxyCF parameter. It should look as shown below.

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <!--parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter-->

 <!--parameter name="SQProxyCF" locked="false">
 <parameter
name="java.naming.factory.initial">com.sun.jndi.fscontext.RefFSContextFactory</pa
rameter>
 <parameter
name="java.naming.provider.url">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter-->

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter>
 </parameter>
 </transportReceiver>

```

2. Start the ESB with the [Sample 251](#) configuration using the following command.

wso2esb-samples.bat -sn 251

3. Log into the ESB server management console at: <https://localhost:9443/carbon/>.

4. Select **Service Bus -> Source view** and update the JMS URL as follows.

jms:/JMS\_QUEUE?transport.jms.ConnectionFactoryJNDIName=MQ\_JMS\_MANAGER&java.namin  
g.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory&java.naming.provide  
r.url=file:/C:/JNDI-Directory&transport.jms.DestinationType=queue&transport.jms.  
ConnectionFactoryType=queue &transport.jms.Destination=JMS\_QUEUE

5. Configure the proxy service as follows.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="https http jms" startOnLoad="true"
trace="disable">
 <target>
 <endpoint>
 <address
uri="jms:/JMS_QUEUE?transport.jms.ConnectionFactoryJNDIName=MQ_JMS_MANAGER&java.n
aming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory&java.naming.prov
ider.url=file:/C:/JNDI-Directory&transport.jms.DestinationType=queue&transport.jm
s.ConnectionFactoryType=queue&transport.jms.Destination=JMS_QUEUE"/>
 </endpoint>
 <inSequence>
 <property name="TRANSPORT_HEADERS" scope="axis2" action="remove" />
 <property name="OUT_ONLY" value="true" />
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 </proxy>
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default "fault" sequence"/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
 </sequence>
 <sequence name="main">
 <log/>
 <drop/>
 </sequence>
</definitions>

```

6. Comment out `myTopicConnectionFactory` and uncomment `SQProxyCF` in the `<ESB_HOME>\samples\axis2Server\repository\conf\axis2.xml` file as follows.

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <!--parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
</parameter-->

 <parameter name="SQProxyCF" locked="false">
 <parameter
name="java.naming.factory.initial">com.sun.jndi.fscontext.RefFSContextFactory</pa
rameter>
 <parameter
name="java.naming.provider.url">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
</parameter>

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter>
 </transportReceiver>

```

7. Start the axis2 Server with the following command.

axis2Server.bat

8. Add the following parameters to service.xml in <ESB\_HOME>\samples\axis2Server\repository\s
ervices\SimpleStockQuoteService.aar.

```

<parameter name="transport.jms.ConnectionFactory">SQProxyCF</parameter>
<parameter name="transport.jms.Destination">JMS_QUEUE</parameter>

```

9. Send the request from the JMS client, and the sample Axis2 server console will print a message as follows.

```

ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
-Dmode=placeorder -Dsymbol=MSFT

```

## Topic Scenario 1: Client to Topic to ESB

In this scenario, the JMS client places an order on the topic `ivtT`. The ESB listens to this topic, gets the message, and sends it to the back-end server to process the request.

1. In `<ESB_HOME>\repository\conf\axis2\axis2.xml`, uncomment the `myTopicConnectionFactory` parameter and comment out the `SQProxyCF` parameter. It should look as shown below.

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter>

 <!--parameter name="SQProxyCF" locked="false"-->
 <parameter
name="java.naming.factory.initial">com.sun.jndi.fscontext.RefFSContextFactory</pa
rameter>
 <parameter
name="java.naming.provider.url">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
</parameter-->

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter>
</transportReceiver>
```

2. Start the ESB with the **Sample 250** configuration by running the following command.  
`wso2esb-samples.bat -sn 250`
3. Log in to the server management console at: <https://localhost:9443/carbon/>.
4. Click **web services -> list -> StockQuoteProxy -> edit (Specific Configuration)**
5. Add a service parameter as follows and save it.  
`name = transport.jms.ConnectionFactory value = myTopicConnectionFactory`
6. Go to the `<ESB_HOME>/samples/axis2Client` directory and build it using the ant command.
7. Go to the `<ESB_HOME>/samples/axis2Client /src/samples/userguide` directory, open the `GenericJMSClient.java` source file, and make the following changes in the code.
  - a. Set the `jms_dest` property default value to `ivtT` (line 45)

- b. Set the `java.naming.provider.url` to `file:/C:/JNDI-Directory` (line 82)
- c. Set the `java.naming.factory.initial` to `com.sun.jndi.fscontext.RefFSContextFactory` (line 85)
- d. Set the lookup key to `MQ_JMS_MANAGER` (line 89)

8. Configure the proxy configuration so that it appears as follows.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="https http jms" startOnLoad="true"
trace="disable">
 <target>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 <inSequence>
 <property name="OUT_ONLY" value="true" />
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 <parameter name="transport.jms.ContentType">
 <rules>
 <jmsProperty>contentType</jmsProperty>
 <default>application/xml</default>
 </rules>
 </parameter>
 <parameter
name="transport.jms.ConnectionFactory">myTopicConnectionFactory</parameter>
 </proxy>
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default "fault" sequence"/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
 </sequence>
 <sequence name="main">
 <log/>
 <drop/>
 </sequence>
</definitions>

```

9. Comment out the non-default connection factories in the `<ESB_HOME>\samples\axis2Server\repository\conf\axis2.xml` file so that it looks as follows.

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <!--parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter-->

 <!--parameter name="SQProxyCF" locked="false">
 <parameter
name="java.naming.factory.initial">com.sun.jndi.fscontext.RefFSContextFactory</pa
rameter>
 <parameter
name="java.naming.provider.url">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter-->

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/D/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">bogusq</parameter>
 </parameter>
 </parameter>
 </transportReceiver>

```

10. Start the axis2 server with the following command.

`axis2Server.bat`

11. Send the request from the JMS client, and the sample Axis2 server console will print a message.

Topic Scenario 2: Client to ESB to Topic

In this scenario, the JMS client sends an order to the ESB, which places it on the topic `ivtT`. The back-end server listens on this topic, and then gets the message and processes the request.

1. In `<ESB_HOME>\repository\conf\axis2\axis2.xml`, comment out the non-default connection factories as follows:

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <!--parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 <parameter name="transport.jms.Destination">ivtT</parameter>
 </parameter-->

 <!--parameter name="SQProxyCF" locked="false">
 <parameter
name="java.naming.factory.initial">com.sun.jndi.fscontext.RefFSContextFactory</pa
rameter>
 <parameter
name="java.naming.provider.url">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter-->

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">bogusq</parameter>
 </parameter>
 </parameter>
 </transportReceiver>

```

2. Start the ESB with the [Sample 251](#) configuration by running the following command.  
wso2esb-samples.bat -sn 251
3. Log into the ESB server management console at <https://localhost:9443/carbon/>.
4. Click **Service Bus -> Source view**, and in the JMS URL, change transport.jms.DestinationType to topic.
5. Add the following parameters to service.xml in <ESB\_HOME>\samples\axis2Server\repository\serv
ices\SimpleStockQuoteService.aar.

```

<parameter
name="transport.jms.ConnectionFactory">myTopicConnectionFactory</parameter>
<parameter name="transport.jms.Destination">ivtT</parameter>

```

6. Configure the proxy service so that it appears as follows.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="https http jms" startOnLoad="true"
trace="disable">
 <target>
 <endpoint>
 <address
uri="jms:/ivtT?transport.jms.ConnectionFactoryJNDIName=MQ_JMS_MANAGER&java.naming.
.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory&java.naming.provider.
url=file:/C:/JNDI-Directory&transport.jms.DestinationType=topic&transport.jms.Con
nectionFactoryType=topic&transport.jms.Destination=ivtT"/>
 </endpoint>
 <inSequence>
 <property name="TRANSPORT_HEADERS" scope="axis2" action="remove" />
 <property name="OUT_ONLY" value="true" />
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 </proxy>
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default "fault" sequence"/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
 </sequence>
 <sequence name="main">
 <log/>
 <drop/>
 </sequence>
</definitions>

```

7. In <ESB\_HOME>\samples\axis2Server\repository\conf\axis2.xml, uncomment myTopicConnectionFactory and comment out SQProxyCF.

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/C:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 <parameter name="transport.jms.Destination">ivtT</parameter>
 </parameter>

 <!--parameter name="SQProxyCF" locked="false">
 <parameter
name="java.naming.factory.initial">com.sun.jndi.fscontext.RefFSContextFactory</pa
rameter>
 <parameter
name="java.naming.provider.url">file:/D:/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">JMS_QUEUE</parameter>
 </parameter-->

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.sun.jndi.fscontext.RefFSContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">file:/D/JNDI-Directory</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">MQ_JMS_MANAGER</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.Destination">bogusq</parameter>
 </parameter>
 </parameter>
 </transportReceiver>

```

8. Start the axis2 server with the following command.

`axis2Server.bat`

9. Send the request from the JMS client, and the sample Axis2 server console will print a message.

For implementation details of other JMS use cases, see [JMS UseCases](#).

#### **Configure with IBM WebSphere Application Server**

This page describes how to configure the WSO2 JMS transport with IBM® WebSphere® Application Server.

1. Set up IBM WebSphere Application Server according to the instructions provided by IBM.
2. Create a JMS queue (e.g., **samplequeue**) and a JMS connection factory (e.g., **QueueConnectionFactory**) as described in the topics under [Setting Up JMS in IBM WebSphere Application Server](#) in the IBM documentation.
3. Copy the following libraries from `<WEBSPHERE_HOME>/java/lib` directory to `<ESB_HOME>/repository/components/lib` directory.

- com.ibm.ws.runtime.jar
- com.ibm.ws.admin.client\_7.0.0.jar
- com.ibm.ws.sib.client.thin.jms\_7.0.0.jar
- com.ibm.ws.webservices.thinclient\_7.0.0.jar
- bootstrap.jar

4. Add the following entries to the <ESB\_HOME>/repository/conf/etc/launch.ini file.

```
javax.jms,\njavax.rmi.CORBA,\n
```

5. Enable the JMS Listener and Sender in <ESB\_HOME>/repository/conf/axis2/axis2.xml by un-commenting the following lines of code.

### JMS Listener

```
<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">\n <parameter name="myQueueConnectionFactory" locked="false">\n <parameter name="java.naming.factory.initial"\nlocked="false">com.ibm.websphere.naming.WsnInitialContextFactory</parameter>\n <parameter name="java.naming.provider.url"\nlocked="false">iiop://localhost:2809</parameter>\n <parameter name="transport.jms.ConnectionFactoryJNDIName"\nlocked="false">QueueConnectionFactory</parameter>\n <parameter name="transport.jms.ConnectionFactoryType"\nlocked="false">queue</parameter>\n <parameter name="transport.jms.Destination">samplequeue</parameter>\n </parameter>\n\n <parameter name="default" locked="false">\n <parameter name="java.naming.factory.initial"\nlocked="false">com.ibm.websphere.naming.WsnInitialContextFactory</parameter>\n <parameter name="java.naming.provider.url"\nlocked="false">iiop://localhost:2809</parameter>\n <parameter name="transport.jms.ConnectionFactoryJNDIName"\nlocked="false">QueueConnectionFactory</parameter>\n <parameter name="transport.jms.ConnectionFactoryType"\nlocked="false">queue</parameter>\n <parameter name="transport.jms.Destination">samplequeue</parameter>\n </parameter>\n </parameter>\n </parameter>\n </parameter>\n </parameter>\n </parameter>\n </parameter>\n </parameter>\n</transportReceiver>
```

### JMS Sender

```
<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender" />
```

For details on the JMS configuration parameters used in the code segments above, see [JMS Connection Factory Parameters](#).

## 6. Start IBM WebSphere Application Server and WSO2 ESB.

You now have instances of IBM WebSphere Application Server and WSO2 ESB configured and running. Next, see [JMS Usecases](#) for implementation details of various JMS use cases.

### **Configure with JBossMQ**

The following instructions describe how to set up the [JMS transport with JBossMQ](#), the default JMS provider in JBoss Application Server 4.2. (JBossMQ was replaced by [JBoss Messaging](#) in JBoss Application Server 5.0.)

To configure the JMS transport with JBossMQ:

1. Copy the following client libraries to the <ESB\_HOME>/repository/components/lib directory.
  - <JBoss\_HOME>/lib/jboss-system.jar
  - <JBoss\_HOME>/client/jbossall-client.jar
2. Enable the JMS transport listener by adding the following listener configuration to the <ESB\_HOME>/repository/conf/axis2/axis2.xml file:

```
<!-- Configuration for JBoss 4.2.2 GA MQ -->
<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="MyQueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.jnp.interfaces.NamingContextFactory</parameter>
 <parameter name="java.naming.factory.url.pkgs"
locked="false">org.jnp.interfaces:org.jboss.naming</parameter>
 <parameter name="java.naming.provider.url"
locked="false">jnp://localhost:1099</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">/ConnectionFactory</parameter>
 <parameter name="transport.jms.Destination"
locked="true">queue/susaQueue</parameter>
 </parameter>
</transportReceiver>
```

3. Enable the JMS transport sender by uncommenting the following line in the <ESB\_HOME>/repository/conf/axis2/axis2.xml file:

```
<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender" />
```

4. Start the ESB and ensure that the logs prints messages indicating that the JMS listener and sender are started and that the JMS transport is initialized.

### **Configure with MSMQ**

This section describes how to configure the WSO2 ESB's JMS transport with Microsoft Message Queuing (MSMQ).

The setup instruction here are only applicable for Windows environments since we invoke Microsoft C++ API for MSMQ via JNI invocations.

The **msmq**: component in WSO2 ESB is a transport for working with MSMQ. This component natively sends and receives directly allocated ByteBuffer instances, allowing access to the JNI layer without memory copying. Using the **ByteBuffer** created with the method **allocateDirect**, the native code can directly access the memory. URI format is `msmq:msmqQueueName`.

Follow the steps below to set up and configure WSO2 ESB with MSMQ.

1. Download axis2-transport-msmq-1.1.0-wso2v6.jar for [64x](#) or [32x](#), and place that in <ESB\_HOME>/repository/components/dropins directory. MSMQ bridging requires JNI invocation, and WSO2 ships two dlls for 64bit and 32bit O/S respectively. Therefore, make sure you download the correct file suitable for your environment.
2. Install Visual C++ 2008 (VC9). It works with Microsoft Visual Studio 2008 Express.
3. Set up MSMQ on a Windows environment. For setup instructions, refer to : <http://msdn.microsoft.com/en-us/library/aa967729.aspx>.
4. If you haven't already, download and install WSO2 ESB as described in [Getting Started](#) .  
Setting up the JMS Listener
5. Add the following configuration to <ESB\_HOME>/repository/conf/axis2/axis2.xml file.

```
<transportReceiver name="msmq" class="org.apache.axis2.transport.msmq.MSMQListener">
 <parameter name="msmq.receiver.host" locked="false">localhost</parameter>
</transportReceiver>
```

#### Setting up the JMS Sender

6. To enable the JMS transport sender, add the following JMS transport listener configuration in <ESB\_HOME>/repository/conf/axis2/axis2.xml file.

```
<transportSender name="msmq" class="org.apache.axis2.transport.msmq.MSMQSender" />
```

For details on the JMS configuration parameters used in the code segments above, see [JMS Connection Factory Parameters](#).

You now have instances of MSMQ and WSO2 ESB configured, up and running. Next, refer to section [JMS Usecases](#) for implementation details of various JMS use cases.

#### **Configure with Tibco EMS**

This section describes how to configure the WSO2 ESB's JMS transport with Tibco EMS . Follow the steps below to set up and configure.

1. Download and set up Tibco EMS in your environment.
2. If you have not done so already, download and install WSO2 ESB as described in [Installation Guide](#).
3. Copy the Tibco EMS client jar files that are shipped with the distribution to the <ESB\_HOME>/repository/components/extensions directory.
  - tibcrypt.jar
  - tibjms.jar
  - tibjmsadmin.jar
  - tibjmsapps.jar
  - tibrvjms.jar
4. WSO2 ESB does not have a default configuration script for TIBCO EMS. Therefore, add the following configuration to the <ESB\_HOME>/repository/conf/axis2/axis2.xml file.

#### Setting up the JMS Listener

```
<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="TopicConnectionFactory" locked="false">
 <parameter locked="false" name="java.naming.factory.initial">
 com.tibco.tibjms.naming.TibjmsInitialContextFactory
 </parameter>
 </parameter>
```

```

<parameter locked="false"
name="java.naming.provider.url">tcp://127.0.0.1:37222</parameter>
 <parameter locked="false"
name="java.naming.security.principal">admin</parameter>
 <parameter locked="false" name="java.naming.security.credentials"/>
 <parameter locked="false"
name="transport.jms.ConnectionFactoryJNDIName">TopicConnectionFactory</parameter>
 <parameter locked="false"
name="transport.jms.JMSSpecVersion">1.0.2b</parameter>
 <parameter locked="false"
name="transport.jms.ConnectionFactoryType">topic</parameter>
 <parameter locked="false" name="transport.jms.UserName">admin</parameter>
 <parameter locked="false" name="transport.jms.Password">admin</parameter>
 <parameter locked="false" name="transport.jms.CacheLevel">session</parameter>
 </parameter>
 <parameter locked="false" name="QueueConnectionFactory">
 <parameter locked="false" name="java.naming.factory.initial">
 com.tibco.tibjms.naming.TibjmsInitialContextFactory
 </parameter>
 <parameter locked="false"
name="java.naming.provider.url">tcp://127.0.0.1:37222</parameter>
 <parameter locked="false"
name="java.naming.security.principal">admin</parameter>
 <parameter locked="false" name="java.naming.security.credentials"/>
 <parameter locked="false"
name="transport.jms.ConnectionFactoryJNDIName">QueueConnectionFactory</parameter>
 <parameter locked="false"
name="transport.jms.JMSSpecVersion">1.0.2b</parameter>
 <parameter locked="false"
name="transport.jms.ConnectionFactoryType">queue</parameter>
 <parameter locked="false" name="transport.jms.UserName">admin</parameter>
 <parameter locked="false" name="transport.jms.Password">admin</parameter>
 <parameter locked="false" name="transport.jms.CacheLevel">session</parameter>
 </parameter>
 <parameter name="default" locked="false">
 <parameter locked="false" name="java.naming.factory.initial">
 com.tibco.tibjms.naming.TibjmsInitialContextFactory
 </parameter>
 <parameter locked="false"
name="java.naming.provider.url">tcp://127.0.0.1:37222</parameter>
 <parameter locked="false"
name="java.naming.security.principal">admin</parameter>
 <parameter locked="false" name="java.naming.security.credentials"/>
 <parameter locked="false"
name="transport.jms.ConnectionFactoryJNDIName">QueueConnectionFactory</parameter>
 <parameter locked="false"
name="transport.jms.JMSSpecVersion">1.0.2b</parameter>
 <parameter locked="false"
name="transport.jms.ConnectionFactoryType">queue</parameter>
 <parameter locked="false" name="transport.jms.UserName">admin</parameter>
 <parameter locked="false" name="transport.jms.Password">admin</parameter>

```

```

<parameter locked="false" name="transport.jms.CacheLevel">session</parameter>
</parameter>
</transportReceiver>

```

#### Setting up the JMS Sender

To enable the JMS transport sender, add the following JMS transport listener configuration in <ESB\_HOME>/repository/conf/axis2/axis2.xml file.

```

<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender">
 <parameter locked="false" name="QueueConnectionFactory">
 <parameter locked="false" name="java.naming.factory.initial">
 com.tibco.tibjms.naming.TibjmsInitialContextFactory
 </parameter>
 <parameter locked="false" name="java.naming.provider.url">tcp://127.0.0.1:37222</parameter>
 <parameter locked="false" name="transport.jms.ConnectionFactoryJNDIName">QueueConnectionFactory</parameter>
 <parameter locked="false" name="transport.jms.JMSSpecVersion">1.0.2b</parameter>
 <parameter locked="false" name="transport.jms.ConnectionFactoryType">queue</parameter>
 </parameter>
</transportSender>

```

For details on the JMS configuration parameters used in the code segments above, see [JMS Connection Factory Parameters](#).

You now have instances of Tibco EMS and WSO2 ESB configured, up and running. Next, see [JMS UseCases](#) for implementation details of various JMS use cases.

#### Configure with SwiftMQ

This section describes how to configure the WSO2 ESB's JMS transport with SwiftMQ. Follow the instructions below to set up and configure.

1. Download and set up SwiftMQ. Instructions can be found in SwiftMQ documentation.
2. If you have not already done so, download and install WSO2 ESB as described in [Installation Guide](#).

SwiftMQ should be up and running before starting the ESB.

3. Copy the following client libraries from <SMQ\_HOME>/lib directory to <ESB\_HOME>/repository/components/lib directory.

- jms.jar
- jndi.jar
- swiftmq.jar

Always use the standard client libraries that come with a particular version of SwiftMQ, in order to avoid version incompatibility issues. We recommend you to remove old client libraries, if any, from all locations including <ESB\_HOME>/repository/components/lib and <ESB\_HOME>/repository/components/droppings before copying the ones relevant to a given version.

4. Next, configure transport listeners and senders in ESB.

Setting up the JMS Listener

To enable the JMS transport listener, add the following listener configuration related to SwiftMQ to <ESB\_HOME>/repository/conf/axis2/axis2.xml file.

```
<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.swiftmq.jndi.InitialContextFactoryImpl</parameter>
 <parameter name="java.naming.provider.url"
locked="false">smqp://localhost:4001/timeout=10000</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">TopicConnectionFactory</parameter>
 <parameter name="transport.jms.JMSSpecVersion" locked="false">1.0</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 </parameter>
 <parameter name="myQueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.swiftmq.jndi.InitialContextFactoryImpl</parameter>
 <parameter name="java.naming.provider.url"
locked="false">smqp://localhost:4001/timeout=10000</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.JMSSpecVersion" locked="false">1.0</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>
 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">com.swiftmq.jndi.InitialContextFactoryImpl</parameter>
 <parameter name="java.naming.provider.url"
locked="false">smqp://localhost:4001/timeout=10000</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.JMSSpecVersion" locked="false">1.0</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>
</transportReceiver>
```

#### Setting up the JMS Sender

To enable the JMS transport sender, add the following configuration in <ESB\_HOME>/repository/conf/axis2/axis2.xml file.

```
<!-- uncomment this and configure to use connection pools for sending messages -->
<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender" />
```

For details on the JMS configuration parameters used in the code segments above, see [JMS Connection Factory Parameters](#).

You now have instances of SwiftMQ and WSO2 ESB configured, up and running. Next, refer to section [JMS Usecases](#) for implementation details of various JMS use cases.

#### **Configure with WebLogic**

The following instructions describe how to configure the JMS transport with Oracle WebLogic 10.3.4.0.  
Starting WebLogic and the ESB

1. Download, set up, and start [Oracle WebLogic Server](#).
2. [Start WSO2 ESB](#).
3. Wrap the weblogic client jar and build a new OSGi bundle using the following `pom.xml`. The exporting of `javax.jms` package and `javax.xml.namespace` package of the client jar should be prevented.
4. Copy the client libraries file `wlfullclient.jar` from the `<WEBLOGIC_HOME>/wlserver_XX/server/lib` directory to the `<ESB_HOME>/repository/components/dropins` directory.

Configuring WebLogic server

Configure the required connection factories and queues in WebLogic. An entry for a JMS queue would look like the following. The configuration files can be found in configuration inside `<WEBLOGIC_HOME>/user_projects/domains/<DOMAIN_NAME>/config/jms` file. Alternatively you can configure using the WebLogic web console which can be accessed through <http://localhost:7001> with default configurations.

```
<queue name="wso2MessageQueue">
 <sub-deployment-name>jms</sub-deployment-name>
 <jndi-name>jms/wso2MessageQueue</jndi-name>
</queue>
```

Once you start the WebLogic server with the above changes, you'll be able to see the following on STDOUT.

```
<Jun 25, 2013 11:20:02 AM IST> <Notice> <WebLogicServer> <BEA-000331> <Started
WebLogic Admin Server "AdminServer" for domain "wso2" running in Development Mode>
<Jun 25, 2013 11:20:02 AM IST> <Notice> <WebLogicServer> <BEA-000365> <Server state
changed to RUNNING>
<Jun 25, 2013 11:20:02 AM IST> <Notice> <WebLogicServer> <BEA-000360> <Server started
in RUNNING mode>
```

You will now need to configure the transport listener, sender, and message store in WSO2 ESB. For details on JMS configuration parameters used in the code segments, see [JMS Connection Factory Parameters](#).

Setting up the JMS listener

To enable the JMS transport listener, add the following listener configuration related to Weblogic in the `<ESB_HOME>/repository/conf/axis2/axis2.xml` file.

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myQueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">weblogic.jndi.WLInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">t3://localhost:7001</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">jms/myconnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.UserName" locked="false">weblogic</parameter>
 <parameter name="transport.jms.Password" locked="false">admin123</parameter>
 </parameter>
 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">weblogic.jndi.WLInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">t3://localhost:7001</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">jms/myConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.UserName" locked="false">weblogic</parameter>
 <parameter name="transport.jms.Password" locked="false">admin123</parameter>
 </parameter>
</transportReceiver>

```

#### Setting up the JMS sender

To enable the JMS transport sender, un-comment the following configuration in the `<ESB_HOME>/repository/conf/axis2/axis2.xml` file.

```
<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender" />
```

#### Setting up a message store in ESB

To set up a message store for WebLogic messages in the ESB, use a configuration similar to the following:

```

<messageStore class="org.wso2.carbon.message.store.persistence.jms.JMSMessageStore"
 name="wso2MessageStore">
 <parameter
name="java.naming.factory.initial">weblogic.jndi.WLInitialContextFactory</parameter>
 <parameter name="store.jms.cache.connection">false</parameter>
 <parameter name="store.jms.password">admin123</parameter>
 <parameter name="java.naming.provider.url">t3://localhost:7001</parameter>
 <parameter name="store.jms.ConsumerReceiveTimeOut">300</parameter>
 <parameter name="store.jms.connection.factory">jms/myConnectionFactory</parameter>
 <parameter name="store.jms.username">weblogic</parameter>
 <parameter name="store.jms.JMSSpecVersion">1.1</parameter>
 <parameter name="store.jms.destination">jms/wso2MessageQueue</parameter>
</messageStore>

```

#### JMS Producer Proxy Service

Use the following proxy service configuration in ESB to publish messages to the WebLogic queue:

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="WeblogicJMSSenderProxy"
 transports="http"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
 <target>
 <inSequence>
 <property name="Accept-Encoding" scope="transport" action="remove"/>
 <property name="Content-Length" scope="transport" action="remove"/>
 <property name="Content-Type" scope="transport" action="remove"/>
 <property name="User-Agent" scope="transport" action="remove"/>
 <log level="custom">
 <property name="STATUS:>
 value="-----Message send by WeblogicJMSSConsumerProxy-----"/>
 </log>
 <property name="OUT_ONLY" value="true"/>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
 <send>
 <endpoint>
 <address
uri="jms:/jms/TestJMSQueue1?transport.jms.ConnectionFactoryJNDIName=jms/TestConnectionFactory1&java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory&java.naming.provider.url=t3://localhost:7001&transport.jms.DestinationType=queue"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <description/>
 </proxy>

```

#### JMS Consumer Proxy Service

Use the following proxy service configuration in ESB to read messages from the WebLogic queue:

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="WeblogicJMSConsumerProxy"
 transports="jms"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
 <target>
 <inSequence>
 <log level="custom">
 <property name="STATUS:>
 value="-----Message consumed by
WeblogicJMSConsumerProxy-----"/>
 </log>
 <log level="full"/>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <parameter name="transport.jms.Destination">jms/TestJMSQueue1</parameter>
 <description/>
 </proxy>

```

### Configure with HornetQ

This section describes how to configure WSO2 ESB's [JMS transport](#) with HornetQ, which is an open source project to build a multi-protocol, asynchronous messaging system.

When configuring WSO2 ESB's [JMS transport](#) with HornetQ, you can either configure with a standalone HornetQ server or with HornetQ embedded in a JBoss Enterprise Application Platform (JBoss EAP) server.

Go to the required tab for step by step instructions based on how you need to configure the ESB's [JMS transport](#) with HornetQ.

[Configure with a standalone HornetQ server](#)  
[Configure with HornetQ embedded in a JBoss EAP server](#)

Follow the instructions below to configure WSO2 ESB's JMS transport with a standalone HornetQ server.

1. Download HornetQ from the [HornetQ Downloads](#) site.
2. Create a sample queue by editing the `<HORNET_HOME>/config/stand-alone/non-clustered/hornetq-jms.xml` file as follows:

```

<queue name="wso2">
 <entry name="/queue/mySampleQueue" />
</queue>

```

3. Add the following two connection entries to the same file. These entries are required to enable the ESB to act as a JMS consumer.

```

<connection-factory name="QueueConnectionFactory">
 <xa>false</xa>
 <connectors>
 <connector-ref connector-name="netty" />
 </connectors>
 <entries>
 <entry name="/QueueConnectionFactory" />
 </entries>
</connection-factory>

<connection-factory name="TopicConnectionFactory">
 <xa>false</xa>
 <connectors>
 <connector-ref connector-name="netty" />
 </connectors>
 <entries>
 <entry name="/TopicConnectionFactory" />
 </entries>
</connection-factory>

```

4. If you have not already done so, download WSO2 ESB and install as described in the [Installation Guide](#).
5. Copy the hornet-all.jar into the <ESB\_HOME>/repository/components/lib folder. This jar can be downloaded from [here](#).

### Note

If you are packing the JARs yourself, make sure you remove the javax.jms package from the assembled JAR to avoid the carbon runtime from picking this implementation of JMS over the bundled-in distribution.

6. Uncomment the following line in the <ESB\_HOME>/repository/conf/axis2/axis2.xml file to enable the JMS transport sender on axis2core.

```
<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender" />
```

7. Enable the JMS listener with the HornetQ configuration parameters in the <ESB\_HOME>/repository/conf/axis2/axis2.xml file by un-commenting the following lines of code.

```

<transportReceiver name="jms"
 class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory" locked="false">
 <parameter
 name="java.naming.factory.initial" locked="false">org.jnp.interfaces.NamingContextFactory</parameter>
 <parameter
 name="java.naming.factory.url.pkgs" locked="false">org.jboss.naming:org.jnp.interfaces</parameter>
 <parameter name="java.naming.provider.url"
 locked="false">jnp://localhost:1099</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
 locked="false">TopicConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
 locked="false">topic</parameter>
 </parameter>
 <parameter name="myQueueConnectionFactory" locked="false">
 <parameter
 name="java.naming.factory.initial" locked="false">org.jnp.interfaces.NamingContextFactory</parameter>
 <parameter
 name="java.naming.factory.url.pkgs" locked="false">org.jboss.naming:org.jnp.interfaces</parameter>
 <parameter name="java.naming.provider.url"
 locked="false">jnp://localhost:1099</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
 locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
 locked="false">queue</parameter>
 </parameter>
 <parameter name="default" locked="false">
 <parameter
 name="java.naming.factory.initial" locked="false">org.jnp.interfaces.NamingContextFactory</parameter>
 <parameter
 name="java.naming.factory.url.pkgs" locked="false">org.jboss.naming:org.jnp.interfaces</parameter>
 <parameter name="java.naming.provider.url"
 locked="false">jnp://localhost:1099</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
 locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
 locked="false">queue</parameter>
 </parameter>
</transportReceiver>

```

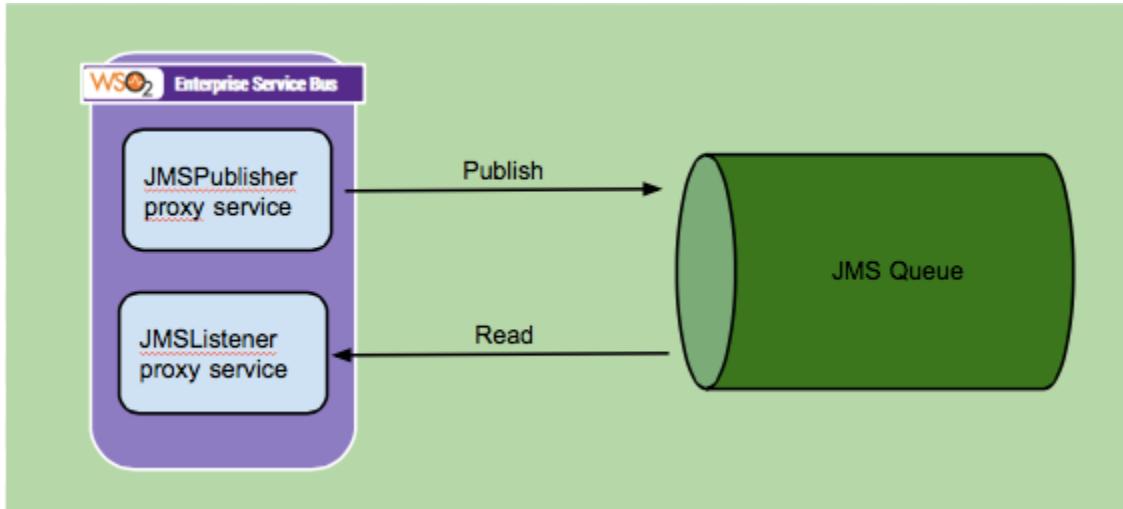
## 8. Start HornetQ with the following command.

- On Windows: <HORNETQ\_HOME>\bin\run.bat --run
- On Linux/Solaris: sh <HORNETQ\_HOME>/bin/run.sh

Now you have configured the ESB's JMS transport with a standalone HornetQ server. The next section describes how you can test the configuration.

### Testing the configuration

To test the configuration we create a proxy service named `JMSPublisher` to publish messages from the ESB to the HornetQ sample queue, and create the `JMSListener` queue to read messages from the HornetQ sample queue.



1. Start the ESB management console. See [Running the Product](#) for more information.
2. Create the JMSPublisher proxy service with the following configuration:

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="JMSPublisher"
 transports="https,http"
 statistics="enable"
 trace="enable"
 startOnLoad="true">
 <target>
 <inSequence>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
 <property name="Accept-Encoding" scope="transport" action="remove"/>
 <property name="Content-Length" scope="transport" action="remove"/>
 <property name="Content-Type" scope="transport" action="remove"/>
 <property name="User-Agent" scope="transport" action="remove"/>
 <property name="OUT_ONLY" value="true"/>
 <log level="full"/>
 <send>
 <endpoint>
 <address
uri="jms:/queue/mySampleQueue?transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory&java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory&java.naming.provider.url=jnp://localhost:1099&transport.jms.DestinationType=queue"/>
 </endpoint>
 </send>
 </inSequence>
 </target>
 <publishWSDL
 uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 <description>HornetQ-WSO2 ESB sample</description>
 </proxy>

```

## Note

- The proxy service is created for the `SimpleStockQuoteService` service shipped with the ESB samples in this example so that the configuration can be tested by sending a request to call one of the operations of the service.
- The `OUT_ONLY` parameter is set to `true` since this proxy service is created only for the purpose of publishing the messages of the ESB to the `mySampleQueue` queue specified in the address URI.
- You may have to change the host name, port etc. of the JMS string based on your environment

3. Create the `JMSListener` proxy service with the following configuration:

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="JMSListener"
 transports="jms"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
 <target>
 <inSequence>
 <log level="custom">
 <property name="JMS_LISTENER_PROXY" value="LOCATED" />
 </log>
 <log level="full"/>
 <drop/>
 </inSequence>
 </target>
 <parameter name="transport.jms.ContentType">
 <rules>
 <jmsProperty>contentType</jmsProperty>
 <default>application/xml</default>
 </rules>
 </parameter>
 <parameter name="transport.jms.Destination">queue/mySampleQueue</parameter>
 <description/>
</proxy>
```

4. Use a client application of your choice to send a request to the endpoint of the `JMSPublisher` proxy service. In this example, the following request is sent using SOAPUI, calling the `placeOrder` operation of the `SimpleStockQuoteService` for which the `JMSPublisher` proxy service is created.

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
 <soapenv:Header/>
 <soapenv:Body>
 <ser:placeOrder>
 <!--Optional:-->
 <ser:order>
 <!--Optional:-->
 <xsd:price>20</xsd:price>
 <!--Optional:-->
 <xsd:quantity>20</xsd:quantity>
 <!--Optional:-->
 <xsd:symbol>IBM</xsd:symbol>
 </ser:order>
 </ser:placeOrder>
 </soapenv:Body>
</soapenv:Envelope>

```

- Check the log on your ESB terminal. You will see the following log, which indicates that the request published in the queue is picked by the JMSListener proxy.

```

[<TIME Stamp>] INFO - LogMediator JMS LISTENER PROXY = LOCATED

[<TIME Stamp>] INFO - LogMediator To: , WSAction: "urn:placeOrder", SOAPAction: "urn:placeOrder", MessageID: ID:be08707e-033d-11e4-8307-25263fbb9173, Direction: request, Envelope: <?xml version="1.0" encoding="utf-8"?><soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Body><soapenv:Envelope xmlns:xsd="http://services.samples/xsd" xmlns:ser="http://services.samples"><soapenv:Body>

<ser:placeOrder>
 <!--Optional:-->
 <ser:order>
 <!--Optional:-->
 <xsd:price>20</xsd:price>
 <!--Optional:-->
 <xsd:quantity>20</xsd:quantity>
 <!--Optional:-->
 <xsd:symbol>IBM</xsd:symbol>
 </ser:order>
</ser:placeOrder>
</soapenv:Body></soapenv:Envelope></soapenv:Body></soapenv:Envelope>

```

Follow the instructions below to set up and configure WSO2 ESB with HornetQ embedded in a JBoss EAP server.

### Setting up JBoss EAP

This section describes the steps to install JBoss EAP server and create a message queue within the server.

- Download JBoss EAP Server 7.0.0 from [JBoss EAP Downloads](#) and run the JBoss EAP installer as described [here](#).
- Execute one of the following commands in command prompt to create a new application user.
  - On Windows: <EAP\_HOME>\bin\add-user.bat -a -u 'SampleUser' -p 'SamplePwd1!'

- g 'guest'
- On Linux/Mac: <EAP\_HOME>/bin/add-user.sh -a -u 'SampleUser' -p 'SamplePwd1!' -g 'guest'

3. Create a sample queue by editing the <EAP\_HOME>/standalone/configuration/standalone-full.xml file. Add the following content within the <hornetq-server> element:

```
<jms-destinations>
 <jms-queue name="sampleQueue">
 <entry name="queue/test"/>
 <entry name="java:jboss/exported/jms/queue/test"/>
 </jms-queue>
</jms-destinations>
```

4. Start the JBoss EAP server by executing one of the following commands in command prompt:

- On Windows: <EAP\_HOME>\bin\standalone.bat -c standalone-full.xml
- On Linux/Mac: <EAP\_HOME>/bin/standalone.sh -c standalone-full.xml

5. Access the management console of the JBoss EAP server using the following URL:

<http://127.0.0.1:9990>

6. Log in to the Management Console using **admin** as both the username and password. In the Profile menu, click Messaging -> Destinations and you will be able to see the queue you added in Step 4 in the **Queues/Topics** section.

The screenshot shows the JBoss EAP 6.2.0.GA management console interface. The top navigation bar includes 'RED HAT JBOSS ENTERPRISE APPLICATION PLATFORM 6.2.0.GA', 'Messages: 0', and a user icon for 'admin'. Below the navigation is a main menu with tabs: 'Profile', 'Runtime', and 'Administration'. Under 'Administration', the 'Subsystems' tree view is expanded, showing 'Connector', 'Container', 'Core', 'Infinispan', 'Messaging' (which is selected), 'Connections', 'Clustering', and 'Destinations' (also selected). The 'Destinations' node has children: 'Security' (with 'Security Subsystem' and 'Security Domains'), 'Web', and 'General Configuration'. The central content area is titled 'MESSAGING DESTINATIONS' and shows the 'Queues/Topics' tab is active. It displays 'JMS Endpoints: Provider default' and 'Queue and Topic destinations.' A table lists existing queues, with 'sampleQueue' selected. An 'Edit' dialog for 'sampleQueue' is open, showing details: Name: sampleQueue, JNDI Names: queue/test java:jboss/exported/jms/queue/test, Durable?: true, and Selector: (empty). Buttons for 'Add' and 'Remove' are also visible.

Now you have configured the JBoss EAP Server. The next section describes how to configure WSO2 ESB to listen and fetch messages from the queue that you created above.

## Configuring WSO2 ESB

1. If you have not already done so, download and [install WSO2 ESB](#).
2. Enable the JMS listener with the JBoss EAP configuration parameters in the <ESB\_HOME>/repository/conf/axis2/axis2.xml file by adding the following lines of code.

```
<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="QueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.jboss.naming.remote.client.InitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">remote://localhost:4447</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">jms/RemoteConnectionFactory</parameter>
 <parameter name="transport.jms.UserName"
locked="false">SampleUser</parameter>
 <parameter name="transport.jms.Password"
locked="false">SamplePwd1!</parameter>
 <parameter name="java.naming.security.principal"
locked="false">SampleUser</parameter>
 <parameter name="java.naming.security.credentials"
locked="false">SamplePwd1!</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>
</transportReceiver>
```

The username and password created for the guest user in the above section are used in the configuration.

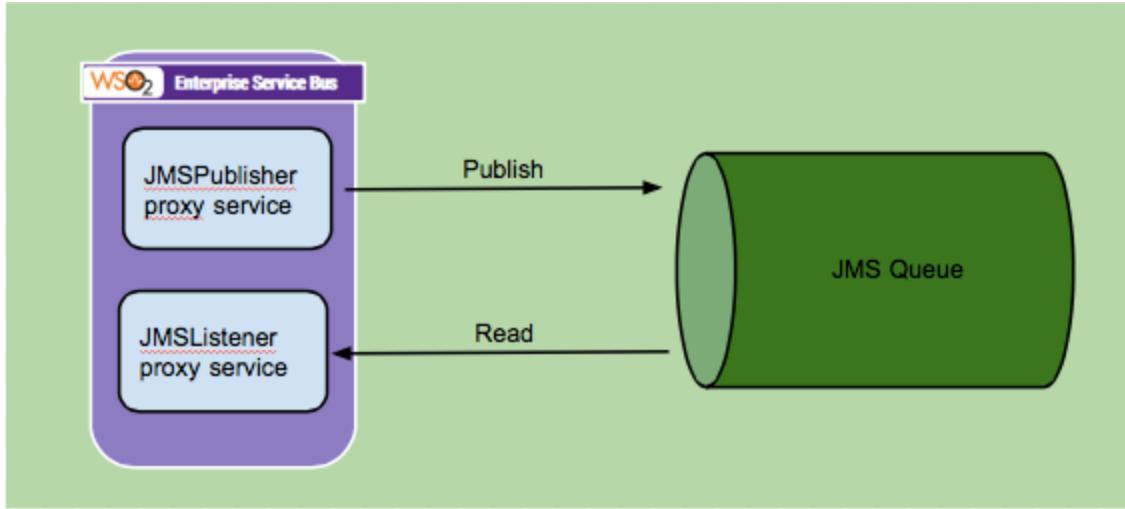
3. Enable the JMS sender by uncommenting the following line of code in the <ESB\_HOME>/repository/conf/axis2/axis2.xml file.

```
<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender" />
```

Now you have configured the ESB's JMS transport with HornetQ embedded in a JBoss EAP server. The next section describes how you can test the configuration.

### Testing the configuration

To test the configuration we create a proxy service named `JMSPublisher` to publish messages from the ESB to the HornetQ sample queue, and create the `JMSListener` queue to read messages from the HornetQ sample queue.



1. Start the ESB management console. See [Running the Product](#) for more information.
2. Create the JMSPublisher proxy service with the following configuration:

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="JMSPublisher"
 transports="https,http"
 statistics="enable"
 trace="enable"
 startOnLoad="true">
 <target>
 <inSequence>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
 <property name="Accept-Encoding" scope="transport" action="remove"/>
 <property name="Content-Length" scope="transport" action="remove"/>
 <property name="Content-Type" scope="transport" action="remove"/>
 <property name="User-Agent" scope="transport" action="remove"/>
 <property name="OUT_ONLY" value="true"/>
 <log level="full"/>
 <send>
 <endpoint>
 <address
uri="jms:/jms/queue/test?transport.jms.ConnectionFactoryJNDIName=jms/RemoteConnectionFactory&java.naming.factory.initial=org.jboss.naming.remote.client.InitialContextFactory&java.naming.provider.url=remote://localhost:4447&transport.jms.DestinationType=queue&transport.jms.UserName=SampleUser&transport.jms.Password=SamplePwd1!&java.naming.security.principal=SampleUser&java.naming.security.credentials=SamplePwd1!"> </endpoint>
 </send>
 </inSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 <description>HornetQ-WSO2 ESB sample</description>
 </proxy>

```

**Note**

- The proxy service is created for the `SimpleStockQuoteService` service shipped with the ESB samples in this example so that the configuration can be tested by sending a request to call one of the operations of the service.
- The `OUT_ONLY` parameter is set to `true` since this proxy service is created only for the purpose of publishing the messages of the ESB to the `mySampleQueue` queue specified in the address URI.
- You may have to change the host name, port etc. of the JMS string based on your environment

3. Create the `JMSListener` proxy service with the following configuration:

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="JMSListener"
 transports="jms"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
 <target>
 <inSequence>
 <log level="custom">
 <property name="JMS_LISTENER_PROXY" value="LOCATED" />
 </log>
 <log level="full" />
 <drop/>
 </inSequence>
 </target>
 <parameter name="transport.jms.ContentType">
 <rules>
 <jmsProperty>contentType</jmsProperty>
 <default>text/plain</default>
 </rules>
 </parameter>
 <parameter
 name="transport.jms.ConnectionFactory">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.Destination">jms/queue/test</parameter>
 <description/>
</proxy>

```

4. Use a client application of your choice to send a request to the endpoint of the `JMSPublisher` proxy service.

In this example, the following request is sent using SOAPUI, calling the `placeOrder` operation of the `SimpleStockQuoteService` for which the `JMSPublisher` proxy service is created.

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
 <soapenv:Header/>
 <soapenv:Body>
 <ser:placeOrder>
 <!--Optional:-->
 <ser:order>
 <!--Optional:-->
 <xsd:price>20</xsd:price>
 <!--Optional:-->
 <xsd:quantity>20</xsd:quantity>
 <!--Optional:-->
 <xsd:symbol>IBM</xsd:symbol>
 </ser:order>
 </ser:placeOrder>
 </soapenv:Body>
</soapenv:Envelope>

```

5. Check the log on your ESB terminal. You will see the following log, which indicates that the request published in the queue is picked by the JMSListener proxy.

```

[<TIME Stamp>] INFO - LogMediator JMS LISTENER PROXY = LOCATED

[<TIME Stamp>] INFO - LogMediator To: , WSAction: "urn:placeOrder", SOAPAction: "urn:placeOrder", MessageID: ID:be08707e-033d-11e4-8307-25263fbb9173, Direction: request, Envelope: <?xml version="1.0" encoding="utf-8"?><soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Body><soapenv:Envelope xmlns:xsd="http://services.samples/xsd" xmlns:ser="http://services.samples"><soapenv:Body>

<ser:placeOrder>
 <!--Optional:-->
 <ser:order>
 <!--Optional:-->
 <xsd:price>20</xsd:price>
 <!--Optional:-->
 <xsd:quantity>20</xsd:quantity>
 <!--Optional:-->
 <xsd:symbol>IBM</xsd:symbol>
 </ser:order>
</ser:placeOrder>
</soapenv:Body></soapenv:Envelope></soapenv:Body></soapenv:Envelope>

```

### **Configure with Multiple Brokers**

If your system has more than one existing brokers, it will be required to configure the JMS transport with multiple brokers. In such situations, each transport receiver should have a separate name.

The following example illustrates how to configure the WSO2 ESB to listen to both ActiveMQ and WSO2 MB messages.

1. Download ActiveMQ (version 5.8.0 or later) from the [Apache ActiveMQ](#) site. Download the WSO2 Message Broker from the [WSO2 Message Broker](#) site.
2. Copy the following client libraries from <AMQ\_HOME>/lib directory to <ESB\_HOME>/repository/components

- nts/lib directory.
- activemq-broker-5.8.0.jar
  - activemq-client-5.8.0.jar
  - geronimo-jms\_1.1\_spec-1.1.1.jar
  - geronimo-j2ee-management\_1.1\_spec-1.0.1.jar
  - hawtbuf-1.9.jar
3. Copy the andes-client-0.13.wso2v10.jar from <MB\_HOME>/client-lib directory to <ESB\_HOME>/repository/components/lib directory.
  4. Configure the <ESB\_HOME>/repository/conf/axis2/axis2.xml file as follows to enable ActiveMQ as a transport listener.

```

<transportReceiver name="jms1"
class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">TopicConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 </parameter>

 <parameter name="myQueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>
</transportReceiver>
```

Enter the following configuration in the same file to enable the WSO2 Message Broker.

```

<!--Uncomment this and configure as appropriate for JMS transport support with
WSO2 MB 2.x.x -->
<transportReceiver name="jms2"
class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 >
 <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">TopicConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 </parameter>

 <parameter name="myQueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 >
 <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 >
 <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>
</transportReceiver>

```

Note that the transport receiver name is different in each configuration.

5. Add a topic and a queue to the `jndi.properties` file located in `<ESB_HOME>/repository/conf` as follows:

```

<connection-factory name="QueueConnectionFactory">
 <xa>false</xa>
 <connectors>
 <connector-ref connector-name="netty" />
 </connectors>
 <entries>
 <entry name="/QueueConnectionFactory" />
 </entries>
</connection-factory>

<connection-factory name="TopicConnectionFactory">
 <xa>false</xa>
 <connectors>
 <connector-ref connector-name="netty" />
 </connectors>
 <entries>
 <entry name="/TopicConnectionFactory" />
 </entries>
</connection-factory>

```

6. Start both ActiveMQ and WSO2 MB.
7. Start the WSO2 ESB server and the management console. If they are already running, restart them.

Now a proxy service can be created with reference to transport receiver JMS1 and/or JMS2. For example, the following proxy service is configured with reference to both transport receivers.

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
name="jmsMultipleListnerProxy"
transports="jms1,jms2"
statistics="disable"
trace="disable"
startOnLoad="true">
<target>
<inSequence>
<log level="full"/>
<send>
<endpoint>
<address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
</endpoint>
</send>
</inSequence>
<outSequence>
<send/>
</outSequence>
</target>
<parameter name="transport.jms.ContentType">
<rules>
<jmsProperty>contentType</jmsProperty>
<default>application/xml</default>
</rules>
</parameter>
<description/>
</proxy>

```

## VFS Transport

The **Virtual File System (VFS) transport** is used by WSO2 ESB to process files in the specified source directory. After processing the files, it moves them to a specified location or deletes them. Note that files cannot remain in the source directory after processing or they will be processed again, so if you need to maintain these files or keep track of which files have been processed, specify the option to move them instead of deleting them after processing. If you want to move files into a database, use the VFS transport and the [DBReport mediator](#) (for an example, see [Sample 271: File Processing](#)).

[Enabling the transport](#) | [Configuring the endpoint](#) | [Security](#) | [Failure tracking](#) | [Transferring large files](#) | [VFS service-level parameters](#) | [VFS URL parameters](#) | [Samples](#)

### Enabling the transport

This transport is based on the Apache Commons VFS implementation, which is provided in the <ESB\_HOME>/repository/components/plugins/commons-vfs<version>.jar file. The transport is a module of the Apache Synapse project, and the synapse-vfs-transport.jar file contains the necessary classes, including those that implement the listener and sender APIs:

- org.apache.synapse.transport.vfs.VFSTransportListener
- org.apache.synapse.transport.vfs.VFSTransportSender

### To enable the VFS transport

- Edit the <ESB\_HOME>/repository/conf/axis2/axis2.xml file and uncomment the VFS listener and the VFS sender as follows:

```
<transportReceiver name="vfs"
class="org.apache.synapse.transport.vfs.VFSTransportListener" />
...
<transportSender name="vfs"
class="org.apache.synapse.transport.vfs.VFSTransportSender" />
```

### Configuring the endpoint

To configure a VFS endpoint, use the `vfs:file` prefix in the URI. For example:

```
<endpoint>
 <address uri="vfs:file:///home/user/test/out" />
</endpoint>
```

### Security

The VFS transport supports the **FTPS protocol** with **Secure Sockets Layer (SSL)**. The configuration is identical to other protocols with the only difference being the URL prefixes and parameters. For more information, see [VFS URL parameters](#) below.

#### **Securing VFS password in a proxy service**

The following instructions describe how to secure the VFS password in a proxy service configuration.

1. Provide configuration for the password decryption by adding the following lines in <ESB\_HOME>/repository/conf/axis2/axis2.xml file:

```
<transportReceiver class="org.apache.synapse.transport.vfs.VFSTransportListener" name="vfs">
<parameter locked="false" name="keystore.identity.location">repository/resources/security/wso2carbon.jks</parameter>
<parameter locked="false" name="keystore.identity.type">JKS</parameter>
<parameter locked="false" name="keystore.identity.store.password">wso2carbon</parameter>
<parameter locked="false" name="keystore.identity.key.password">wso2carbon</parameter>
<parameter locked="false" name="keystore.identity.alias">wso2carbon</parameter>
</transportReceiver>
```

If you need to secure the passwords provided in the above configuration, you can do so using [secure vault](#). The secured configuration would look like this:

```
<transportReceiver
class="org.apache.synapse.transport.vfs.VFSTransportListener" name="vfs">
<parameter locked="false"
name="keystore.identity.location">repository/resources/security/wso2carbon
.jks</parameter>
<parameter locked="false" name="keystore.identity.type">JKS</parameter>
<parameter locked="false" name="keystore.identity.store.password"
svns:secretAlias="vfs.transport.keystore.password">password</parameter>
<parameter locked="false" name="keystore.identity.key.password"
svns:secretAlias="vfs.transport.key.password">password</parameter>
<parameter locked="false"
name="keystore.identity.alias">wso2carbon</parameter>
</transportReceiver>
```

2. Manually encrypt the password using Cipher Tool. See, [Encrypting Passwords in Cipher Tool](#) in the administration guide.
3. Provide the encrypted password value in your proxy configuration by adding the following parameter:

```
<parameter
name="transport.vfs.FileURI">smb://{{wso2:vault-decrypt('encryptedValue')}}</parameter>
```

## Failure tracking

To track failures in file processing, which can occur when a resource becomes unavailable, the VFS transport creates and maintains a failed records file. This text file contains a list of files that failed to be processed. When a failure occurs, an entry with the failed file name and the timestamp is logged in the text file. When the next polling iteration occurs, the VFS transport checks each file against the failed records file, and if a file is listed as a failed record, it will skip processing and schedule a move task to move that file.

## Transferring large files

If you need to transfer large files using the VFS transport, you can avoid out-of-memory failures by taking the following steps:

1. In <ESB\_HOME>/repository/conf/axis2/axis2.xml, in the messageBuilders section, add the

binary message builder as follows:

```
<messageBuilder contentType="application/binary"
class="org.apache.axis2.format.BinaryBuilder"/>
```

and in the `messageFormatters` section, add the binary message formatter as follows:

```
<messageFormatter contentType="application/binary"
class="org.apache.axis2.format.BinaryFormatter"/>
```

2. In the proxy service where you use the VFS transport, add the following parameter to enable streaming (see [VFS service-level parameters](#) below for more information):

```
<parameter name="transport.vfs.Streaming">true</parameter>
```

3. In the same proxy service, before the Send mediator, add the following property:

### Note

You also need to add the following property if you want to use the VFS transport to transfer files from VFS to VFS.

```
<property name="ClientApiNonBlocking" value="true" scope="axis2"
action="remove"/>
```

For more information, see [Example 3 of the Send Mediator](#).

### VFS service-level parameters

The VFS transport does not have any global parameters to be configured. Rather, it has a set of service-level parameters that must be specified for each proxy service that uses the VFS transport.

Parameter Name	Description
transport.vfs.FileURI	<p>The URI where the files you want to process are located. You can specify a URL (see <a href="#">VFS URL parameters</a> below).</p> <p>When you need to access the absolute path of the URL, you can define the</p> <pre>&lt;parameter name="transport.vfs.FileURI"&gt;sftp://[ username:[ port]][ absolute-path]?sftpPathFromRoot=true&lt;/parameter&gt;</pre>
transport.vfs.ContentType	<p>Content type of the files processed by the transport. To specify the encoding, use a semi-colon and the character set. For example:</p> <pre>&lt;parameter name="transport.vfs.ContentType"&gt;text/plain; charset=UTF-8&lt;/parameter&gt;</pre> <p>When writing a file, you can set a different encoding with the <code>CHARACTER_SET_ENCODING</code> property.</p> <pre>&lt;property name="CHARACTER_SET_ENCODING" value="UTF-8" scope="axis2" /&gt;</pre>
transport.vfs.FileNamePattern	If the VFS listener should process only a subset of the files available at the <code>FileURI</code> , you can use the <code>FileNamePattern</code> parameter to select those files by name using a regular expression.

transport.PollInterval	The polling interval for the transport receiver to poll the file URI location. This value is in seconds. Unless you add "ms" for milliseconds, e.g., "2" or "2000ms" to specify 2 seconds.
transport.vfs.ActionAfterProcess	Whether to move, delete or take no action on the files after the transport has processed them.
transport.vfs.ActionAfterFailure	Whether to move, delete or take no action on the files if a failure occurs.
transport.vfs.MoveAfterProcess	Where to move the files after processing if ActionAfterProcess is MOVE.
transport.vfs.MoveAfterFailure	Where to move the files after processing if ActionAfterFailure is MOVE.
transport.vfs.ReplyFileURI	The location where reply files should be written by the transport.
transport.vfs.ReplyFileName	The name for reply files written by the transport.
transport.vfs.MoveTimestampFormat	The pattern/format of the timestamps added to file names as prefixes when moving files.
transport.vfs.Streaming	Whether files should be transferred in streaming mode, which is useful when dealing with large files.
transport.vfs.ReconnectTimeout	Reconnect timeout value in seconds to be used in case of an error when trying to connect to the file system.
transport.vfs.MaxRetryCount	Maximum number of retry attempts to carry out in case of errors.
transport.vfs.Append	When writing the response to a file, whether the response should be appended to the file or overwriting the file. This value should be defined as a query parameter in the file URI. For example: <pre>"vfs:file:///home/user/test/out? transport.vfs.Append=true"</pre> or: <pre>&lt;parameter name= "transport.vfs.ReplyFileURI"&gt; file:///home/user/test/out? transport.vfs.Append=true &lt;/parameter&gt;</pre>
transport.vfs.MoveAfterFailedMove	Where to move the failed file.
transport.vfs.FailedRecordsFileName	The name of the file that maintains the list of failed files.

transport.vfs.FailedRecordsFileDestination	Where to store the failed records file.
transport.vfs.MoveFailedRecordTimestampFormat	Entries in the failed records file include the name of the file that failed and the timestamp property configures the time stamp format.
transport.vfs.FailedRecordNextRetryDuration	The time in milliseconds to wait before retrying the move task.
transport.vfs.Locking	By default, file locking is enabled in the VFS transport. This parameter lets you enable locking on a per service basis. You can also disable locking globally by specifying the <code>transport.vfs.locking=false</code> configuration selectively enable locking only for a set of services.
transport.vfs.FileProcessCount	This setting allows you to throttle the VFS listener by processing files in batches. You can specify the number of files you want to process in each batch.
transport.vfs.FileProcessInterval	The interval in milliseconds between two file processes.
transport.vfs.ClusterAware	Whether VFS coordination support is enabled in a clustered deployment or not.
transport.vfs.FileSizeLimit	Only file sizes that are less than the defined limit will be processed.
transport.vfs.AutoLockReleaseInterval	The timeout value for stale locks where the VFS transport will ignore those locks if the period is reached. (The time period is calculated from the time the lock is created until it is accessed.) If you need stale locks to never timeout provide -1 as the timeout value.
transport.vfs.SFTPIdentities	Location of the private key
transport.vfs.SFTPIIdentityPassPhrase	Passphrase of the private key
transport.vfs.SFTPUserDirIsRoot	If the SFTP user directory should be treated as root

## VFS URL parameters

VFS connection-specific parameters can be set as URL query parameters. For example, to use FTPS with SSL, you could specify the URL as follows:

```
<parameter
name="transport.vfs.FileURI">vfs:ftps://test:test123@10.200.2.63/vfs/in?vfs.ssl.keystore=/home/user/openssl/keystore.jks&vfs.ssl.truststore=/home/user/openssl/vfs-truststore.jks&vfs.ssl.kspassword=importkey&vfs.ssl.tspassword=wso2vfs&vfs.ssl.keypassword=importkey</parameter>
```

Following are details on the URL parameters you can set. To configure the proxy over ftp/sftp click [here](#).

Parameter Name	Description	Possible Values
vfs.passive	Enable FTP passive mode. This is required when the FTP client and server are not in the same network.	true   false

transport.vfs.Append	If file with same name exists, this parameter tells whether to create a new file and write content or append content to existing file	true   false
vfs.protection	Set data channel protection level using FTP PROT command	<ul style="list-style-type: none"> <li>• C - Clear</li> <li>• S - Safe(SSL protocol only)</li> <li>• E - Confidential(SS protocol only)</li> <li>• P - Private</li> </ul>
vfs.ssl.keystore	Private key store to use for mutual SSL. Your keystore must be signed by a certificate authority. For more information, see <a href="http://docs.oracle.com/cd/E19509-01/820-3503/ggfen/index.html">http://docs.oracle.com/cd/E19509-01/820-3503/ggfen/index.html</a> .	String - Path of keystore
vfs.ssl.kspassword	Private key store password	String
vfs.ssl.keypassword	Private key password	String
vfs.ssl.truststore	Trust store to use for FTPS	String - Path of keystore
vfs.ssl.tspassword	Trust store password	String
transport.vfs.CreateFolder	If the directory does not exists create and write the file	true   false
transport.vfs.SendFileSynchronously	Whether to send files synchronously to the file host. When this parameter is set to true, files will be sent one after another to the file host. This synchronous write can be configured on a per host basis.	true   false

## Samples

- Sample 254: Using the File System as Transport Medium (VFS)
- Sample 255: Switching from FTP Transport Listener to Mail Transport Sender
- Sample 265: Accessing a Windows Share Using the VFS Transport
- Sample 271: File Processing
- Sample 654: Smooks Mediator

## Local Transport

Apache Axis2's local transport implementation is used to make fast, in-VM (Virtual Machine) service calls and transfer data within proxy services. The transport does not have a receiver implementation. The following class implements the sender API:

- `org.apache.axis2.transport.local.NonBlockingLocalTransportSender`

- WS-Security cannot be used with the local transport. Since the local is mainly used to make calls within the same VM, WS-Security is generally not required in scenarios where it is used.
  - If you want to make calls across tenants, you should use a non local transport even if they run from the same VM.
  - If you need to use local transport with callout mediator, you do not need to perform configuration mentioned in this section as callout mediator requires blocking local transport which is configured by default in WSO2 ESB distribution.

To use this transport, configure an endpoint with the `local://` prefix. For example, to make an in-VM call to the HelloService, use `local://services/HelloService`. Note that the local transport cannot be used to send

REST API calls, which require the HTTP/S transports.

### Configuring the Local Transport

By default, WSO2 ESB provides CarbonLocalTransportSender and CarbonLocalTransportReceiver, which are used for internal communication among Carbon components and are not suitable for ESB service invocation. To enable the local transport for service invocation, follow these steps.

1. In the carbon.xml file at location <ESB\_HOME>/repository/conf, an endpoint is available as follows by default.

```
<ServerURL>local://services/</ServerURL>
```

Replace it with

```
<ServerURL>https://${carbon.local.ip}:${carbon.management.port}${carbon.context}/services/</ServerURL>
```

2. In the axis2.xml file at location <PRODUCT\_HOME>/repository/conf/axis2/axis2.xml, there is a transport sender and receiver named 'local' specified as follows in two different places:

```
<transportReceiver name="local"
class="org.wso2.carbon.core.transports.local.CarbonLocalTransportReceiver" />

<transportSender name="local"
class="org.wso2.carbon.core.transports.local.CarbonLocalTransportSender" />
```

Remove both these lines and add following line.

```
<transportSender name="local"
class="org.apache.axis2.transport.local.NonBlockingLocalTransportSender" />
```

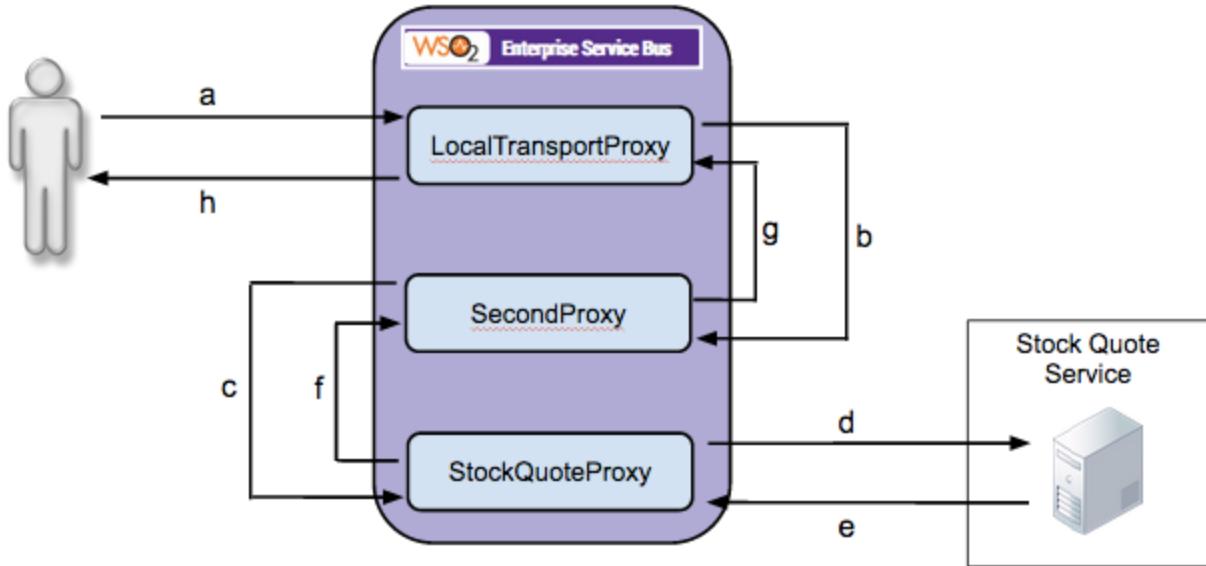
For more information about transports, see [Working with Transports](#).

### Example

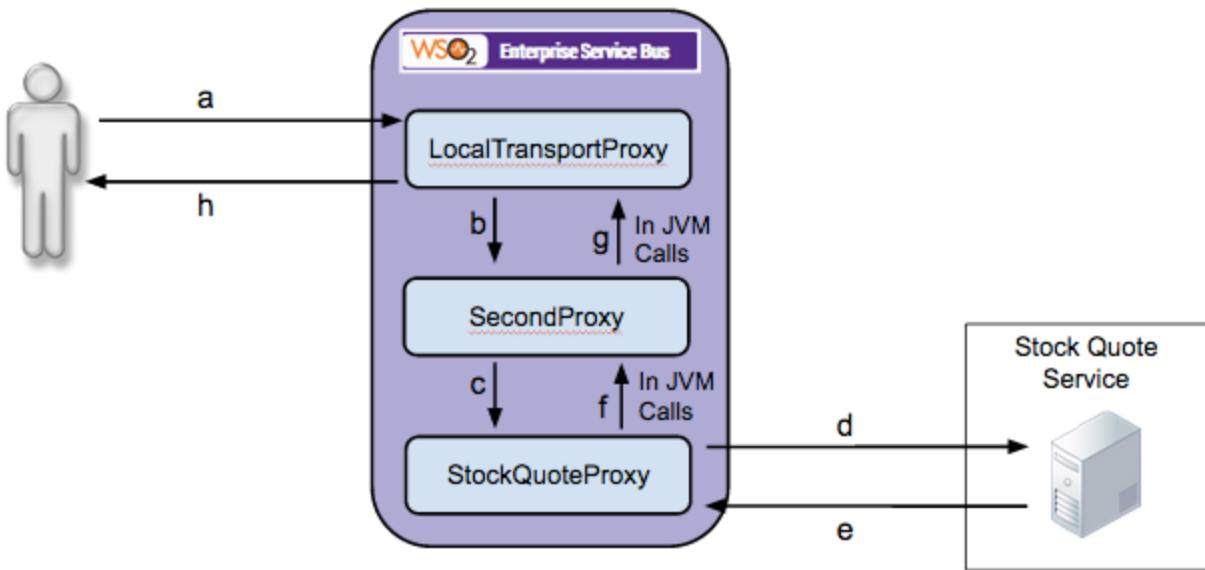
There are three proxy services in WSO2 ESB named LocalTransportProxy, SecondProxy and StockQuoteProxy. The invocation of services take place in the following order.

- The stockquote client invokes LocalTransportProxy.
- The message received is forwarded to SecondProxy.
- The message is forwarded to StockQuoteProxy.
- StockQuoteProxy invokes the backend service.
- StockQuoteProxy receives a response.
- The response is returned by StockQuoteProxy to SecondProxy.
- The response is returned by SecondProxy to LocalTransportProxy.
- The response is returned by LocalTransportProxy to the client.

When local transport is not used, the messages sent by one proxy service to another (i.e. flows b, c and g) goes through the network as shown in the following diagram.



Local transport can be used as shown below to prevent message flows between proxy services from going through the network. When local transport calls are in JVM calls, the time taken for communication between proxy services will be reduced since no network overhead will be introduced.



Run [sample 268](#) for a demonstration of this scenario.

### MailTo Transport

The MailTo transport supports sending messages (E-Mail) over SMTP and receiving messages over POP3 or IMAP. This transport implementation is available as a module of the WS-Commons Transports project.

### Enabling the transport

The JAR consisting of the MailTo transport implementation is named `axis2-transport-mail.jar` and the following sender and receiver classes should be included in the ESB configuration to enable the MailTo transport:

- `org.apache.axis2.transport.mail.MailTransportSender`
- `org.apache.axis2.transport.mail.MailTransportListener`

### To enable the MailTo transport sender

- Edit the <ESB\_HOME>/repository/conf/axis2/axis2.xml file and uncomment the MailTo sender as follows:

```
<transportSender name="mailto"
class="org.apache.axis2.transport.mail.MailTransportSender">
 <parameter name="mail.smtp.host">smtp.gmail.com</parameter>
 <parameter name="mail.smtp.port">587</parameter>
 <parameter name="mail.smtp.starttls.enable">true</parameter>
 <parameter name="mail.smtp.auth">true</parameter>
 <parameter name="mail.smtp.user">synapse.demo.0</parameter>
 <parameter name="mail.smtp.password">mailpassword</parameter>
 <parameter name="mail.smtp.from">synapse.demo.0@gmail.com</parameter>
</transportSender>
```

## Note

If you want to use multiple mail boxes to send emails, you can have multiple MailTo senders in the <ESB\_HOME>/repository/conf/axis2/axis2.xml file, and include the name of the relevant <transportSender> in the synapse configuration to use a particular mail box.

For example, if you edit the <ESB\_HOME>/repository/conf/axis2/axis2.xml file and add the following <transportSender>:

```
<transportSender name="mailtoWSO2"
class="org.apache.axis2.transport.mail.MailTransportSender">
 <parameter name="mail.smtp.host">smtp.gmail.com</parameter>
 <parameter name="mail.smtp.port">587</parameter>
 <parameter name="mail.smtp.starttls.enable">true</parameter>
 <parameter name="mail.smtp.auth">true</parameter>
 <parameter name="mail.smtp.user">synapse.demo.0</parameter>
 <parameter name="mail.smtp.password">mailpassword</parameter>
 <parameter name="mail.smtp.from">synapse.demo.0@gmail.com</parameter>
</transportSender>
```

You can use the mailtoWSO2 mail box by specifying the following in a synapse configuration:

```
<endpoint>
 <address uri="mailtoWSO2:user@host" />
</endpoint>
```

The MailTo transport sender is generally configured globally so that all services can share the same transport sender configuration.

### To enable the MailTo transport receiver

- Edit the <ESB\_HOME>/repository/conf/axis2/axis2.xml file and uncomment the MailTo listener as follows:

```
<transportReceiver name="mailto"
class="org.apache.axis2.transport.mail.MailTransportListener">
</transportReceiver>
```

The MailTo transport receiver should be configured at service level and each service configuration should explicitly state the mail transport receiver configuration. This is required to enable different services to receive mails over different mail accounts and configurations.

## Note

You need to provide correct parameters for a valid mail account at the service level.

### Service Level Transport Receiver Parameters

The MailTo transport listener implementation can be configured by setting the parameters as described in the JavaMail API documentation. For IMAP related properties, see [Package Summary - IMAP](#). For POP3 properties, see [Package Summary - POP3](#). The MailTo transport listener also supports the following transport parameters in addition to the parameters described in the JavaMail API documentation.

In the following transport parameter tables, the literals displayed in italics in the **Possible Values** column should be considered as fixed literal constant values. Those values can be directly specified in the transport configuration.

Parameter Name	Description	e.g.Required	Possible Values	Default Value
transport.mail.Address	The mail address from which this service should fetch incoming mails.	Yes	A valid e-mail address	
transport.mail.Folder	The mail folder in the server from which the listener should fetch incoming mails.	No	A valid mail folder name (e.g., inbox)	inbox folder if that is available or else the root folder
transport.mail.Protocol	The mail protocol to be used to receive messages.	No	<i>pop3</i> , <i>imap</i>	imap
transport.mail.PreserveHeaders	A comma separated list of mail header names that this receiver should preserve in all incoming messages.	No	A comma separated list	
transport.mail.RemoveHeaders	A comma separated list of mail header names that this receiver should remove from incoming messages.	No	A comma separated list	
transport.mail.ActionAfterProcess	Action to perform on the mails after processing them.	No	<i>MOVE</i> , <i>DELETE</i>	DELETE

transport.mail.ActionAfterFailure	Action to perform on the mails after a failure occurs while processing them.	No	MOVE, DELETE	DELETE
transport.mail.MoveAfterProcess	Folder to move the mails after processing them.	Required if ActionAfterProcess is MOVE	A valid mail folder name	
transport.mail.MoveAfterFailure	Folder to move the mails after encountering a failure.	Required if ActionAfterFailure is MOVE	A valid mail folder name	
transport.mail.ProcessInParallel	Whether the receiver should process incoming mails in parallel or not. This works only if the mail protocol supports processing incoming mails in parallel. (e.g., IMAP)	No	true, false	false
transport.ConcurrentPollingAllowed	Whether the receiver should poll for multiple messages concurrently.	No	true, false	false
transport.mail.MaxRetryCount	Maximum number of retry operations to be performed when fetching incoming mails.	Yes	A positive integer	
transport.mail.ReconnectTimeout	The reconnect timeout in milliseconds to be used when fetching incoming mails.	Yes	A positive integer	

### Global Transport Sender Parameters

For a list of parameters supported by the MailTo transport sender, see [Package Summary - SMTP](#). In addition to the parameters described there, the MailTo transport sender supports the following parameters.

Parameter Name	Description	Required	Possible Values	Default Value
transport.mail.SMTPBccAddresses	If one or more e-mail addresses need to be specified as BCC addresses for outgoing mails, this parameter can be used.	No	A comma separated list of e-mail addresses	
transport.mail.Format	Format of the outgoing mail.	No	Text, Multi part	Text

For more information, see

- [Working with Transports](#)
- [Sample 255: Switching from FTP Transport Listener to Mail Transport Sender](#)
- [Sample 256: Proxy Services with the MailTo Transport](#)
- [Sample 271: File Processing](#)

### MSMQ Transport

The **msmq**: component is a transport for working with Microsoft Message Queuing

- . This component natively sends and receives direct allocated ByteBuffer instances. This allows you to access the JNI layer without expensive memory copying. In fact, using ByteBuffer created with the method allocateDirect can be passed to the JNI layer, and the native code is able to directly access the memory.

### URI format

```
msmq:msmqQueueName
```

### Examples

```
msmq:DIRECT=OS:localhost\\private$\\test?concurrentConsumers=1
msmq:DIRECT=OS:localhost\\private$\\test?deliveryPersistent=true&priority=5&timeToLive=10
```

Configuring the MSMQ transport

1. In the axis2.xml file at location <PRODUCT\_HOME>/repository/conf/axis2, define the MSMQ sender/listener pair as follows:

```
<transportSender name="msmq" class="org.apache.axis2.transport.msmq.MSMQSender" />

<transportReceiver name="msmq"
class="org.apache.axis2.transport.msmq.MSMQListener">
 <parameter name="msmq.receiver.host"
locked="false">localhost</parameter>
</transportReceiver>
```

2. Download axis2-transport-msmq-1.1.0-wso2v6.jar for **64-bit** or **32-bit** operating systems to <ESB\_HOME>/repository/components/dropins. This file provides the JNI invocation required by MSMQ bridging. Be sure to download the correct one for your operating system.
3. Make sure MQ installed and running. For more information, see <http://msdn.microsoft.com/en-us/library/aa967729.aspx>.
4. Make sure that you have installed Visual C++ 2008 (VC9) and that it works with Microsoft Visual Studio 2008 Express.

The MSMQ examples only work on Windows, since they invoke Microsoft C++ API for MSMQ via JNI invocation.

For more information, see:

- [Configure the JMS Transport with MSMQ](#)
- [Sample 270: Transport switching from HTTP to MSMQ and MSMQ to HTTP](#)

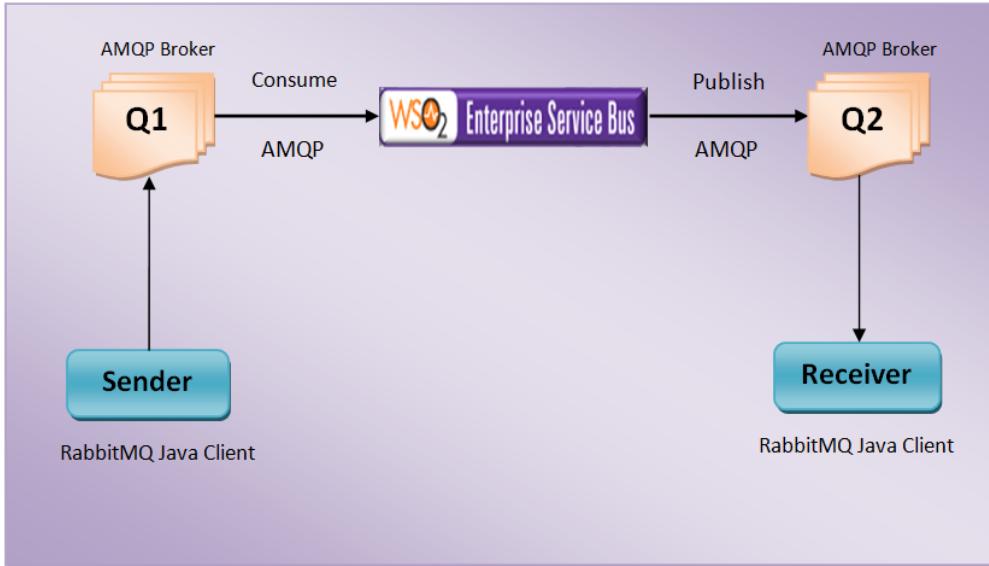
### RabbitMQ AMQP Transport

**AMQP** is a wire-level messaging protocol that describes the format of the data that is sent across the network. If a system or application can read and write AMQP, it can exchange messages with any other system or application that understands AMQP, regardless of the implementation language.

The RabbitMQ AMQP transport is implemented using the [RabbitMQ Java Client](#). It allows you to send or receive AMQP messages by calling an AMQP broker (RabbitMQ) directly, without the need to use different transport mechanisms, such as JMS.

The following diagram illustrates a scenario where the ESB uses the RabbitMQ AMQP transport to exchange

messages between RabbitMQ Java Clients by calling RabbitMQ brokers.



As you can see in the diagram, the Sender uses the RabbitMQ Java Client to publish messages to an AMQP queue (Q1), and the Receiver users it to consume messages from an AMQP queue (Q2). In this scenario, a proxy service in the ESB listens to Q1, and when a message becomes available on the queue, the proxy service consumes it and publishes it to Q2.

This section provides information on configuring the RabbitMQ transport in WSO2 ESB along with information on how to enable connection recovery, how to enable SSL support as well as a sample RabbitMQ proxy service to describe how to use the RabbitMQ transport.

[ [Configuring the RabbitMQ AMQP transport](#) ] [ [RabbitMQ AMQP transport parameters](#) ] [ [Connection recovery in RabbitMQ](#) ] [ [SSL enabled RabbitMQ transport](#) ] [ [Creating the RabbitMQ proxy service](#) ]

## Configuring the RabbitMQ AMQP transport

1. Open `<ESB_HOME>/repository/conf/axis2/axis2.xml` for editing.
2. In the transport **listeners** section, add the following RabbitMQ transport listener, replacing the values with your host, port, username, and password to connect to the AMQP broker.

```

<transportReceiver name="rabbitmq"
class="org.apache.axis2.transport.rabbitmq.RabbitMQListener">
 <parameter name="AMQPConnectionFactory" locked="false">
 <parameter name="rabbitmq.server.host.name"
locked="false">192.168.0.3</parameter>
 <parameter name="rabbitmq.server.port" locked="false">5672</parameter>
 <parameter name="rabbitmq.server.user.name" locked="false">user</parameter>
 <parameter name="rabbitmq.server.password"
locked="false">abc123</parameter>
 </parameter>
</transportReceiver>

```

As an optional step, you can create multiple connection factories if you want this listener to connect to multiple brokers.

3. In the transport **senders** section, add the following RabbitMQ transport sender, which will be used for sending AMQP messages to a queue:

```
<transportSender name="rabbitmq"
class="org.apache.axis2.transport.rabbitmq.RabbitMQSender" />
```

#### 4. Start the ESB server.

### RabbitMQ AMQP transport parameters

Following are details on the listener parameters you can set:

Parameter	Description	Required
rabbitmq.connection.factory	The name of the connection factory.	Yes
rabbitmq.exchange.name	Name of the RabbitMQ exchange to which the queue is bound. Use this parameter instead of <code>rabbitmq.queue.routing.key</code> , if you need to use the default exchange and publish to a queue.	No
rabbitmq.queue.name	The queue name to send or consume messages. If you do not specify this parameter, you need to specify the <code>rabbitmq.queue.routing.key</code> parameter.	Yes
rabbitmq.queue.auto.ack	Whether to automatically send back an acknowledgement when consuming messages from queue.	No
rabbitmq.consumer.tag	The client-generated consumer tag to establish context.	No
rabbitmq.channel.consumer.qos	The consumer qos value. You need to specify this parameter only if the <code>rabbitmq.queue.auto.ack</code> parameter is set to <code>false</code> .	No
rabbitmq.queue.durable	Whether the queue should remain declared even if the broker restarts.	No
rabbitmq.queue.exclusive	Whether the queue should be exclusive or should be consumable by other connections.	No
rabbitmq.queue.auto.delete	Whether to keep the queue even if it is not being consumed anymore.	No
rabbitmq.queue.routing.key	The routing key of the queue.	No
rabbitmq.exchange.type	The type of the exchange.	No
rabbitmq.exchange.durable	Whether the exchange should remain declared even if the broker restarts.	No
rabbitmq.exchange.auto.delete	Whether to keep the queue even if it is not used anymore.	No

<code>rabbitmq.message.content.type</code>	The content type of the consumer.	No. The default value is <code>text/xml</code> .
	<p><b>Note</b></p> <p>If the content type is specified in the message, this parameter does not override the specified content type.</p>	

Following are details on the sender parameters you can set:

Parameter	Description	Required
<code>rabbitmq.server.host.name</code>	Host name of the server.	Yes
<code>rabbitmq.server.port</code>	Port number of the server.	Yes
<code>rabbitmq.exchange.name</code>	The name of the RabbitMQ exchange to which the queue is bound. Use this parameter instead of <code>rabbitmq.queue.routing.key</code> , if you need to use the default exchange and publish to a queue.	No
<code>rabbitmq.queue.routing.key</code>	The exchange and queue binding key that will be used to route messages.	No
<code>rabbitmq.replyto.name</code>	The name of the call back- queue. Specify this parameter if you expect a response.	No
<code>rabbitmq.queue.delivery.mode</code>	The delivery mode of the queue. Possible values are 1 and 2. 1 - Non-persistent. 2 - Persistent. This is the default value.	No
<code>rabbitmq.exchange.type</code>	The type of the exchange.	No
<code>rabbitmq.queue.name</code>	The queue name to send or consume messages. If you do not specify this parameter, you need to specify the <code>rabbitmq.queue.routing.key</code> parameter.	Yes
<code>rabbitmq.queue.durable</code>	Whether the queue should remain declared even if the broker restarts. The default value is <code>false</code> .	No
<code>rabbitmq.queue.exclusive</code>	Whether the queue should be exclusive or should be consumable by other connections. The default value is <code>false</code> .	No
<code>rabbitmq.queue.auto.delete</code>	Whether to keep the queue even if it is not being consumed anymore. The default value is <code>false</code> .	No
<code>rabbitmq.exchange.durable</code>	Whether the exchange should remain declared even if the broker restarts.	No

For the `rabbitmq.server` properties refer to the server on which RabbitMQ is running.

## Connection recovery in RabbitMQ

In case of a network failure or broker shutdown, the ESB will try to recreate the connection. The following parameters need to be configured in the transport listener to enable connection recovery in RabbitMQ.

```
<parameter name="rabbitmq.connection.retry.interval" locked="false">10000</parameter>
<parameter name="rabbitmq.connection.retry.count" locked="false">5</parameter>
```

If the parameters specified above are set, the ESB will retry 5 times with 10000 ms time intervals to reconnect when the connection is lost. If reconnecting also fails, the ESB will terminate the connection. If you do not specify values for the above parameters, the ESB uses 30000ms as the default retry interval and 3 as the default retry count.

Optionally, you can configure the following parameter in the listener:

```
<parameter name="rabbitmq.server.retry.interval" locked="false">10000</parameter>
```

The parameter specified above sets the retry interval with which the RabbitMQ client tries to reconnect. Generally having this value less than the value specified as `rabbitmq.connection.retry.interval` will help synchronize the reconnection of the ESB and the RabbitMQ client.

## SSL enabled RabbitMQ transport

To enable SSL support in RabbitMQ, you need to configure the transport listener with parameters required to enable SSL, as well as the parameters that provide information related to keystores and truststores.

Following is a sample SSL enabled transport listener configuration:

```
<transportReceiver name="rabbitmq"
class="org.apache.axis2.transport.rabbitmq.RabbitMQListener">
<parameter name="AMQPConnectionFactoryKS" locked="false">
<parameter name="rabbitmq.server.host.name" locked="false">localhost</parameter>
<parameter name="rabbitmq.server.port" locked="false">5671</parameter>
<parameter name="rabbitmq.server.user.name" locked="false"></parameter>
<parameter name="rabbitmq.server.password" locked="false"></parameter>
<parameter name="rabbitmq.connection.retry.interval"
locked="false">10000</parameter>
<parameter name="rabbitmq.connection.retry.count" locked="false">5</parameter>
<parameter name="rabbitmq.connection.ssl.enabled" locked="false">true</parameter>
<parameter name="rabbitmq.connection.ssl.version" locked="false">SSL</parameter>
<parameter name="rabbitmq.connection.ssl.keystore.location"
locked="false">../client/keycert.p12</parameter>
<parameter name="rabbitmq.connection.ssl.keystore.type"
locked="false">PKCS12</parameter>
<parameter name="rabbitmq.connection.ssl.keystore.password"
locked="false">MySecretPassword</parameter>
<parameter name="rabbitmq.connection.ssl.truststore.location"
locked="false">ssl/rabbitstore</parameter>
<parameter name="rabbitmq.connection.ssl.truststore.type"
locked="false">JKS</parameter>
<parameter name="rabbitmq.connection.ssl.truststore.password"
locked="false">rabbitstore</parameter>
</parameter>
</transportReceiver>
```

Configuring parameters that provide information related to keystores and truststores can be optional based on your

broker configuration.

For example, if `fail_if_no_peer_cert` is set to `false` in the RabbitMQ broker configuration, then you only need to specify `<parameter name="rabbitmq.connection.ssl.enabled" locked="false">true</parameter>`. Additionally, you can also set `<parameter name="rabbitmq.connection.ssl.version" locked="false">true</parameter>` parameter to specify the SSL version. If `fail_if_no_peer_cert` is set to `true`, you need to provide keystore and truststore information.

Following is a sample broker configuration where `fail_if_no_peer_cert` is set to `false`:

```
{ssl_options, [{cacertfile,"/path/to/testca/cacert.pem"}, {certfile,"/path/to/server/cert.pem"}, {keyfile,"/path/to/server/key.pem"}, {verify,verify_peer}, {fail_if_no_peer_cert,false}]}
```

The same parameter names are applicable for transport sender configurations.

### Creating the RabbitMQ proxy service

Following is a sample RabbitMQ proxy service named AMQPProxy, which consumes AMQP messages from one RabbitMQ broker and publishes them to another:

**Sample Proxy Service**

```
<proxy xmlns="http://ws.apache.org/ns/synapse" name="AMQPProxy" transports="rabbitmq" statistics="disable" trace="disable" startOnLoad="true">
 <target>
 <inSequence>
 <log level="full"/>
 <property name="OUT_ONLY" value="true"/>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
 </inSequence>
 <endpoint>
 <address
uri="rabbitmq:/AMQPProxy?rabbitmq.server.host.name=192.168.0.3&rabbitmq.server.port=5672&rabbitmq.server.user.name=user&rabbitmq.server.password=abc123&rabbitmq.queue.name=queue2&rabbitmq.exchange.name=exchange2"/>
 </endpoint>
 </target>
 <parameter name="rabbitmq.queue.name">queue1</parameter>
 <parameter name="rabbitmq.exchange.name">exchange1</parameter>
 <parameter name="rabbitmq.connection.factory">AMQPConnectionFactory</parameter>
 <description></description>
 </proxy>
```

Note the following:

- The transport key name is `rabbitmq`. You need to specify this key name in the `transports` parameter (ie., `transports="rabbitmq"`).
- The proxy is defined as `OUT_ONLY`, because it does not expect a response from the endpoint.
- The endpoint specifies where the messages will be published. The URI prefix is `rabbitmq` so that the RabbitMQ AMQP transport will be used to publish the message. Be sure to specify the rest of the parameters in the URI as shown in the sample proxy service above. (NOTE: if you are configuring the URI through the

management console instead of entering the configuration directly in the configuration file, you must encode the ampersands in the URI as "&" instead of "&".) If you do not know which [RabbitMQ exchange](#) to use, leave the value blank to use the default exchange.

- The `rabbitmq.queue.name` parameter specifies the queue on which the proxy service listens and consumes messages. If you do not specify a name for this parameter, the name of the proxy service will be used as the queue name.
- The `rabbitmq.exchange.name` parameter specifies the RabbitMQ exchange to which the queue is bound. If you do not want to use a specific exchange, leave this value blank to use the default exchange.
- The `rabbitmq.connection.factory` parameter specifies the listener that listens on the queue and consumes messages. In this example, the connection factory is set to the name of the listener we created earlier (ie., `AMQPConnectionFactory`).

You can modify the sample proxy service above to handle scenarios where you only want to receive AMQP messages but need to send messages in a different format, or you want to receive messages in a different format and send only AMQP messages. You can also modify the proxy service to work with a different transport. For example, you can create a proxy that uses the RabbitMQ AMQP transport to listen to messages and then sends them over HTTP or JMS.

### ***Sample java clients***

This section describes the sample Java clients that you can use to send and receive AMQP messages. These clients can be used to test the scenario where the Sender publishes a message to a RabbitMQ AMQP queue, which is consumed by the ESB and published to another queue, which in turn is consumed by the Receiver. When you run these clients, the Receiver will get the messages sent by the Sender, confirming that you have correctly configured the RabbitMQ AMQP transport.

AMQP sender

The following Java client sends SOAP XML messages to a RabbitMQ queue:

```

ConnectionFactory factory = new ConnectionFactory();
factory.setHost(host);
factory.setUsername(username);
factory.setPassword(password);
factory.setPort(port);
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();
channel.queueDeclare(queueName, false, false, false, null);
channel.exchangeDeclare(exchangeName, "direct", true);
channel.queueBind(queueName, exchangeName, routingKey);

// The message to be sent
String message = "<soapenv:Envelope
 xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope">\n" +
 "<soapenv:Header>\n" +
 "<soapenv:Body>\n" +
 " <p:greet xmlns:p=\"http://greet.service.kishanthan.org\">\n" +
 " <in>" + name + "</in>\n" +
 " </p:greet>\n" +
 "</soapenv:Body>\n" +
 "</soapenv:Envelope>";

// Populate the AMQP message properties
AMQP.BasicProperties.Builder builder = new
AMQP.BasicProperties().builder();
builder.messageId(messageID);
builder.contentType("text/xml");
builder.replyTo(replyToAddress);
builder.correlationId(correlationId);
builder.contentEncoding(contentEncoding);

// Custom user properties
Map<String, Object> headers = new HashMap<String, Object>();
headers.put("SOAP_ACTION", "greet");
builder.headers(headers);

// Publish the message to exchange
channel.basicPublish(exchangeName, queueName, builder.build(), message.getBytes());

```

This client was tested with SOAP messages sent and consumed from AMQP broker queues with the content type “text/xml”. When specifying the queue name for publishing messages, be sure to specify the same queue where the RabbitMQ transport listener is listening.

AMQP receiver

When specifying the queue name for consuming messages, be sure to specify the same queue configured in the proxy service endpoint.

```

ConnectionFactory factory = new ConnectionFactory();
factory.setHost(hostName);
factory.setUsername(userName);
factory.setPassword(password);
factory.setPort(port);
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();
channel.queueDeclare(queueName, false, false, false, null);
channel.exchangeDeclare(exchangeName, "direct", true);
channel.queueBind(queueName, exchangeName, routingKey);

// Create the consumer
QueueingConsumer consumer = new QueueingConsumer(channel);
channel.basicConsume(queueName, true, consumer);

// Start consuming messages
while (true) {
 QueueingConsumer.Delivery delivery = consumer.nextDelivery();
 String message = new String(delivery.getBody());
}

```

For more information on implementation details of common RabbitMQ use cases, see the following topics:

- [ESB as a RabbitMQ Message Consumer](#)
- [ESB as a RabbitMQ Message Producer](#)
- [Remote Procedure Call\(RPC\) with RabbitMQ](#)

#### ESB as a RabbitMQ Message Consumer

This section describes how WSO2 ESB can be configured as a RabbitMQ message consumer.



Following is a sample scenario that demonstrates how the ESB is configured to listen to a rabbitMQ queue, consume messages, and send the messages to a HTTP back-end service.

#### Note

To create proxy services, sequences, endpoints, message stores and message processors in the ESB, you can either use the ESB Management Console or copy the XML configuration to the source view. To access the source view on the ESB Management Console, go to **Manage -> Service Bus -> Source View**.

#### Prerequisites

- Configure the RabbitMQ AMQP transport. For information on how to configure the transport, see [Configuring the RabbitMQ AMQP transport](#).
- Start the ESB server.

Configure the sample

1. Create a custom proxy service with the following configuration. For more information on creating proxy services, see [Working with Proxy Services](#).

```

<?xml version="1.0" encoding="UTF-8"?>
<proxy
 xmlns="http://ws.apache.org/ns/synapse" name="AMQPProxy" transports="rabbitmq" statics="disable" trace="enable" startOnLoad="true">
 <target>
 <inSequence>
 <log level="full"/>
 <property name="OUT_ONLY" value="true"/>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
 <send>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </inSequence>
 </target>
 <outSequence>
 <drop/>
 </outSequence>
 <parameter name="rabbitmq.queue.name">queue</parameter>
 <parameter name="rabbitmq.exchange.name">exchange</parameter>
 <parameter
 name="rabbitmq.connection.factory">AMQPConnectionFactory</parameter>
 <description/>
 </proxy>

```

2. WSO2 ESB comes with a default Axis2 server, which you can use as the back-end service for this sample. To start the Axis2 server, navigate to <ESB\_HOME>/samples/axis2server, and run axis2Server.sh on Linux or axis2Server.bat on Windows.
3. Deploy the **SimpleStockQuoteService** client by navigating to <ESB\_HOME>/samples/axis2Server/src /SimpleStockQuoteService, and running the **ant** command on the command prompt or shell script. This will build the sample and deploy the service for you. For more information on sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance with a custom proxy service and a back-end service deployed. Next, we will send a message to the back-end service through the ESB using a sample client. Execute the sample client

Run the following client to publish a getquote request to the RabbitMQ server exchange that is running on port 5672

```

ConnectionFactoryfactory =newConnectionFactory();
factory.setHost("localhost");
factory.setUsername("guest");
factory.setPassword("guest");
factory.setPort(5672);
Connectionconnection =factory.newConnection();
Channelchannel =connection.createChannel();
channel.queueDeclare("queue",false,false,null);
channel.exchangeDeclare("exchange","direct",true);
channel.queueBind("queue", "exchange", "route");

//Themessagegetobesent
Stringmessage ="<m:placeOrder xmlns:m=\"http://services.samples\">>" +
+ "<m:order>" +
+ "<m:price>100</m:price>\n" +
+ "<m:quantity>20</m:quantity>" +
+ "<m:symbol>RMQ</m:symbol>" +
+ "</m:order>" +
+ "</m:placeOrder>";

//PopulatetheAMQPmessagelocation
AMQP.BasicProperties.Builderbuilder =newAMQP.BasicProperties().builder();
builder.contentType("text/xml");
builder.contentEncoding(contentEncoding);

//Publishthemessage
channel.basicPublish("exchange", "queue", builder.build(), message.getBytes());

```

#### Analyzing the output

The direct exchange is bound to the queue with route--keyqueue that is consumed by the ESB RabbitMQ transport receiver. From there the message will be sent to the AMQPProxy and it will be forwarded to the given http url.

If you analyze the console running the sample Axis2 server, you will see the following message indicating that the server has accepted an order

```
Accepted order #1 for : 7078 stocks of IBM at $ 73.73786002620719
```

#### ESB as a RabbitMQ Message Producer

This section describes how WSO2 ESB can be used to send messages to a RabbitMQ queue.



Following is a sample scenario that demonstrates how the ESB is configured to listen to HTTP requests and publish them to a RabbitMQ server (message exchange).

## Note

To create proxy services, sequences, endpoints, message stores and message processors in the ESB, you can either use the ESB Management Console or copy the XML configuration to the source view. To access the source view on the ESB Management Console, go to **Manage -> Service Bus -> Source View**.

### Prerequisites

- Configure the RabbitMQ AMQP transport. For information on how to configure the transport, see [Configuring the RabbitMQ AMQP transport](#).
- Start the ESB server.

### Configure the sample

1. Create a custom proxy service with the following configuration. For more information on creating proxy services, see [Working with Proxy Services](#).

```
<?xml version="1.0" encoding="UTF-8"?>
<proxy
 xmlns="http://ws.apache.org/ns/synapse" name="AMQPPProducerSample" transports="http"
 statistics="disable" trace="disable" startOnLoad="true">
 <target>
 <inSequence>
 <property name="OUT_ONLY" value="true"/>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
 <send>
 <endpoint>
 <address
 uri="rabbitmq:/AMQPPProducerSample?rabbitmq.server.host.name=localhost&rabbitmq.server.port=5672&rabbitmq.queue.name=queue&rabbitmq.queue.route.key=route&rabbitmq.exchange.name=exchange"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <description/>
</proxy>
```

2. Use the following as a RabbitMQ consumer that will consume and display the incoming messages to the RabbitMQ queue.

```

ConnectionFactoryfactory =newConnectionFactory();
factory.setHost("localhost");
factory.setUsername("guest");
factory.setPassword("guest");
factory.setPort(5672);

Connectionconnection =factory.newConnection();
Channelchannel =connection.createChannel();
channel.queueDeclare("queue",false,false,false,null);
channel.exchangeDeclare("exchange", "direct",true);
channel.queueBind("queue", "exchange", "route");

//Createtheconsumer
QueueingConsumerconsumer =newQueueingConsumer(channel);
channel.basicConsume("queue",true,consumer);

//Startconsumingmessages
while(true)
{
 QueueingConsumer.Deliverydelivery =consumer.nextDelivery();
 Stringmessage =newString(delivery.getBody());
}

```

Execute the sample client

Execute the following command from <ESB\_HOME>/sample/axis2Client, to send an HTTP message to ESB proxy service.

```

ant stockquote -Daddurl=http://localhost:8280/services/AMQPProducerSample
-Dmode=placeorder

```

Analyzing the output

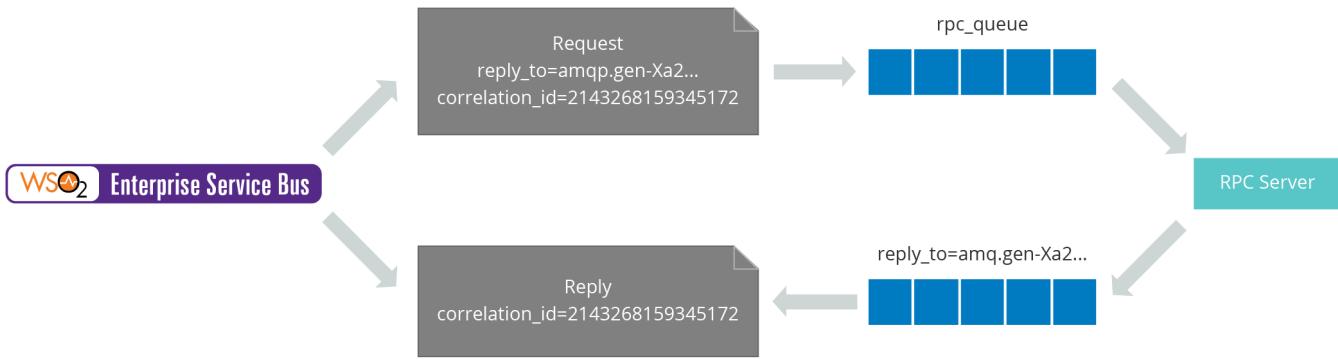
You will see that the http request is sent to the given ESB Proxy service and that it is forwarded to the RabbitMQ server via the RabbitMQ AMQP transport sender.

You can view the messages received at the RabbitMQ queue in the RabbitMQ SimpleProducer console.

#### Remote Procedure Call(RPC) with RabbitMQ

You can send request-response messages using the RabbitMQ transport by implementing a Remote Procedure Call(RPC) scenario with RabbitMQ.

The following diagram illustrates a remote procedure call scenario with RabbitMQ:



The remote procedure call works as follows:

- When WSO2 ESB starts up, it creates an anonymous exclusive callback queue.
- For a remote procedure call request, the ESB sends a message with the following properties:
  - `reply_to` - This is set to the callback queue
  - `correlation_id` - This is set to a unique value for every request.
- The request is then sent to the `rpc_queue`.
- The RPC Server waits for requests on that queue. When a request appears, it does the job and sends a message with the result back to the ESB, using the queue from the `reply_to` field with the same `correlation_id`.
- The ESB waits for data on the `reply_to` queue. When a message appears, it checks the `correlation_id` property. If it matches the value from the request it returns the response to the application.

Following is a sample proxy service named `RabbitMQRPCProxy` that sends request-response messages using the RabbitMQ transport.

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="RabbitMQRPCProxy"
 startOnLoad="true"
 trace="enable"
 transports="http">
<description/>
<target>
<inSequence>
 <log level="full">
 <property name="received" value="true"/>
 </log>
 <send>
 <endpoint>
 <address
uri="rabbitmq://?rabbitmq.server.host.name=localhost&rabbitmq.server.port=5672&
;rabbitmq.server.user.name=guest&rabbitmq.server.password=guest&rabbitmq.queue
.name=rpc_queue&rabbitmq.queue.routing.key=rpc_queue&rabbitmq.replyto.name=dum
my"/>
 </endpoint>
 </send>
</inSequence>
<outSequence>
 <log level="full">
 <property name="response" value="true"/>
 </log>
 <send/>
</outSequence>
</target>
</proxy>

```

Following is the code for a sample RPC server:

```

package rpc;

import com.rabbitmq.client.AMQP.BasicProperties;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.QueueingConsumer;

import java.io.IOException;

public class RPCServer {

 public static void main(String[] argv) {
 Connection connection = null;
 Channel channel;
 try {
 ConnectionFactory factory = new ConnectionFactory();
 factory.setHost("localhost");
 connection = factory.newConnection();
 channel = connection.createChannel();
 QueueingConsumer consumer = new QueueingConsumer(channel);
 channel.basicConsume("rpc_queue", false, consumer);

 System.out.println(" [x] Awaiting RPC requests");
 }
 }
}

```

```

 while (true) {
 String response = null;
 QueueingConsumer.Delivery delivery = consumer.nextDelivery();
 BasicProperties props = delivery.getProperties();
 BasicProperties replyProps =
 new
BasicProperties.Builder().correlationId(props.getCorrelationId()).contentType("text/xm
l")
 .build();

 response =
 "<soapenv:Envelope
xmlns:soapenv=\\"http://schemas.xmlsoap.org/soap/envelope/\\" " +
 "xmlns:ser=\\"http://services.samples\""
xmlns:xsd=\\"http://services.samples/xsd\\>\n" +
 " <soapenv:Header>\n" +
 " <soapenv:Body>\n" +
 " <ser:placeOrder>\n" +
 " <!--Optional:-->\n" +
 " <ser:order>\n" +
 " <!--Optional:-->\n" +
 " <xsd:price>10</xsd:price>\n" +
 " <!--Optional:-->\n" +
 " <xsd:quantity>5</xsd:quantity>\n" +
 " <!--Optional:-->\n" +
 " <xsd:symbol>RMQ</xsd:symbol>\n" +
 " </ser:order>\n" +
 " </ser:placeOrder>\n" +
 " </soapenv:Body>\n" +
 "</soapenv:Envelope>";

 String replyToQueue = props.getReplyTo();
 System.out.println("Publishing to : " + replyToQueue);
 channel.basicPublish("", replyToQueue, replyProps,
response.getBytes("UTF-8"));
 channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
 }

 } catch (InterruptedException e) {
 e.printStackTrace();
 } catch (IOException e) {
 e.printStackTrace();
 } finally {
 if (connection != null) {
 try {
 connection.close();
 } catch (IOException e) {
 e.printStackTrace();
 }
 }
 }
 }
}

```

```
}
```

## TCP Transport

The TCP transport allows you to send and receive SOAP messages over TCP. The TCP transport is included with the WSO2 ESB distribution but must be enabled before use. To enable the TCP transport, open the <ESB\_HOME>/repository/conf/axis2/axis2.xml file in a text editor and add the following transport receiver configuration and sender configuration:

```
<!-- Enable TCP message -->
<transportReceiver name="tcp"
class="org.apache.axis2.transport.tcp.TCPTTransportListener">
 <parameter name="transport.tcp.port">6060</parameter>
</transportReceiver>

<transportSender name="tcp"
class="org.apache.axis2.transport.tcp.TCPTTransportSender" />
```

If you want to use the sample Axis2 client to send TCP messages, uncomment the TCP transport sender configuration in the following file:

`samples/axis2Client/client_repo/conf/axis2.xml`

### Transport receiver parameters

Parameter Name	Description	Required	Possible Values	Default Value
port	The port on which the TCP server should listen for incoming messages	No	A positive integer less than 65535	8000
hostname	The host name of the server to be displayed in WSDLs, etc.	No	A valid host name or an IP address	

For more information, see [Working with Transports](#).

## UDP Transport

The UDP transport allows you the ESB to handle messages in user datagram protocol (UDP) format. The axis2-transport-udp.jar archive file contains the following UDP transport implementation classes, which are part of the Apache WS-Commons Transports project:

- org.apache.axis2.transport.udp.UDPListener
- org.apache.axis2.transport.udp.UDPSender

To enable the UDP transport, open the file <ESB\_HOME>/repository/conf/axis2/axis2.xml in a text editor and add the following transport configurations.

```
<transportReceiver name="udp" class="org.apache.axis2.transport.udp.UDPListener"/>
<transportSender name="udp" class="org.apache.axis2.transport.udp.UDPSender" />
```

If you want to use the sample Axis2 client to send UDP messages, add the UDP transport sender configuration in the <ESB\_HOME>/samples/axis2Client/client\_repo/conf/axis2.xml file as well. For an example of using the UDP transport, see [Sample 267: Switching from UDP to HTTP/S.](#)

## HL7 Transport

The HL7 Transport allows you to handle [Health Level 7 International \(HL7\)](#) messages. It is available when you [install the Axis2 Transport HL7 feature](#). This page describes how to enable and configure the HL7 transport in the following sections:

- Enabling the transport
- Configuring the transport
  - Conformance profile
  - Message pre-processing
  - Acknowledgement
    - Configuring application acknowledgement
  - Validating messages
  - Configuring the thread pool
  - Storing messages

WSO2 ESB uses the HAPI parser to provide HL7 support, which currently does not support HL7v3.

### Enabling the transport

You enable the HL7 transport in <ESB\_HOME>/repository/conf/axis2/axis2.xml as follows:

```
<transportReceiver name="hl7"
class="org.wso2.carbon.business.messaging.hl7.transport.HL7TransportListener">
 <parameter name="port">9292</parameter>
</transportReceiver>
<transportSender name="hl7"
class="org.wso2.carbon.business.messaging.hl7.transport.HL7TransportSender">
 <!--parameter name="non-blocking">true</parameter-->
</transportSender>
...
<messageFormatters>
 <messageFormatter contentType="application/edi-hl7"
class="org.wso2.carbon.business.messaging.hl7.message.HL7MessageFormatter" />
...
</messageFormatters>
...
<messageBuilders>
 <messageBuilder contentType="application/edi-hl7"
class="org.wso2.carbon.business.messaging.hl7.message.HL7MessageBuilder" />
</messageBuilders>
```

### Configuring the transport

When [creating an HL7 proxy service](#), you can optionally configure the following behavior of the HL7 transport.

#### **Conformance profile**

Add the parameter "transport.hl7.ConformanceProfilePath" in the proxy service to point to a URL where the HL7 conformance profile (XML file) can be found.

#### **Message pre-processing**

Add the parameter "transport.hl7.MessagePreprocessorClass" in the proxy service to point to an implementation class of the interface "org.wso2.carbon.business.messaging.hl7.common.HL7MessagePreprocessor", which is used to process raw HL7 messages before parsing them so that potential errors in the messages can be rectified using the transport.

### Acknowledgement

You can enable or disable automatic message acknowledgement. When automatic message acknowledgement is enabled, an ACK is immediately sent back to the client after receiving a message. When it is disabled, the user is given control to send back an ACK/NACK message from an ESB sequence after any message validations or related tasks.

When using a transport such as HTTP, to create an ACK/NACK message from an HL7 message in the flow, specify an axis2 scope message context property "HL7\_GENERATE\_ACK" and set its value to true. This ensures that an ACK/NACK message is created automatically when a message is sent (using the HL7 formatter). By default, an ACK message is created. If a NACK message is required instead, use the message context properties "HL7\_RESULT\_MODE" and "HL7\_NACK\_MESSAGE" as described below.

In the proxy service, add the following parameters to enable or disable auto-acknowledgement and validation:

```
<proxy>...
 <parameter name="transport.hl7.AutoAck">true|false</parameter> <!-- default is true
-->
</proxy>
```

When 'AutoAck' is false, you can set the following properties inside an ESB sequence.

```
<property name="HL7_RESULT_MODE" value="ACK|NACK" scope="axis2" /> <!-- notice the
properties should be in axis2 scope -->
```

When the result mode is 'NACK', you can use the following property to provide a custom description of the error message.

```
<property name="HL7_NACK_MESSAGE" value="<ERROR MESSAGE>" scope="axis2" />
```

You can use the property "HL7\_RAW\_MESSAGE" in the axis2 scope to retrieve the original raw EDI format HL7 message in an ESB sequence. The user doesn't have to convert from XML to EDI again, so this usage may be particularly helpful inside a custom mediator.

To control the encoding type of incoming messages, set the Java system property "ca.uhn.hl7v2.llp.charset". Configuring application acknowledgement

In general, we don't wait for the back-end application's response before sending an "accept-acknowledgement" message to the client. If you do want to wait for the application's response before sending the message, define the following property in the InSequence:

```
<property name="HL7_APPLICATION_ACK" value="true" scope="axis2" />
```

In this case, the request thread will wait until the back-end application returns the response before sending the "accept-acknowledgement" message to the client. You can configure how long request threads wait for the application's response by configuring the time-out in milliseconds at the transport level:

```
<transportReceiver name="hl7"
class="org.wso2.carbon.business.messaging.hl7.transport.HL7TransportListener">
 <parameter name="transport.hl7.TimeOut">1000</parameter>
</transportReceiver>
```

For more information on configuring the proxy service for application acknowledgement, see [Application acknowledgement in Creating an HL7 Proxy Service](#).

### **Validating messages**

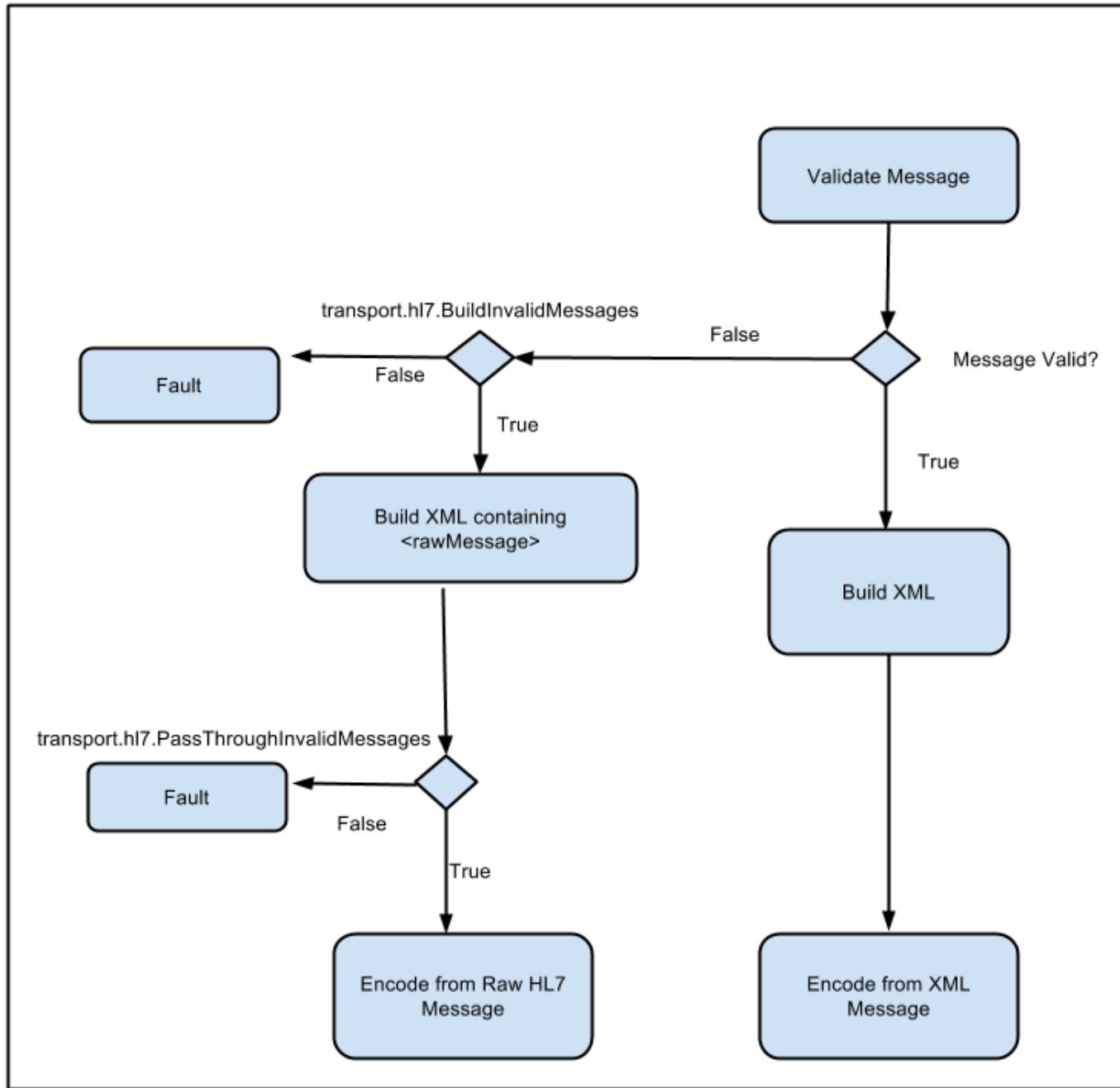
By default, the HL7 transport validates messages before building their XML representation. You configure validation with the following parameter in the proxy service:

```
<proxy>...
 <parameter name="transport.hl7.ValidateMessage">true|false</parameter> <!-- default
 is true -->
</proxy>
```

When `transport.hl7.ValidateMessage` is set to false, you can set the following parameters to handle invalid messages:

- `transport.hl7.BuildInvalidMessages`: when set to true, builds a SOAP envelope with the contents of the raw HL7 message inside the `<rawMessage>` element.
- `transport.hl7.PassThroughInvalidMessages`: when `BuildInvalidMessages` is set to true, you use this parameter to specify whether to pass this message through (true) or to throw a fault (false).

The following diagram illustrates these flows.



### Configuring the thread pool

The HL7 transport uses a thread pool to manage connections. A larger thread pool provides greater performance, because the transport can process more messages simultaneously, but it also uses more memory. You can add the following properties to the proxy service to configure the thread pool to suit your environment:

- `transport.hl7.corePoolSize`: the core number of threads in the pool. Default is 10.
- `transport.hl7.maxPoolSize`: the maximum number of threads that can be in the pool. Default is 20.
- `transport.hl7.idleThreadKeepAlive`: the time in milliseconds to keep idle threads alive before releasing them. Default is 10000 (10 seconds).

### Storing messages

You can use the HL7 message store to automatically store HL7 messages, allowing you to audit and replay messages as needed. The HL7 store is a custom message store implementation on top of Open JPA. To use the

message store, you take the following steps:

1. Create an empty database in your RDBMS, and then create a message store configuration that points to that database.
2. Create a sequence that points to that message store configuration.

For example:

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <proxy name="HL7Store" startOnLoad="true" trace="disable" transports="hl7">
 <description/>
 <target>
 <inSequence>
 <property name="HL7_RESULT_MODE" value="ACK" scope="axis2"/>
 <log level="full"/>
 <property name="messageType" value="application/edi-hl7" scope="axis2"/>
 <clone>
 <target sequence="StoreSequence"/>
 <target sequence="SendSequence"/>
 </clone>
 </inSequence>
 </target>
 <parameter name="transport.hl7.AutoAck">false</parameter>
 <parameter name="transport.hl7.Port">55557</parameter>
 <parameter name="transport.hl7.ValidateMessage">false</parameter>
 </proxy>

 <sequence name="StoreSequence">
 <property name="OUT_ONLY" value="true"/>
 <store messageStore="HL7StoreJPA"/>
 </sequence>

 <sequence name="SendSequence">
 <in>
 <send>
 <endpoint>
 <address uri="hl7://localhost:9988"/>
 </endpoint>
 </send>
 </in>
 <out>
 <log level="full"/>
 <drop/>
 </out>
 </sequence>

 <messageStore class="org.wso2.carbon.business.messaging.hl7.store.jpa.JPASTore"
 name="HL7StoreJPA">
 <parameter
name="openjpa.ConnectionDriverName">org.apache.commons.dbcp.BasicDataSource</parameter>
 <parameter
name="openjpa.ConnectionProperties">DriverClassName=com.mysql.jdbc.Driver,
Url=jdbc:mysql://localhost/hl7storejpa, MaxActive=100, MaxWait=10000,
TestOnBorrow=true, Username=root, Password=root</parameter>
 <parameter name="openjpa.jdbc.DBDictionary">blobTypeName=LONGBLOB</parameter>
 </messageStore>
</definitions>

```

In this configuration, when the HL7 proxy service runs, an HL7 service will start listening on the port defined in the `transport.hl7.Port` service parameter. When an HL7 message arrives, the proxy will send an ACK back to the

client as specified in the HL7\_RESULT\_MODE property. The Clone mediator is used inside the proxy to replicate the message into the Send and Store sequences, where the message is sent to the specified endpoint and is also stored in the message store HL7StoreJPA.

The HL7StoreJPA message store is a custom message store implemented in `org.wso2.carbon.business.messaging.hl7.store.jpa.JPAStore`. It takes [OpenJPA properties](#) as parameters. In this example, the `openjpa.ConnectionProperties` and `openjpa.ConnectionDriverName` properties are used to create an Apache DBCP pooled connection set to a MySQL database. You will need to create the database specified in the connection properties and provide the database authentication details matching your database. You may also need to place the JDBC drivers for your database into `<ESB_HOME>/repository/components/lib`.

You can view the messages in this message store using the HL7 Console UI. You can search for messages on the unique message UUID or HL7 specific Control ID. The search field supports the wildcard '%' to allow LIKE queries. The table can also be filtered to search for content within messages.

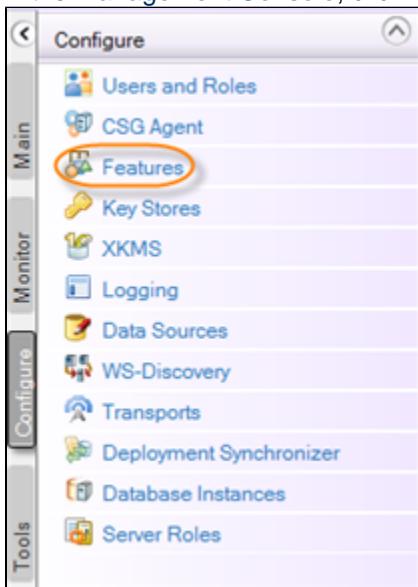
Selected messages can be edited and injected into a proxy service. Reinjecting a message to the same service will result in a new message being stored with a different message UUID.

#### Installing the HL7 Transport

The [HL7 Transport](#) is available as an installable feature in WSO2 ESB.

#### To install the HL7 transport:

1. In the Management Console, click **Configure -> Features**.



2. On the Available Features tab, select the repository <http://wso2.com/projects/carbon/provisioning-wso2-carbon-with-equinox-p2>. If it's not already in the list, click **Add Repository** and add it.
3. Type **HL7** in the Filter by Feature Name box, uncheck the Group Features by Category check box, and then click **Find Features**.
4. From the list of available features, select and install **Business Adaptor HL7**. For more information, see [Installing Features](#). Be sure to restart the ESB after you have completed the installation.
5. Enable the HL7 transport listener and sender as described in [Working with Transports](#).

You can now create an HL7 proxy service using this transport. For information on configuring the HL7 transport, see [HL7 Transport](#).

#### Creating an HL7 Proxy Service

After you [install](#) and [configure the HL7 transport](#), you can create a proxy service that uses this transport to connect to an HL7 server. This proxy service will receive HL7-client connections and send them to the HL7 server. It can also receive XML messages over HTTP/HTTPS and transform them into HL7 before sending them to the server, and it will transform the HL7 responses back into XML.

If you want to try the example configurations on this page, you must have an HL7 client and HL7 back-end application set up and running. To see a sample that illustrates how to create an HL7 client and back-end application, see:  
<https://github.com/wso2/carbon-mediation/tree/v4.6.6/components/business-adaptors/hl7/org.wso2.carbon.business.messaging.hl7.samples/src/main/java/org/wso2/carbon/business/messaging/hl7/samples>

### To create the HL7 proxy service:

1. In the [Management Console](#), create a custom proxy service (see [Adding a Proxy Service](#)).
2. In the transports list, specify https, http, and hl7.
3. Create an Address Endpoint with the URL of the HL7 server host and port, such as: `hl7://localhost:9988`
4. Add the following parameter to the proxy service (required to enable the transport):  
`<parameter name="transport.hl7.Port">9292</parameter>`

For example:

```
<proxy xmlns="http://ws.apache.org/ns/synapse" name="hl7testproxy"
transports="https,http,hl7" statistics="disable" trace="disable" startOnLoad="true">
 <target>
 <inSequence>
 <log level="full" />
 </inSequence>
 <outSequence>
 <log level="full" />
 <send />
 </outSequence>
 <endpoint
name="endpoint_urn_uuid_9CB8D06C91A1E996796270828144799-1418795938">
 <address uri="hl7://localhost:9988" />
 </endpoint>
 </target>
 <parameter name="transport.hl7.Port">9292</parameter>
</proxy>
```

For information on additional configuration you can set on the HL7 transport, see [HL7 Transport](#).  
Accept acknowledgement

If user doesn't want to wait for the back-end service to process the message and only needs acknowledgment from the system that the message was received, you can configure the proxy service to send an ACK/NACK message after the message is received. For example:

```

<proxy name="HL7Proxy" transports="hl7" startOnLoad="true" trace="disable">
 <description/>
 <target>
 <inSequence>
 <property name="HL7_RESULT_MODE" value="ACK" scope="axis2"/>
 <property name="HL7_GENERATE_ACK" value="true" scope="axis2"/>
 <send>
 <endpoint
name="endpoint_urn_uuid_9CB8D06C91A1E996796270828144799-1418795938">
 <address uri="hl7://localhost:9988"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <log level="custom">
 <property name="OUT" value="*****out sequence
proxy2*****" />
 </log>
 <drop/>
 </outSequence>
 </target>
 <parameter name="transport.hl7.AutoAck">false</parameter>
 <parameter name="transport.hl7.ValidateMessage">false</parameter>
</proxy>

```

#### Application acknowledgement

If you want to wait for the application's response before sending the acknowledgment message (see [Configuring application acknowledgement](#)), you add the HL7\_APPLICATION\_ACK property to the inSequence and any additional HL7 properties and transport parameters as needed. For example:

```

<proxy xmlns="http://ws.apache.org/ns/synapse" name="HL7Proxy" transports="hl7"
statistics="disable" trace="disable" startOnLoad="true">
 <target>
 <inSequence>
 <property name="HL7_APPLICATION_ACK" value="true" scope="axis2"/>
 <send>
 <endpoint
name="endpoint_urn_uuid_9CB8D06C91A1E996796270828144799-1418795938">
 <address uri="hl7://localhost:9988"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <property name="HL7_RESULT_MODE" value="NACK" scope="axis2"/>
 <property name="HL7_NACK_MESSAGE" value="error msg" scope="axis2"/>
 <send/>
 </outSequence>
 </target>
 <parameter name="transport.hl7.AutoAck">false</parameter>
 <parameter name="transport.hl7.ValidateMessage">true</parameter>
 <description></description>
</proxy>

```

For additional examples of proxy services that handle HL7 messages, see [Exchanging HL7 Messages with the File](#)

## System.

### Exchanging HL7 Messages with the File System

The following samples demonstrate how to read and write HL7 messages to and from the file system using the [HL7 Transport](#) and [VFS Transport](#):

- Transferring messages from file system to file system
- Transferring messages between HL7 and the file system
- Transferring messages between HL7 and FTP

For more information on creating an HL7 proxy service, see [Creating an HL7 Proxy Service](#).

#### ***Transferring messages from file system to file system***

WSO2 ESB supports transferring messages between HL7 and the file system using the HL7 and VFS transports. To begin, ensure that you have the VFS and HL7 transports enabled by uncommenting the relevant transportReceiver and transportSender elements inside the <ESB\_HOME>/repository/conf/axis2/axis2.xml file. You must also uncomment the relevant builder/formatter pair to enable the ESB to work with the HL7 message format. For more information, see [HL7 Transport](#) and [VFS Transport](#).

Once you have enabled the transports and started the ESB, use the following proxy service configuration to run the sample.

This sample uses the UNIX temporary directory /tmp/ in several VFS parameters; be sure to change them to match a location in your file system.

```
<proxy xmlns="http://ws.apache.org/ns/synapse" name="FileSystemToFileSystem"
transports="vfs">
 <target>
 <inSequence>
 <property name="OUT_ONLY" value="true" scope="default" type="STRING"/>
 <property name="transport.vfs.ReplyFileName"
expression="get-property('transport','FILE_NAME')" scope="transport" type="STRING"/>
 <log level="full"/>
 <send>
 <endpoint>
 <address uri="vfs:file:///tmp/out"/>
 </endpoint>
 </send>
 </inSequence>
 </target>
 <parameter name="transport.PollInterval">5</parameter>
 <parameter name="transport.vfs.FileURI">/tmp/in</parameter>
 <parameter name="transport.vfs.FileNamePattern">.*\.\hl7</parameter>
 <parameter name="transport.vfs.ContentType">application/edi-hl7; charset="iso-8859-15"</parameter>
 <parameter name="transport.hl7.ValidateMessage">false</parameter>
 </proxy>
```

Now, copy the following HL7 message into a text editor and save it as an .hl7 file inside the directory you specified with the `transport.vfs.FileURI` parameter (/tmp/in in the above example).

MSH|^~\&|Abc|Def|Ghi|JKL|20131231000000||ADT^A01|1234567|P|2.6|||NE|NE|CH|

The proxy service is configured to detect .hl7 files in the `transport.vfs.FileURI` directory. We have also configured the VFS content type as the `application/edi-hl7` MIME type with an optional charset encoding. When you save the .hl7 file to that directory, the proxy service invokes the HL7 builders/formatters and builds the

HL7 message into its equivalent XML format. It then prints the XML representation of the message on the WSO2 ESB console and forwards the message to the VFS endpoint '/tmp/out'.

For more information on configuring these transports and general properties you can set, see the following topics:

- [HL7 Transport](#)
- [VFS Transport](#)
- [Properties Reference](#)

### ***Transferring messages between HL7 and the file system***

Now, let's look at how we can send and receive files between an HL7 service and the file system. For this scenario, we will use the [HAPI Test Panel](#) to send messages to WSO2 ESB. As with the previous example, ensure that you have the VFS and HL7 transports and their builders/formatters enabled, and replace the temporary directories shown in the sample with actual directories on your file system.

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="HL7ToFileSystem"
 transports="hl7"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
 <target>
 <inSequence>
 <log level="full"/>
 <property name="HL7_RESULT_MODE" value="ACK" scope="axis2"/>
 <property name="OUT_ONLY" value="true"/>
 <property name="transport.vfs.ReplyFileName"
 expression="fn:concat(get-property('SYSTEM_DATE',
'yyyyMMdd.HHmmssSSS'), '.xml')"
 scope="transport"/>
 <send>
 <endpoint>
 <address uri="vfs:file:///tmp/out"/>
 </endpoint>
 </send>
 </inSequence>
 </target>
 <parameter name="transport.hl7.AutoAck">false</parameter>
 <parameter name="transport.hl7.Port">55555</parameter>
 <parameter name="transport.hl7.ValidateMessage">false</parameter>
 <description/>
</proxy>
```

To invoke this proxy, use the HAPI Test Panel to connect to the HL7 service at the specified port and send a test message. When this proxy service runs, an HL7 service will start listening on the port defined in the `transport.hl7.Port` parameter. When the HL7 message arrives, the proxy will send an ACK back to the client as specified in the `HL7_RESULT_MODE` property. The HL7 message is then processed through the ESB and sent to the VFS endpoint, which will save the HL7 message in '/tmp/out'.

### ***Transferring messages between HL7 and FTP***

The following configuration is similar to the previous example, but it illustrates how to process files between an HL7 endpoint and files accessed through FTP. To run this sample, first set up an endpoint by starting a new receiver connection in the HAPI Test Panel on port 9988. You then configure this endpoint in the Send mediator as shown in the following proxy service example. The proxy service will detect .hl7 files in the `transport.vfs.FileURI` directory and send them to the HL7 endpoint.

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="SFTPToHL7"
 transports="vfs"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
 <target>
 <inSequence>
 <property name="OUT_ONLY" value="true" scope="default" type="STRING"/>
 <log level="full"/>
 <send>
 <endpoint>
 <address uri="hl7://localhost:9988"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <drop/>
 </outSequence>
 </target>
 <parameter name="transport.vfs.ReconnectTimeout">2</parameter>
 <parameter name="transport.vfs.ActionAfterProcess">MOVE</parameter>
 <parameter name="transport.PollInterval">5</parameter>
 <parameter name="transport.hl7.AutoAck">false</parameter>
 <parameter
 name="transport.vfs.MoveAfterProcess">vfs:sftp://user:pass@localhost/vfs/out</parameter>
 <parameter
 name="transport.vfs.FileURI">vfs:sftp://user:pass@localhost/vfs/in</parameter>
 <parameter
 name="transport.vfs.MoveAfterFailure">vfs:sftp://user:pass@localhost/vfs/failed</parameter>
 <parameter name="transport.vfs.FileNamePattern">.*\..hl7</parameter>
 <parameter
 name="transport.vfs.ContentType">application/edi-hl7; charset="iso-8859-15"</parameter>
 <parameter name="transport.vfs.ActionAfterFailure">MOVE</parameter>
 <parameter name="transport.hl7.ValidateMessage">false</parameter>
 <description/>
 </proxy>

```

Once you start the endpoint and the ESB, if you place an HL7 message in the `transport.vfs.FileURI` directory , you will be able to see the message passed to the HL7 endpoint in the HAPI Test Panel.

## Multi-HTTPS Transport

The **Multi-HTTPS transport** is similar to the [HTTPS-NIO transport](#), but it allows you to have different SSL profiles with separate truststores and keystores for different hosts using the same ESB. WSO2 ESB uses truststores and a keystores for SSL protocol implementation. The ESB can listen to different host IPs and ports for incoming HTTPS connections, and each IP/Port will have a separate SSL profile configured.

---

## Enabling the transport

The following `transport receiver` and `transport sender` classes should be included in the ESB configuration in order to enable the Multi-HTTPS transport.

### **Transport receiver**

The receiver class can either be `org.apache.synapse.transport.nhttp.HttpCoreNIOMultiSSLListener` or `org.apache.synapse.transport.passthru.PassThroughHttpMultiSSLListener`.

You can enable the Multi-HTTPS transport receiver by adding the following configuration in the `<ESB_HOME>/repository/conf/Axis2/axis2.xml` file under the **Transport Ins (Listeners)** section:

```

<transportReceiver name="multi-https"
class="org.apache.synapse.transport.nhttp.HttpCoreNIOMultiSSLListener">
 <parameter name="port">8343</parameter>
 <parameter name="non-blocking">true</parameter>
 <parameter name="SSLProfiles">
 <profile>
 <bindAddress>192.168.1.2</bindAddress>
 <KeyStore>
 <Location>/path/to/testhost1.p12</Location>
 <Type>PKCS12</Type>
 <Password>test</Password>
 <KeyPassword>test</KeyPassword>
 </KeyStore>
 </profile>
 <profile>
 <bindAddress>192.168.1.3</bindAddress>
 <KeyStore>
 <Location>/path/to/testhost2.p12</Location>
 <Type>PKCS12</Type>
 <Password>test</Password>
 <KeyPassword>test</KeyPassword>
 </KeyStore>
 </profile>
 <profile>
 <bindAddress>192.168.1.4</bindAddress>
 <KeyStore>
 <Location>/path/to/testhost3.p12</Location>
 <Type>PKCS12</Type>
 <Password>test</Password>
 <KeyPassword>test</KeyPassword>
 </KeyStore>
 <TrustStore>
 <Location>/path/to/testtrust.jks</Location>
 <Type>JKS</Type>
 <Password>nopassword</Password>
 </TrustStore>
 <SSLVerifyClient>require</SSLVerifyClient>
 </profile>
 </parameter>
</transportReceiver>
```

### **Transport sender**

The sender class can either be `org.apache.synapse.transport.nhttp.HttpCoreNIOSSLSender` or `org.apache.synapse.transport.passthru.PassThroughHttpSSLSender`.

You can enable the Multi-HTTPS transport sender by adding the following configuration in the `<ESB_HOME>/repository/conf/Axis2/axis2.xml` file under the **Transport Outs (Senders)** section:

```

<transportSender name="https"
class="org.apache.synapse.transport.nhttp.HttpCoreNIOSSL Sender">
 <parameter name="non-blocking" locked="false">true</parameter>
 <parameter name="customSSLProfiles">
 <profile>
 <servers>localhost:8244</servers>
 <KeyStore>
 <Location>repository/resources/security/esb.jks</Location>
 <Type>JKS</Type>
 <Password>123456</Password>
 <KeyPassword>123456</KeyPassword>
 </KeyStore>
 <TrustStore>
 <Location>repository/resources/security/esbtruststore.jks</Location>
 <Type>JKS</Type>
 <Password>123456</Password>
 </TrustStore>
 </profile>
 </parameter>
 <parameter name="keystore" locked="false">
 <KeyStore>
 <Location>repository/resources/security/wso2carbon.jks</Location>
 <Type>JKS</Type>
 <Password>wso2carbon</Password>
 <KeyPassword>wso2carbon</KeyPassword>
 </KeyStore>
 </parameter>
 <parameter name="truststore" locked="false">
 <TrustStore>
 <Location>repository/resources/security/client-truststore.jks</Location>
 <Type>JKS</Type>
 <Password>wso2carbon</Password>
 </TrustStore>
 </parameter>
 <parameter name="HostnameVerifier">AllowAll</parameter>
</transportSender>

```

## Synchronizing the profiles in a cluster

If you are running in a clustered environment and want your SSL profiles to be synchronised across the cluster nodes, you can move the `SSLProfiles` parameter from `axis2.xml` to `<ESB_HOME>/repository/deployment/server/multi_ssl_profiles.xml`. Then you can add the `SSLProfilesConfigPath` parameter to the Multi-HTTPS transport receiver configuration in the `axis2.xml` file and point to the new destination of the configuration.

For example, the Multi-HTTPS transport configuration in the `axis2.xml` file will now look as follows:

```

<transportReceiver name="multi-https"
class="org.apache.synapse.transport.nhttp.HttpCoreNIOMultiSSLListener">
 <parameter name="port">8343</parameter>
 <parameter name="non-blocking">true</parameter>
 <parameter name="SSLProfilesConfigPath">
 <filePath>/repository/deployment/server/multi_ssl_profiles.xml</filePath>
 </parameter>
</transportReceiver>

```

To synchronise this configuration between two ESB nodes, you must enable ESB clustering and the SVN-Based Deployment Synchronizer. For more information, see [Introduction to Deployment Synchronizer](#).

The `<ESB_HOME>/repository/deployments/server` directory will then be synchronized on the ESB nodes when the nodes are run in a clustered environment. If you change the `multi_ssl_profiles.xml` file, you must manually reload it into each ESB node by invoking the `reloadSSLProfileConfig` in the `org.apache.synapse.MultisSLProfileReload` MBean in JConsole. For more information, see [Monitoring the ESB](#).

## Dynamic SSL profiles

In addition to updating `axis2.xml` with the SSL profile configurations, you can dynamically load the SSL profiles at runtime using a periodic schedule or JMX invocation. Now instead of reloading the entire `axis2.xml` at runtime, you can reload the new configuration files that contain only the custom profile information for the sender and receiver.

### **Enabling dynamic SSL profiles**

The following configuration changes should be done in the Multi-HTTPS transport receiver and sender.

#### **Do the following changes in the Multi-HTTPS transport receiver:**

- Edit the `<ESB_HOME>/repository/conf/Axis2/axis2.xml` file and add the `dynamicSSLProfilesConfig` parameter as follows to the multi-https transport listener:

```

<transportReceiver name="multi-https"
class="org.apache.synapse.transport.nhttp.HttpCoreNIOMultiSSLListener">
 <parameter name="port">8343</parameter>
 <parameter name="non-blocking">true</parameter>

 <parameter name="dynamicSSLProfilesConfig">
 <filePath>repository/conf/sslprofiles/listenerprofiles.xml</filePath>
 <fileReadInterval>3600000</fileReadInterval>
 </parameter>

</transportReceiver>

```

- Create the `listenerprofiles.xml` file with the following configuration in the `<ESB_HOME>/repository/conf/sslprofiles` directory:

### **Note:**

You can configure the file path for the `listenerprofiles.xml` file as required.

### Configuration for listenerprofiles.xml

```
<parameter name="SSLProfiles">
<profile>
 <bindAddress>192.168.0.123</bindAddress>
 <KeyStore>
 <Location>repository/resources/security/esb.jks</Location>
 <Type>JKS</Type>
 <Password>123456</Password>
 <KeyPassword>123456</KeyPassword>
 </KeyStore>
 <TrustStore>
 <Location>repository/resources/security/esbtruststore.jks</Location>
 <Type>JKS</Type>
 <Password>123456</Password>
 </TrustStore>
 <SSLVerifyClient>require</SSLVerifyClient>
 </profile>
</parameter>
```

The SSL profile will be applied to each request that is received at the IP specified within the `<bindAddress>` element.

#### Do the following changes in the Multi-HTTPS transport sender:

- Edit the `<ESB_HOME>/repository/conf/Axis2/axis2.xml` file and add the `dynamicSSLProfilesConfig` parameter as follows:

```
<transportSender name="https"
class="org.apache.synapse.transport.nhttp.HttpCoreNIOSSLSSender">

 <parameter name="dynamicSSLProfilesConfig">
 <filePath>repository/conf/sslprofiles/senderprofiles.xml</filePath>
 <fileReadInterval>3600000</fileReadInterval>
 </parameter>

</transportSender>
```

- Create the `senderprofiles.xml` file with the following configuration in the `<ESB_HOME>/repository/conf/sslprofiles` directory:

#### Note:

You can configure the file path for the `senderprofiles.xml` file as required.

### Configuration for senderprofiles.xml

```
<parameter name="customSSLProfiles">
<profile>
<servers>localhost:8244,192.168.1.234:8245</servers>
<KeyStore>
<Location>repository/resources/security/esb.jks</Location>
<Type>JKS</Type>
<Password>123456</Password>
<KeyPassword>123456</KeyPassword>
</KeyStore>
<TrustStore>
<Location>repository/resources/security/esbtruststore.jks</Location>
<Type>JKS</Type>
<Password>123456</Password>
</TrustStore>
</profile>
</parameter>
```

The SSL profile will be applied to each request that is sent to the destination server specified within the `<servers>` element as IP:Port combination.

### ***Loading SSL profiles dynamically at runtime***

You can either use a periodic schedule or a JMX invocation to apply custom profiles at runtime. The following section describes the two options in detail:

- Periodic schedule - If you use this option, the ESB will automatically check updates of the file content and apply the custom profiles based on the value specified in the `fileReadInterval` parameter. For example, if you have set the `fileReadInterval` as 1 hour, the ESB will automatically check updates of the file content and apply the custom profile every 1 hour.
- JMX Invocation - If you use this option, custom profiles will be applied dynamically by invoking the `notifyFileUpdate` method in the respective sender/listener MBean under the `ListenerSSLProfileReload` or `SenderSSLProfileReload` group in JConsole.

The following table provides information on the parameters that you can set when you enable dynamic SSL profiles:

Parameter Name	Description	Default Value
<code>filePath</code>	The relative/absolute file path of the custom SSL profile configuration XML file.	-
<code>fileReadInterval</code>	The time interval (in milliseconds) in which configuration updates will be loaded and applied at runtime. This value should be greater than 1 minute.	3600000

### **MQ Telemetry Transport**

MQ Telemetry Transport (MQTT) is a simple and lightweight network protocol for device communication. This is an easy to implement protocol that is based on the principle of publish/subscribe. These characteristics make MQTT ideal for use in constrained environments.

For example,

- When the network is expensive, has low bandwidth or is unreliable.
- When running on an embedded device with limited processor or memory resources.

The MQTT transport implementation requires an MQTT server instance to be able to send and receive messages. The recommended MQTT server is the Mosquitto message broker.

Configuration parameters for the MQTT receiver and sender are XML fragments that represent MQTT connection factories.

Following is a sample MQTT connection factory configuration that consists of four connection factory parameters:

```
<parameter locked="false" name="mqttConFactory">
 <parameter locked="false"
name="mqtt.server.host.name">localhost</parameter>
 <parameter locked="false" name="mqtt.server.port">1883</parameter>
 <parameter locked="false"
name="mqtt.client.id">esb.test.listener</parameter>
 <parameter locked="false" name="mqtt.topic.name">esb.test2</parameter>
</parameter>
```

## MQTT connection factory parameters

Following are details on the MQTT parameters that you can set:

Parameter Name	Description	Required	Possible Values
mqtt.server.host.name	The name of the host.	Yes	A valid host name
mqtt.server.port	The port ID.	Yes	1883,1885
mqtt.client.id	The client ID.	Yes	A valid client ID
mqtt.topic.name	The name of the topic	Yes	A valid topic name
mqtt.subscription.qos	The QoS value.	No	0,1,2
mqtt.session.clean	Whether session clean should be enabled or not.	No	true, false
mqtt.ssl.enable	Whether ssl should be enabled or not.	No	true , false
mqtt.subscription.username	The username for the subscription.	No	A valid user name
mqtt.subscription.password	The password for the subscription.	No	A valid password
mqtt.temporary.store.directory	The path of the directory to be used as the persistent data store for quality of service purposes.	No	A valid local path. The default value is the ESB temp path.
mqtt.blocking.sender	Whether blocking sender should be enabled or not.	No	true , false

mqtt.content.type	The content type.	No	A valid content type. the default content type is text/plain.
mqtt.message.retained	Whether the messaging engine should retain a published message or not. This parameter can be used only in the transport sender.	No	true, false The default value is false.

For a sample that demonstrates how Axis2 publishes a message on a particular topic, and how an MQTT client subscribed to that topic receives it, see [Sample 272: Publishing and Subscribing using WSO2 ESB's MQ Telemetry Transport](#).

## WebSocket Transport

The WSO2 ESB WebSocket transport implementation is based on the [WebSocket protocol](#), and consists of an Axis2 sender implementation for WebSockets and secure WebSockets. This transport supports bidirectional message mediation.

### Enabling the transport

The following transport sender class should be included in the ESB configuration to enable the WebSocket transport.

- org.wso2.carbon.websocket.transport.WebsocketTransportSender

### To enable the WebSocket transport sender

- Edit the <ESB\_HOME>/repository/conf/axis2/axis2.xml file and uncomment the following WebSocket sender configuration:

```
<transportSender name="ws"
class="org.wso2.carbon.websocket.transport.WebsocketTransportSender">
 <parameter name="ws.outflow.dispatch.sequence"
locked="false">outflowDispatchSeq</parameter>
 <parameter name="ws.outflow.dispatch.fault.sequence"
locked="false">outflowFaultSeq</parameter>
</transportSender>
```

### To enable the secure WebSocket transport sender

- Edit the <ESB\_HOME>/repository/conf/axis2/axis2.xml file and uncomment the following secure WebSocket sender configuration:

```

<transportSender name="wss"
class="org.wso2.carbon.websocket.transport.WebsocketTransportSender">
 <parameter name="ws.outflow.dispatch.sequence"
locked="false">outflowDispatchSeq</parameter>
 <parameter name="ws.outflow.dispatch.fault.sequence"
locked="false">outflowFaultSeq</parameter>
 <parameter name="ws.trust.store" locked="false">

<ws.trust.store.location>repository/resources/security/client-truststore.jks</ws.
trust.store.location>
 <ws.trust.store.Password>wso2carbon</ws.trust.store.Password>
</parameter>
</transportSender>

```

## Working with Modules

A **module** is an archive file that bundles a set of classes, related libraries and third party library dependencies.

The folder structure of the module archive file will look as follows:

```

Test.mar
 - META-INF
 - module.xml
TestModule.class
TestHandler.class

```

WSO2 ESB has the following modules deployed by default:

Name	Version	Description
rampart	1.61-wso2v14	This module provides WS-Security and WS-SecureConversation functionalitie s for Axis2, based on the Apache WSS4J, Apache XML-Security and Apache Rahas implementations.
addressing	4.4.0	This is the WS-Addressing implementation on Axis2, supporting the WS-Address ing 1.0 Recommendation, as well as the submission version (2004/08).
relay	4.4.1	Unwraps the binary messages coming from the Message Relay for Admin Services.
rahas	1.61-wso2v14	This module is used to enable STS for service where it adds the RequestSecurit yToken operation to a service that the module is engaged to.

For more information on Axis2 Modules, see [Apache Axis2 Architecture Guide](#).

WSO2 ESB provides a user friendly interface to work with modules. See the following topics for more information on how to work with modules:

- [Adding a Module](#)
- [Engaging Modules](#)
- [Writing an Axis2 Module](#)

### Adding a Module

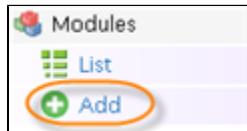
You can add a module to WSO2 ESB if you need to extend its capabilities.

Follow the instructions below to add a Module to the ESB:

1. Sign in. Enter your user name and password to log on to the ESB Management Console.
2. Click on "Main" in the left menu to access the "Manage" menu.



3. In the "Manage" menu, click on "Add" under "Modules."



4. The "Add modules" page appears.

The screenshot shows the 'Add modules' page. At the top, it says 'Add modules'. Below that is a section titled 'Upload new modules (.mar)'. It has a 'Module Archive (.mar)\*' input field, a 'Browse...' button, and a '+' button. At the bottom are 'Upload' and 'Cancel' buttons.

5. Click "Browse" to select the module archive file (MAR) you want to upload. To learn the detailed information about custom modules, see [Writing an Axis2 Module](#).

The screenshot shows the 'Add modules' page again. The 'Upload new modules (.mar)' section is visible. The 'Browse...' button next to the 'Module Archive (.mar)\*' input field is highlighted with an orange circle.

**Tip**

Use the plus sign button to upload more than one module.

### Add modules

#### Upload new modules (.mar)

Module Archive (.mar)\*

C:\Documents and Settings\Alina\Desktop\moduleMy

[Browse...](#)



Module Archive (.mar)\*

C:\Documents and Settings\Alina\Desktop\my\_mediator

[Browse...](#)

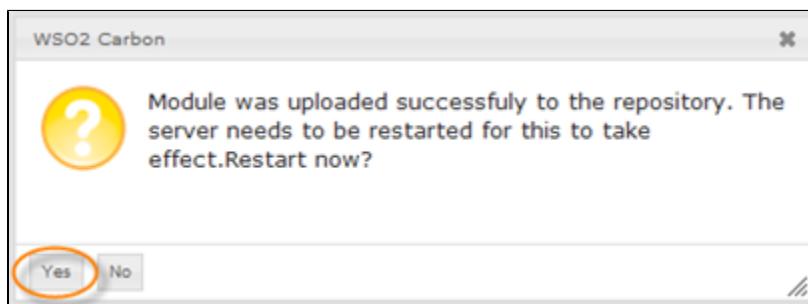


[Upload](#) [Cancel](#)

6. Click "Upload."

[Upload](#) [Cancel](#)

7. If the file was uploaded successfully, a message will appear prompting you to restart the server to activate the module. Click "OK."



Once the server has been restarted, the module will be active and displayed on the "Deployed Modules" page.

### Engaging Modules

A module is an archive file that bundles a set of classes, related libraries and third party library dependencies. After uploading a module to the system, you can engage the module to make it active. You can engage a module on a global level as well as at service level.

#### To engage a module at service level

Add the engagedModules service parameter to the proxy service and specify the modules you want to engage as comma separated values. For example, if you need to engage the rampart module and the sample-logging module, add the following parameter to the proxy service:

```
<parameter name="engagedModules">rampart, sample-logging</parameter>
```

Following is a sample proxy service where the engagedModules parameter is specified to engage the sample-logging module:

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="Pass"
 transports="http,https"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
 <target>
 <outSequence>
 <send/>
 </outSequence>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </target>
 <parameter name="engagedModules">sample-logging</parameter>
 <description/>
 </proxy>

```

### To engage a module on a global level

1. In the <ESB\_HOME>/repository/conf/axis2/axis2.xml file, under the **Global Engaged Modules** section, add a <module ref="modulename" /> entry for every module that you want to engage on a global level.
2. Restart the server.

For example, to engage the rampart module on a global level, add the following entry and restart the server.

```
<module ref="rampart" />
```

### Note

The WS-Addressing module is always engaged on a global level by default.

### Writing an Axis2 Module

Axis2 provides extended support for modules (see the [Architecture Guide](#) for more details about modules in Axis2).

The following steps show the actions that need to be performed to deploy a custom module for a given Web service:

1. [Create the Module Implementation](#)
2. [Create the Handlers](#)
3. [Create the module.xml](#)
4. [Modify the axis2.xml](#) (if you need custom phases)
5. [Modify the services.xml](#) to engage modules at the deployment time
6. [Package in a MAR \(Module Archive\)](#)
7. [Deploy the module in Axis2](#)

There is an example of a simple logging module below. This module contains one handler that just logs the message that is passed through it. Axis2 uses MAR (Module Archive) to deploy modules in Axis2.

### **Create the Module Implementation**

LoggingModule is the implementation class of the Axis2 module. Axis2 modules should implement the org.apache.axis2.modules.Module interface with the following methods:

```
public void init(ConfigurationContext configContext, AxisModule module) throws
AxisFault;//Initialize the module
public void shutdown(ConfigurationContext configurationContext) throws AxisFault;//End
of module processing
public void engageNotify(AxisDescription axisDescription) throws AxisFault;
public void applyPolicy(Policy policy, AxisDescription axisDescription) throws
AxisFault ;
public boolean canSupportAssertion(Assertion assertion) ;
```

- **init(), shutdown(), engageNotify()** - The methods that can be used to control the module initialization and the termination,
- **applyPolicy(), canSupportAssertion()** - The methods that are used to perform policy related operations.

With the input parameter AxisConfiguration, the user is provided with the complete configuration hierarchy. This can be used to fine-tune the module behavior by the module writers.

For a simple logging service (as in the example), it is possible to keep these methods blank in the implementation class.

### **Create the Handlers**

In the example below, a LogHandler is created.

A module in "Axis2" can contain one or more handlers that perform various SOAP header processing at different phases. To write a handler one should implement org.apache.axis2.engine.Handler. But for convenience org.apache.axis2.handlers.AbstractHandler provides an abstract implementation of the Handler interface.

For the logging module a handler is written with the following methods:

- **public void invoke(MessageContext ctx);** - Is the method that is called by the "Axis2" engine when the control is passed to the handler.
- **public void revoke(MessageContext ctx);** - Is called when the handlers are revoked by the "Axis2" engine.

```

public class LogHandler extends AbstractHandler implements Handler {
 private static final Log log = LogFactory.getLog(LogHandler.class);
 private String name;

 public String getName() {
 return name;
 }

 public InvocationResponse invoke(MessageContext msgContext) throws AxisFault {
 log.info(msgContext.getEnvelope().toString());
 return InvocationResponse.CONTINUE;
 }

 public void revoke(MessageContext msgContext) {
 log.info(msgContext.getEnvelope().toString());
 }

 public void setName(String name) {
 this.name = name;
 }
}

```

#### **Create the module.xml**

module.xml contains the deployment configurations for a particular module. It contains details such as the implementation class of the module (in this example it is the LoggingModule class and various handlers that will run in different phases). The module.xml for the logging module will be as follows:

```

<module name="logging" class="userguide.loggingmodule.LoggingModule">
 <InFlow>
 <handler name="InFlowLogHandler" class="userguide.loggingmodule.LogHandler">
 <order phase="loggingPhase" />
 </handler>
 </InFlow>

 <OutFlow>
 <handler name="OutFlowLogHandler" class="userguide.loggingmodule.LogHandler">
 <order phase="loggingPhase" />
 </handler>
 </OutFlow>

 <OutFaultFlow>
 <handler name="FaultOutFlowLogHandler"
class="userguide.loggingmodule.LogHandler">
 <order phase="loggingPhase" />
 </handler>
 </OutFaultFlow>

 <InFaultFlow>
 <handler name="FaultInFlowLogHandler"
class="userguide.loggingmodule.LogHandler">
 <order phase="loggingPhase" />
 </handler>
 </InFaultFlow>
</module>

```

There are four flows defined in the `module.xml`:

- **InFlow** - Represents the handler chain that will run when a message is coming in.
- **OutFlow** - Represents the handler chain that will run when the message is going out.
- **OutFaultFlow** - Represents the handler chain that will run when there is a fault, and the fault is going out.
- **InFaultFlow** - Represents the handler chain that will run when there is a fault, and the fault is coming in.

The following set of tags describes the name of the handler, handler class, and the phase in which this handler is going to run.

- **handler**
  - **name** - Is the name given for the particular instance of this handler class.
  - **class** - Is the actual implementation class for this handler. Since we are writing a logging handler, we can reuse the same handler in all these phases. However, this may not be the same for all the modules.
- **order**
  - **phase** - Describes the phase in which this handler runs.

```

<handler name="InFlowLogHandler" class="userguide.loggingmodule.LogHandler">
<order phase="loggingPhase" />
</handler>

```

To learn more about Phase rules, check out the article [Axis2 Execution Framework](#).

#### **Modify the `axis2.xml`**

In this handler the loggingPhase is defined by the module writer. It is not a predefined handler phase, hence the module writer should introduce it to the axis2.xml (not the services.xml) so that the Axis2 engine knows where to place the handler in different flows (inFlow, outFlow, etc.).

The following XML lines show the respective changes made to the axis2.xml in order to deploy the logging module in the "Axis2" engine. This is an extract of the phase section of axis2.xml.

```

<!-- ===== -->
<!-- Phases -->
<!-- ===== -->

<phaseOrder type="inflow">
 <!-- System pre defined phases -->
 <phase name="TransportIn"/>
 <phase name="PreDispatch"/>
 <phase name="Dispatch" class="org.apache.axis2.engine.DispatchPhase">
 <handler name="AddressingBasedDispatcher"
 class="org.apache.axis2.dispatchers.AddressingBasedDispatcher">
 <order phase="Dispatch"/>
 </handler>

 <handler name="RequestURIBasedDispatcher"
 class="org.apache.axis2.dispatchers.RequestURIBasedDispatcher">
 <order phase="Dispatch"/>
 </handler>

 <handler name="SOAPActionBasedDispatcher"
 class="org.apache.axis2.dispatchers.SOAPActionBasedDispatcher">
 <order phase="Dispatch"/>
 </handler>

 <handler name="SOAPMessageBodyBasedDispatcher"
 class="org.apache.axis2.dispatchers.SOAPMessageBodyBasedDispatcher">
 <order phase="Dispatch"/>
 </handler>

 <!-- System pre defined phases -->
 <!-- After Postdispatch phase module author or service author can add any
phase he wants -->
 <phase name="OperationInPhase"/>
 <phase name="loggingPhase"/>
 </phaseOrder>
 <phaseOrder type="outflow">
 <!-- user can add his own phases to this area -->
 <phase name="OperationOutPhase"/>
 <phase name="loggingPhase"/>
 <!--system predefined phases-->
 <!--these phases will run irrespective of the service-->
 <phase name="PolicyDetermination"/>
 <phase name="MessageOut"/>
 </phaseOrder>
 <phaseOrder type="INfaultflow">

```

```
<!-- user can add his own phases to this area -->
<phase name="OperationInFaultPhase"/>
<phase name="loggingPhase"/>
</phaseOrder>
<phaseOrder type="Outfaultflow">
<!-- user can add his own phases to this area -->
<phase name="OperationOutFaultPhase"/>
<phase name="loggingPhase"/>
<phase name="PolicyDetermination"/>
```

```
<phase name="MessageOut" />
</phaseOrder>
```

The custom phase `loggingPhase` is placed in all the flows, hence that phase will be called in all the message flows in the engine. Since the module is associated with this phase, the `LogHandler` inside the module will now be executed in this phase.

#### ***Modify the services.xml***

Up to this point, it was created the required classes and configuration descriptions for the logging module and the required phases.

Next step is to "engage" (use) this module in one of the services, for example `MyService`. It is necessary to modify the `services.xml` of `MyService` in order to engage this module.

The code for the service can be found here:

- Axis2\_HOME/samples/userguide/src/userguide/example2

```
<service name="MyServiceWithModule">

 <description>
 This is a sample Web service with a logging module engaged.
 </description>

 <module ref="logging"/>

 <parameter name="ServiceClass"
locked="xsd:false">userguide.example2.MyService</parameter>
 <operation name="echo">
 <messageReceiver class="org.apache.axis2.receivers.RawXMLINOutMessageReceiver" />
 </operation>
 <operation name="ping">
 <messageReceiver class="org.apache.axis2.receivers.RawXMLINOutMessageReceiver" />
 </operation>
 </service>
```

In this example, the service name was changed. In addition the line `<module ref="logging"/>` was added to `services.xml`. This informs the "Axis2" engine that the module logging should be engaged for this service. The handler inside the module will be executed in their respective phases as described by the `module.xml`.

#### ***Package in a MAR (Module Archive)***

Before deploying the module, it is necessary to create the MAR file for this module. This can be done using the JAR command and then renaming the created JAR file. Else you can find the `logging.mar` that has already been created in the `Axis2_HOME/samples/userguide` directory.

#### ***Deploy the module in Axis2***

Deploying a module in Axis2 requires the user to create a directory with the name "modules" in the `webapps/axis2/WEB-INF` directory of their servlet container, and then copying the MAR file to that directory.

## **WSO2 ESB Analytics**

This section describes how you can use WSO2 ESB analytics to publish information that is related to the processing

carried out in WSO2 ESB to the Analytics Dashboard.

See the following topics for detailed information on how to work with WSO2 ESB analytics.

- Prerequisites to Publish Statistics
- Analyzing WSO2 ESB with the Analytics Dashboard
- Monitoring WSO2 ESB with WSO2 ESB Analytics
- Extending ESB Analytics
- Monitoring WSO2 ESB with WSO2 Analytics in a Multi-tenant Environment

## Prerequisites to Publish Statistics

The following sections cover the prerequisites that should be completed in order to publish information relating to the processing carried out by WSO2 ESB in the Analytics Dashboard of WSO2 ESB Analytics.

- Downloading WSO2 ESB Analytics
- Installing WSO2 ESB Analytics
- Running WSO2 ESB Analytics
- Enabling statistics for the mediation flow
- Configuring ESB Analytics
- Running WSO2 ESB
- Enabling Statistics for ESB artifacts

### ***Downloading WSO2 ESB Analytics***

Follow the instructions below to download the binary distribution of WSO2 ESB Analytics

The binary distribution contains the binary files for both MS Windows, and Linux-based operating systems. You can also download, and [build the source code](#).

1. Go to the [WSO2 ESB home page](#).
2. Under **Downloads**, click **Analytics**.

### ***Installing WSO2 ESB Analytics***

The procedure for installing WSO2 ESB Analytics is the same as that for WSO2 Data Analytics Server. Therefore, for detailed information about supporting applications, installing in different operating systems, starting and running the product etc., see the [Installation Guide in WSO2 Data Analytics Server documentation](#).

### ***Running WSO2 ESB Analytics***

Once WSO2 ESB Analytics is downloaded, you can start its server and access its Management Console. For detailed instructions to run a WSO2 product, see [Running the Product](#).

You may need more than one WSO2 ESB Analytics server in a clustered deployment depending on the load of requests handled by the ESB cluster. The possible deployment patterns for WSO2 ESB Analytics is the same as that for WSO2 Data Analytics Server. For more information on how to run more than one ESB - Analytics server, see [Clustering Data Analytics Server](#).

### ***Enabling statistics for the mediation flow***

Enable mediation statistics and message tracing as required by configuring the `<ESB_HOME>/repository/conf/synapse.properties` file as described below.

In a clustered deployment, the following configurations should be done for each node in the ESB cluster.

Property	Required Value	Purpose
mediation.flow.statistics.enable	true	<p>Setting this property to true enables the collection of the following information being processed by individual mediation components:</p> <ul style="list-style-type: none"> <li>• The time spent on each component</li> <li>• The time spent to process the message</li> <li>• The fault count of a single component</li> </ul>
mediation.flow.statistics.tracer.collect.payloads	true	Setting this property to true enables the collection of the message payload before an artifact is processed by individual mediation components.
mediation.flow.statistics.tracer.collect.properties	true	<p>Setting this property to true enables the collection of the following information being processed by individual mediation components:</p> <ul style="list-style-type: none"> <li>• Message context properties</li> <li>• Message transport-scoped properties</li> </ul>
mediation.flow.statistics.collect.all	true/false	<p>If this property is set to true, statistics are enabled for all the artifacts in the WSO2 ESB server by default.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p>You can enable statistics for specific artifacts by setting the mediation.flow.statistics.collect.property property. This property is also set to true by default.</p> </div> <p>If this property is set to false, statistics tracing needs to be manually enabled for each artifact required as explained in the <a href="#">Statistics for ESB artifacts</a>.</p>

### Configuring ESB Analytics

The required configuration details described below are available by default. Follow this section to understand the Analytics related configurations used in the process and do any modifications if required. In a clustered deployment, the following configurations should be checked for each ESB node in the cluster.

This step involves providing the information required by WSO2 ESB to publish data to the WSO2 ESB Analytics server in order to analyze the data using the Analytics Dashboard. An event publisher is configured with the URL to which WSO2 ESB related information are published as events.

Follow the procedure below to configure WSO2 ESB Analytics.

1. Open the <ESB\_HOME>/repository/deployment/server/eventpublishers/MessageFlowConfigurationPublisher.xml file, and enter relevant configurations as described in the table below. This file is used to enter the information required to publish WSO2 ESB data to the WSO2 ESB Analytics server.

Property	Required Value	Example
username	The username to be used when accessing the WSO2 ESB Analytics server to publish configurations.	admin

password	The password to be used when accessing the WSO2 ESB Analytics server to publish configurations.	admin
receiverURL	The URL of the thrift port to which the ESB configurations should be published. The format of the URL is as follows. tcp://<localhost>:<THRIFT_PORT>	tcp://localhost:7612

2. Open the <ESB\_HOME>/repository/deployment/server/eventpublishers/MessageFlowStatisticsPublisher.xml file, and do the configurations and specified in the table below.

Property	Required Value	Example
username	The username to be used when accessing the WSO2 ESB Analytics server to publish statistics.	admin
password	The password to be used when accessing the WSO2 ESB Analytics server to publish statistics.	admin
receiverURL	The URL of the thrift port to which the ESB statistics should be published. The format of the URL is as follows. tcp://<localhost>:<THRIFT_PORT>	tcp://localhost:7612

### Running WSO2 ESB

The WSO2 ESB server should run simultaneously with the WSO2 ESB Analytics server. For detailed instructions to run a WSO2 product, see [Running the Product](#).

Open the Management Consoles of the two WSO2 products in two separate browsers to avoid signing off from one Management Console when you sign into the other.

### Enabling Statistics for ESB artifacts

You can select the specific proxy services, REST APIs, sequences and endpoints that you want to monitor via WSO2 ESB Analytics by enabling statistics for them. To enable statistics for an individual artifact, you need to click the **Enable Statistics** link for it in the ESB Management Console. See the following sections for detailed information on enabling statistics for the required artifact types.

In a clustered deployment, statistics should be enabled for the required artifacts in all the ESB nodes if you have not enabled [deployment synchronising](#). If deployment synchronising is enabled, you need to enable statistics for the required artifacts only in one node.

For example, you can enable statistics for a proxy service as described below.

1. Start the WSO2 ESB server and log into the Management Console.
2. In the **Main** tab, look for the **Services** section and click **List** under the same section to open the **Deployed Services** page. Then click on the required proxy service as shown below.

## Deployed Services

4 active services. 4 deployed service group(s).

Services								
	echo	axis2	Unsecured	WSDL1.1	WSDL2.0	Try this service	Download	
<input type="checkbox"/>	StockQuoteProxy	proxy	Unsecured	WSDL1.1	WSDL2.0	Try this service	Download	<a href="#">Design View</a>
<input type="checkbox"/>	Version	axis2	Unsecured	WSDL1.1	WSDL2.0	Try this service	Download	<a href="#">Source View</a>
<input type="checkbox"/>	wso2carbon-sts	sts	Unsecured	WSDL1.1	WSDL2.0			

Select all in this page | Select none      [Delete](#)

3. Click **Enable Statistics** and **Enable Tracing** links. Click **OK** in the information message that appears to confirm that statistics/tracing is enabled.

### Service Dashboard (StockQuoteProxy)

Service Details		Client Operations	
Service Name	StockQuoteProxy	<a href="#">Try this service</a>	
Service Description	No service description found	<a href="#">Generate Axis2 Client</a>	
Service Group Name	StockQuoteProxy	<a href="#">WSDL1.1</a>	<a href="#">WSDL2.0</a>
Deployment Scope	request	Endpoints	
Service Type	proxy	<a href="http://Rukshanis-MacBook-Air.local:8282/services/StockQuoteProxy">http://Rukshanis-MacBook-Air.local:8282/services/StockQuoteProxy</a> <a href="https://Rukshanis-MacBook-Air.local:8245/services/StockQuoteProxy">https://Rukshanis-MacBook-Air.local:8245/services/StockQuoteProxy</a>	
Operations			
<input checked="" type="checkbox"/> Active <a href="#">Deactivate</a>			
Quality of Service Configuration		Statistics	
<b>Specific Configuration</b> <a href="#">Edit</a> <a href="#">Enable Statistics</a> <a href="#">Redeploy</a> <a href="#">Enable Tracing</a>		Request Count 0 Response Count 0 Fault Count 0 Maximum Response Time 0 ms Minimum Response Time 0 ms Average Response Time 0.0 ms	

The following table specifies the path to the **Enable Statistics** link for each ESB artifact type that can be analyzed using the Analytics Dashboard.

ESB Artifact Type	Path
Proxy Service	<b>Main tab =&gt; Services =&gt; List =&gt; Deployed Services page =&gt; &lt;Proxy_Service_Name&gt;</b>
REST API	<b>Main tab =&gt; APIs =&gt; Deployed APIs page</b>
Sequence	<b>Main tab =&gt; Sequences =&gt; Mediation Sequences page</b>
Endpoint	<b>Main tab =&gt; Endpoints =&gt; Manage Endpoints page</b>

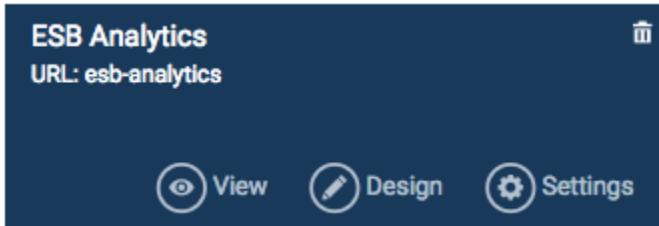
## Analyzing WSO2 ESB with the Analytics Dashboard

Use the procedure below to analyze WSO2 ESB using WSO2 Analytics.

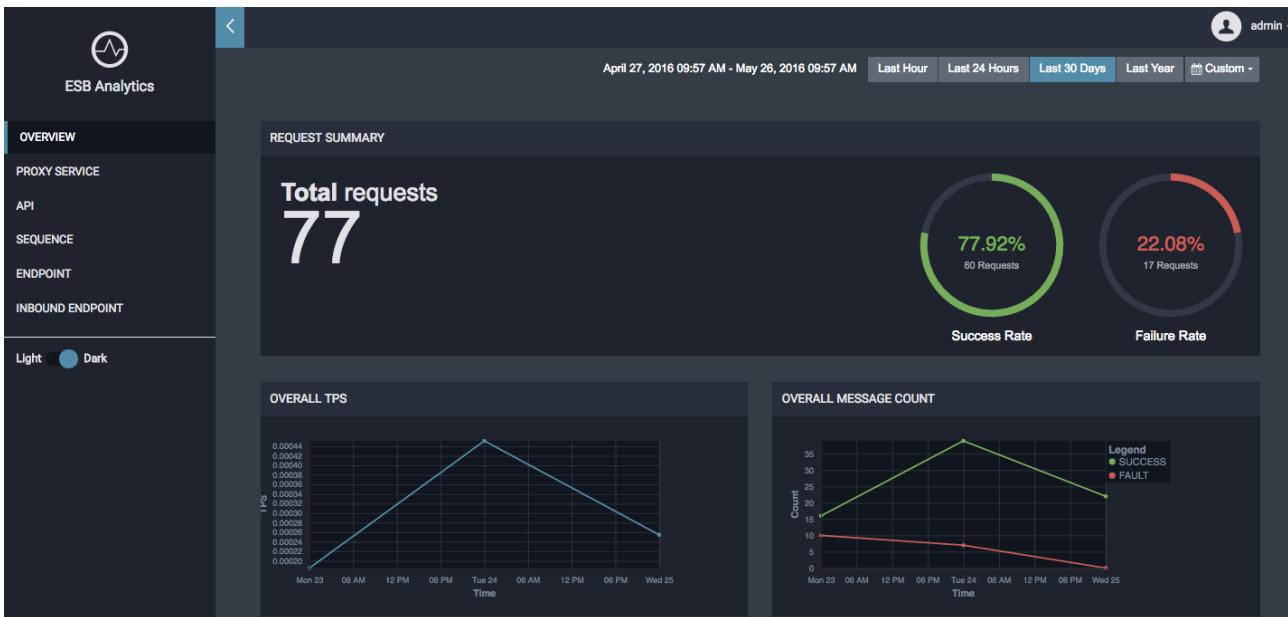
- To tune the performance for publishing ESB statistics in the Analytics Dashboard, see [Tuning Analytics](#).
- To troubleshoot ESB Analytics, you can use a debugging log by adding the following line to the <ESB\_HOME>/repository/conf/log4j.properties file.  
`log4j.logger.org.wso2.carbon.das.messageflow.data.publisher=DEBUG`

The Analytics Dashboard cannot be viewed using the Internet Explorer 10 and older versions.

1. Access the WSO2 ESB Analytics Management Console using the following URL, and log in using your credentials.  
`https://<ESB_ANALYTICS_HOST>:<ESB_ANALYTICS_PORT>/carbon/`
2. In the **Main** tab, click **Analytics Dashboard**, and log into the Analytics Dashboard by entering your credentials in the log in dialog box that appears. The following dashboard is displayed by default.



3. Click **View** to open the **ESB Analytics** dashboard. This dashboard is displayed as shown in the example below.



The contents of the **Overview** page are displayed by default. To view statistics for a specific date range, select the required date range from the top right menu bar. If you want to view statistics for a specific date, click **Custom** in the menu bar and enter the required date. For more information on analyzing the statistics displayed in the **Overview** page, see [Analyzing ESB Statistics Overview](#).

4. To view statistics for a proxy service, click on the relevant proxy service in the **TOP PROXY SERVICES BY REQUEST COUNT** gadget. This opens the **Overview/Proxy** page. You can also click **Proxy Service** in the left navigator to open this page and search for the required proxy service. For more information on analyzing the information displayed in this page, see [Analyzing Statistics for Proxy Services](#).

5. To view statistics for a REST API, click on the relevant REST API in the **TOP APIs BY REQUEST COUNT** gadget. This opens the **Overview/API** page. You can also click **API** in the left navigator to open this page and search for the required REST API. For more information on analyzing the information displayed in this page, see [Analyzing Statistics for REST APIs](#).
6. To view statistics for an endpoint, click on the relevant endpoint in the **TOP ENDPOINTS BY REQUEST COUNT** gadget. This opens the **Overview/Endpoint** page. You can also click **Endpoint** in the left navigator to open this page and search for the required endpoint. For more information on analyzing the information displayed in this page, see [Analyzing Statistics for Endpoints](#).
7. To view statistics for an inbound endpoint, click on the relevant inbound endpoint in the **TOP INBOUND ENDPOINTS BY REQUEST COUNT** gadget. This opens the **Overview/Inbound Endpoint** page. You can also click **Inbound Endpoint** in the left navigator to open this page and search for the required endpoint. For more information on analyzing the information displayed in this page, see [Analyzing Statistics for Inbound Endpoints](#).
8. To view statistics for a mediation sequence, click on the relevant sequence in the **TOP SEQUENCES BY REQUEST COUNT** gadget. This opens the **Overview/Sequence** page. You can also click **Sequence** in the left navigator to open this page and search for the required sequence. For more information on analyzing the information displayed in this page, see [Analyzing Statistics for Sequences](#).

### Analyzing ESB Statistics Overview

The **Overview** page of the **ESB Analytics** dashboard allows you to form a basic understanding of the performance of your web services and applications by analyzing the activities carried out by WSO2 ESB. This dashboard displays information such as the overall success rate of the ESB in terms of request handling, and the ranking of different ESB artifacts (such as proxy services, REST APIs, sequences etc.) based on their usage. This section explains how this information can be analyzed.

When you open the **ESB Analytics** dashboard, the **Overview** page is displayed by default. For more information about accessing the ESB Analytics Dashboard, see [Analyzing WSO2 ESB with the Analytics Dashboard](#).

At any given time, this page displays the statistics for a selected time interval. Make sure you select the required time interval in the following bar displayed at the top of the page. If you want to define a custom time interval, click **Custom** and select the start and end dates of the required time interval in the calendar that appears.



[Request Summary](#) | [Overall TPS](#) | [Overall Message Count](#) | [Top Proxy Services by Request Count](#) | [Top APIs by Request Count](#) | [Top Endpoints by Request Count](#) | [Top Inbound Endpoints by Request Count](#) | [Top Sequences by Request Count](#)

### Request Summary

<b>View (Example)</b>	<div style="background-color: #2e3436; color: white; padding: 10px; border-radius: 5px;"> <p style="margin: 0;">REQUEST SUMMARY</p> <p>Total requests <b>67051</b></p> <div style="display: flex; justify-content: space-around; margin-top: 10px;"> <div style="text-align: center;">  <p>99.98% 67050 Requests</p> <p>Success Rate</p> </div> <div style="text-align: center;">  <p>0.02% 13 Requests</p> <p>Failure Rate</p> </div> </div> </div>
<b>Description</b>	<p>This indicates the total number of requests that the ESB has handled during the selected period of time. The number of requests that have been successfully processed as well as the number of requests that have failed during the selected time period are displayed as percentages of the total number of requests handled.</p>

<b>Purpose</b>	This gadget can be used for the following purposes. <ul style="list-style-type: none"> <li>Understanding the extent to which the ESB was utilized during a selected time interval.</li> <li>Checking success rate of the ESB during a selected time interval.</li> </ul>
<b>Recommended Action</b>	If the success rate is too low, you can check the individual proxy services and REST APIs individually to identify which proxy services/REST APIs are affecting the overall success rate and take corrective action where required.

## Overall TPS

<b>View (Example)</b>	
<b>Description</b>	This indicates the number of transactions that the ESB has processed during a second.
<b>Purpose</b>	This indicator allows you to analyse the overall efficiency of the ESB in terms of the request processing speed.
<b>Recommended Action</b>	A high TPS value indicates that the ESB is handling a high load of requests. This requires you to ensure that your system is running smoothly in terms of resource allocation, and add more ESB nodes if required.

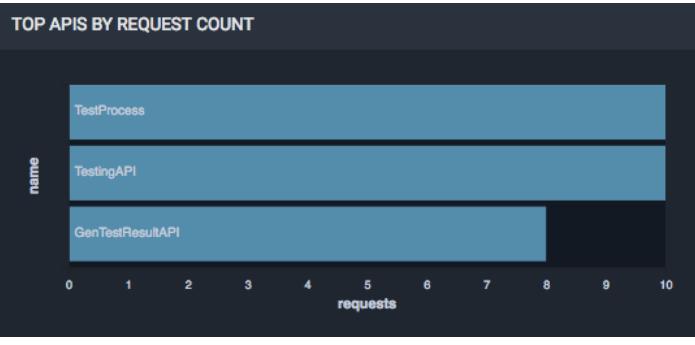
## Overall Message Count

<b>View (Example)</b>	
<b>Description</b>	This provides a graphical view of the total count of successful messages as well as failed messages during a selected time period.
<b>Purpose</b>	This helps you to identify any correlations that may exist during message failure and time.
<b>Recommended Action</b>	If the message failure is particularly high during a specific time, you can check whether any unusual occurrences have taken place during that time (e.g., system downtime) and take appropriate action.

## Top Proxy Services by Request Count

<b>View (Example)</b>	 <table border="1"> <thead> <tr> <th>name</th> <th>requests</th> </tr> </thead> <tbody> <tr><td>LicenseServiceProxy</td><td>~70,000</td></tr> <tr><td>TestProxy1</td><td>~65,000</td></tr> <tr><td>TestProxy9</td><td>~60,000</td></tr> <tr><td>TestProxy2</td><td>~55,000</td></tr> <tr><td>TestProxy8</td><td>~50,000</td></tr> </tbody> </table>	name	requests	LicenseServiceProxy	~70,000	TestProxy1	~65,000	TestProxy9	~60,000	TestProxy2	~55,000	TestProxy8	~50,000
name	requests												
LicenseServiceProxy	~70,000												
TestProxy1	~65,000												
TestProxy9	~60,000												
TestProxy2	~55,000												
TestProxy8	~50,000												
<b>Description</b>	<p>This ranks the most frequently used proxy services based on the usage. You can click on each proxy service displayed in this gadget to view more detailed information about it in the <b>OVERVIEW/PROXY/&lt;PROXY_SERVICE_NAME&gt;</b> page.</p>												
<b>Purpose</b>	<p>This allows you to identify the most frequently used proxy services during specific time periods and understand usage patterns relating to your applications and services based on that.</p>												
<b>Recommended Action</b>	<p>You can identify the most utilized proxy services during time intervals with a high rate of request failure, click on them to view whether their individual failure rates (in general) are high, and take corrective action if required.</p>												

## Top APIs by Request Count

<b>View (Example)</b>	 <table border="1"> <thead> <tr> <th>name</th> <th>requests</th> </tr> </thead> <tbody> <tr><td>TestProcess</td><td>~9.5</td></tr> <tr><td>TestingAPI</td><td>~9.5</td></tr> <tr><td>GenTestResultAPI</td><td>~8.5</td></tr> </tbody> </table>	name	requests	TestProcess	~9.5	TestingAPI	~9.5	GenTestResultAPI	~8.5
name	requests								
TestProcess	~9.5								
TestingAPI	~9.5								
GenTestResultAPI	~8.5								
<b>Description</b>	<p>This ranks the most frequently used REST APIs based on the usage.</p>								
<b>Purpose</b>	<p>This allows you to identify the most frequently used REST APIs during specific time periods and understand usage patterns relating to your applications and services based on that.</p>								
<b>Recommended Action</b>	<p>You can identify the most utilized REST APIs during time intervals with a high rate of request failure, click on them to view whether their individual failure rates (in general) are high, and take corrective action if required.</p>								

## Top Endpoints by Request Count

<b>View (Example)</b>	<p>TOP ENDPOINTS BY REQUEST COUNT</p> <table border="1"> <thead> <tr> <th>Endpoint Name</th> <th>Requests</th> </tr> </thead> <tbody> <tr> <td>InsuranceServiceEp</td> <td>~70,000</td> </tr> <tr> <td>LicenseServiceEp</td> <td>~60,000</td> </tr> <tr> <td>EmissionTestServiceEp</td> <td>~50,000</td> </tr> <tr> <td>PaymentServiceEp</td> <td>~40,000</td> </tr> </tbody> </table>	Endpoint Name	Requests	InsuranceServiceEp	~70,000	LicenseServiceEp	~60,000	EmissionTestServiceEp	~50,000	PaymentServiceEp	~40,000
Endpoint Name	Requests										
InsuranceServiceEp	~70,000										
LicenseServiceEp	~60,000										
EmissionTestServiceEp	~50,000										
PaymentServiceEp	~40,000										
<b>Description</b>	This ranks the most frequently used endpoints based on the usage.										
<b>Purpose</b>	This allows you to identify the most frequently used endpoints during specific time periods and understand usage patterns relating to your applications and services based on that.										
<b>Recommended Action</b>	You can identify the most utilized endpoints during time intervals with a high rate of request failure, click on them to view whether their individual failure rates (in general) are high, and take corrective action if required.										

### Top Inbound Endpoints by Request Count

<b>View (Example)</b>	
<b>Description</b>	This ranks the most frequently used inbound endpoints based on the usage.
<b>Purpose</b>	This allows you to identify the most frequently used inbound endpoints during specific time periods and understand usage patterns relating to your applications and services based on that.
<b>Recommended Action</b>	You can identify the most utilized inbound endpoints during time intervals with a high rate of request failure, click on them to view whether their individual failure rates (in general) are high, and take corrective action if required.

### Top Sequences by Request Count

<b>View (Example)</b>	<p>TOP SEQUENCES BY REQUEST COUNT</p> <table border="1"> <thead> <tr> <th>Sequence Name</th> <th>Requests</th> </tr> </thead> <tbody> <tr> <td>LogAndRespond</td> <td>~16</td> </tr> <tr> <td>AddPropertySend</td> <td>~10</td> </tr> </tbody> </table>	Sequence Name	Requests	LogAndRespond	~16	AddPropertySend	~10
Sequence Name	Requests						
LogAndRespond	~16						
AddPropertySend	~10						
<b>Description</b>	This ranks the most frequently used sequences based on their usage.						
<b>Purpose</b>	This helps to identify the most frequently used mediation patterns, which in turn helps to understand the nature of activities carried out by users of your applications and services.						

<b>Recommended Action</b>	You can identify the most utilized <a href="#">sequences</a> during time intervals with a high rate of request failure, click on them to view whether their individual failure rates (in general) are high, and take corrective action if required.
---------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Analyzing Statistics for Proxy Services

The WSO2 Analytics Dashboard displays statistics relating to a selected proxy service. The information displayed includes the overall success and failure rates, the number of successful/failed requests over time, the message latency over time, the list of messages processed during the selected time interval, and the mediation flow that the messages sent to the proxy service have passed through.

Use one of the following methods to view information in this page.

- When you open the **ESB Analytics** dashboard, the **Overview** page is displayed by default. Click on a proxy service in the **TOP PROXY SERVICES BY REQUEST COUNT** gadget to view information relating to that proxy service.
- Click on the **PROXY SERVICE** page and search for the proxy service for which you want to view information.

At any given time, this page displays the statistics for a selected time interval. Make sure you select the required time interval in the following bar displayed at the top of the page. If you want to define a custom time interval, click **Custom** and select the start and end dates of the required time interval in the calendar that appears.

Last Hour    Last 24 Hours    Last 30 Days    Last Year    Custom ▾

[Proxy Request Count](#) | [Message Count](#) | [Message Latency](#) | [Messages](#) | [Message Flow](#)

### Proxy Request Count

<b>View (Example)</b>	<p>The screenshot shows a dark-themed gadget titled "PROXY REQUEST COUNT". It displays the total number of requests for a proxy service named "LicenseServiceProxy" as "69810". Below this, two donut charts show the success and failure rates. The success rate is 99.98% (89799 Requests) and the failure rate is 0.02% (11 Requests).</p>
<b>Description</b>	This indicates the total number of requests handled by a <a href="#">proxy service</a> during a selected time period. The number of requests successfully handled by the proxy service as well as the number of failed requests are displayed as percentages of the total number requests handled by the proxy.
<b>Purpose</b>	This gadget can be used for the following purposes. <ul style="list-style-type: none"> <li>It allows you to identify the throughput handled by a proxy service during different time periods in order to understand the usage patterns the related applications/services.</li> <li>Comparing the success rate during different time intervals helps you to identify any unusual occurrences that have taken place during specific times (e.g., system downtime).</li> </ul>
<b>Recommended action</b>	Check the success/failure rate for a proxy at different time intervals. If the success rate was low only during particular time interval(s), check for unusual activities during that time interval(s) (e.g., system downtime, unavailability of the backend service etc.). If the success rate is low at all time intervals, check the proxy service configuration as well as the configurations of other artifacts used by the proxy service (i.e. sequences, endpoints etc.) for errors.

## Message Count

<b>View (Example)</b>	
<b>Description</b>	This provides a graphical view of the count for both successful and failed messages for a proxy service during a selected time interval.
<b>Purpose</b>	This allows you to identify any correlation that may exist between message failure rate, throughput and time.
<b>Recommended action</b>	If the message failure is particularly high during a specific time, you can check whether any unusual occurrences have taken place during that time (e.g., system downtime, unavailability of the back-end service) and take appropriate action.

## Message Latency

<b>View (Example)</b>	
<b>Description</b>	The time taken by the selected proxy service to complete processing a request sent to it.
<b>Purpose</b>	This allows you to analyse the efficiency of a proxy service in terms of the time taken to process messages during a selected time interval.
<b>Recommended action</b>	If the message latency is high during a specific time interval, view the number of requests handled by the proxy service during the same time interval in order to check whether there has been an overload of requests.

## Messages

<b>View (Example)</b>	
<b>Description</b>	<p>This provides the list of message IDs that were processed by the selected proxy service. Details including the host, time stamp and whether the message is successfully processed are displayed for each message ID. Messages can be sorted in an ascending or descending order based on the message ID, host, start time or status. You can search for specific message IDs, as well as click on a message ID to view more details about it in the <b>OVERVIEW/MESSAGE/&lt;MESSAGE_ID&gt;</b> page.</p>
<b>Purpose</b>	<p>This allows you to identify the individual message IDs that were not successfully processed and click on them to find more details in order to take corrective action.</p>
<b>Recommended action</b>	<ul style="list-style-type: none"> <li>Sort the messages by the status and check whether message failure correlates with specific time intervals. If such a pattern is identified, check whether any unusual activity (e.g., system downtime, unavailability of the backend service) has occurred during that time.</li> <li>Sort the messages by the status and check whether message failure correlates with a specific host. If message failures mainly occur for a specific port(s) investigate further.</li> <li>Click on the message IDs that have failed for further information about them in the <b>OVERVIEW/MESSAGE/&lt;MESSAGE_ID&gt;</b> page.</li> </ul>

## Message Flow

<b>View (Example)</b>	
<b>Description</b>	<p>This provides a graphical view of the mediation flow through which the messages handled by a proxy service have passed.</p>
<b>Purpose</b>	<p>This allows you to understand the different stages of mediation that a selected message ID has passed.</p>
<b>Recommended action</b>	<ul style="list-style-type: none"> <li>Check whether the message flow is correctly configured based on your requirement, and make modifications if required.</li> <li>Click on different mediators included in the message flow to view more details about them.</li> </ul>

## Analyzing Statistics for REST APIs

This section explains how to analyze the performance of REST APIs in ESB using WSO2 Analytics. The information displayed includes the overall success and failure rates, the number of successful/failed requests over time, the message latency over time, the list of messages processed during the selected time interval, and the mediation flow that the messages sent to the REST API have passed through.

Use one of the following methods to view information in this page.

- When you open the **ESB Analytics** dashboard, the **Overview** page is displayed by default. Click on a REST API in the **TOP APIS BY REQUEST COUNT** gadget to view information relating to that REST API.
- Click on the **API** page and search for the REST API for which you want to view information.

At any given time, this page displays the statistics for a selected time interval. Make sure you select the required time interval in the following bar displayed at the top of the page. If you want to define a custom time interval, click **Custom** and select the start and end dates of the required time interval in the calendar that appears.



[API Request Count](#) | [Message Count](#) | [Message Latency](#) | [Messages](#) | [Message Flow](#)

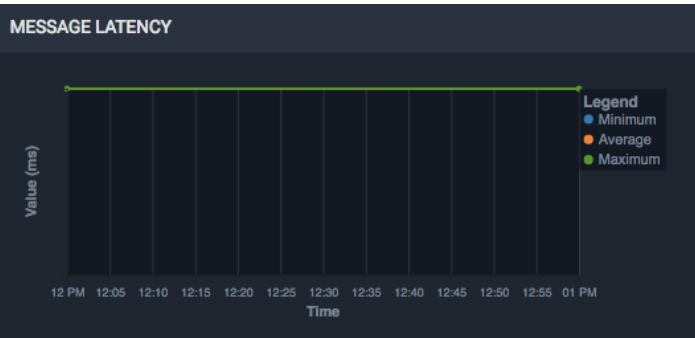
## API Request Count

<b>View (Example)</b>	<p>API REQUEST COUNT</p> <p>Total requests for GenTestResultAPI</p> <p>8</p> <p>100.00% 8 Requests</p> <p>Success Rate</p> <p>0.00% 0 Requests</p> <p>Failure Rate</p>
<b>Description</b>	<p>This indicates the total number of requests handled by a REST API during a selected time period, and the rate of success within the same period.</p>
<b>Purpose</b>	<p>This allows you to identify the throughput handled by a REST API during different time periods in order to understand the usage patterns the related applications/services. Comparing the success rate during different time intervals helps you to identify any unusual occurrences that have taken place during specific times (e.g., system downtime).</p>
<b>Recommended action</b>	<p>Check the success/failure rate for a REST API at different time intervals. If the success rate was low only during particular time interval(s), check for unusual activities during that time interval(s) (e.g., system downtime, unavailability of the backend service etc.). If the success rate is low at all time intervals, check the REST API configuration as well as the configurations of other artifacts used by the REST API (i.e. sequences, endpoints etc.) for errors.</p>

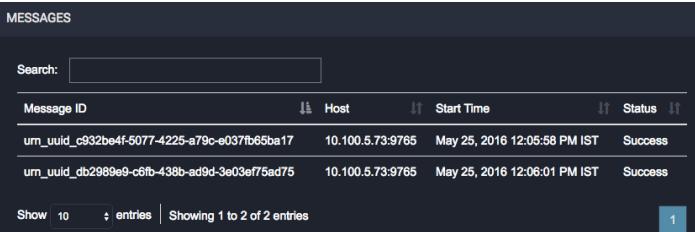
## Message Count

<b>View (Example)</b>	
<b>Description</b>	This provides a graphical view of the count for both successful and failed messages for a REST API during a selected time interval.
<b>Purpose</b>	This allows you to identify any correlation that may exist between message failure rate, throughput and time.
<b>Recommended action</b>	If the message failure is particularly high during a specific time, you can check whether any unusual occurrences have taken place during that time (e.g., system downtime, unavailability of the back-end service) and take appropriate action.

## Message Latency

<b>View (Example)</b>	
<b>Description</b>	The time taken by the selected REST API to complete processing a request sent to it.
<b>Purpose</b>	This allows you to analyse the efficiency of a REST API in terms of the time taken to process messages during a selected time interval.
<b>Recommended action</b>	If the message latency is high during a specific time interval, view the number of requests handled by the REST API during the same time interval in order to check whether there has been an overload of requests.

## Messages

<b>View (Example)</b>	
---------------------------	--------------------------------------------------------------------------------------

<b>Description</b>	This provides the list of message IDs that were processed by a REST API. Details including the port, time stamp and whether the message is successfully processed are displayed for each message ID. Messages can be sorted in an ascending or descending order based on the message ID, host, start time or status. You can search for specific message IDs, as well as click on a message ID to view more details about it in the <b>OVERVIEW/MESSAGE/&lt;MESSAGE_ID&gt;</b> page.
<b>Purpose</b>	This allows you to identify the individual message IDs that were not successfully processed and click on them to find more details in order to take corrective action.
<b>Recommended action</b>	<ul style="list-style-type: none"> <li>Sort the messages by the status and check whether message failure correlates with specific time intervals. If such a pattern is identified, check whether any unusual activity (e.g., system downtime, unavailability of the backend service) has occurred during that time.</li> <li>Sort the messages by the status and check whether message failure correlates with a specific host. If message failures mainly occur for a specific port(s) investigate further.</li> <li>Click on the message IDs that have failed for further information about them in the <b>OVERVIEW/MESSAGE/&lt;MESSAGE_ID&gt;</b> page.</li> </ul>

## Message Flow

<b>View (Example)</b>	
<b>Description</b>	This provides a graphical view of the mediation flow through which the messages handled by a REST API have passed.
<b>Purpose</b>	This allows you to understand the different stages of mediation that a selected message ID has passed.
<b>Recommended action</b>	<ul style="list-style-type: none"> <li>Check whether the message flow is correctly configured based on your requirement, and make modifications if required.</li> <li>Click on different mediators included in the message flow to view more details about them.</li> </ul>

## Analyzing Statistics for Sequences

This section explains how to analyze mediation sequences in WSO2 ESB using WSO2 Analytics.

Use one of the following methods to view information in this page.

- When you open the **ESB Analytics** dashboard, the **Overview** page is displayed by default. Click on a sequence in the **TOP PROXY SEQUENCES BY REQUEST COUNT** gadget to view information relating to that sequence
- Click on the **SEQUENCE** page and search for the sequence for which you want to view information.

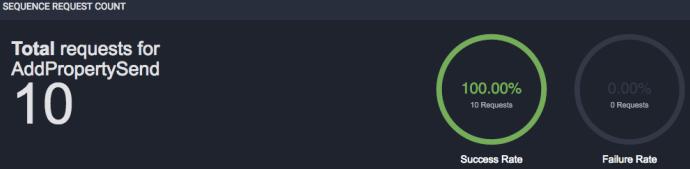
At any given time, this page displays the statistics for a selected time interval. Make sure you select the required time interval in the following bar displayed at the top of the page. If you want to define a custom

time interval, click **Custom** and select the start and end dates of the required time interval in the calendar that appears.

[Last Hour](#) [Last 24 Hours](#) [Last 30 Days](#) [Last Year](#) [Custom](#)

[Sequence Request Count](#) | [Message Count](#) | [Message Latency](#) | [Messages](#) | [Message Flow](#)

### Sequence Request Count

<b>View (Example)</b>	 <p>SEQUENCE REQUEST COUNT</p> <p>Total requests for AddPropertySend <b>10</b></p> <p>100.00% 10 Requests</p> <p>0.00% 0 Requests</p> <p>Success Rate Failure Rate</p>
<b>Description</b>	This indicates the total number of requests that have passed through a mediation sequence during a selected time period, and the rate of success within the same period.
<b>Purpose</b>	This allows you to identify the throughput of messages that have passed through a mediation sequence during different time periods in order to understand the usage patterns the related applications/services. Comparing the success rate during different time intervals helps you to identify any unusual occurrences that have taken place during specific times (e.g., system downtime).
<b>Recommended action</b>	Check the success/failure rate for a sequence at different time intervals. If the success rate was low only during particular time interval(s), check for unusual activities during that time interval(s) (e.g., system downtime, unavailability of the backend service etc.). If the success rate is low at all time intervals, check the sequence for configuration errors.

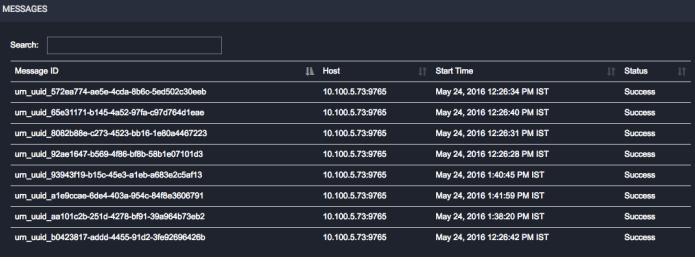
### Message Count

<b>View (Example)</b>	 <p>MESSAGE COUNT</p> <p>Count</p> <p>Time</p> <p>Legend: SUCCESS (Green Line), FAULT (Red Line)</p> <table border="1"> <thead> <tr> <th>Time</th> <th>SUCCESS</th> <th>FAULT</th> </tr> </thead> <tbody> <tr> <td>06 PM</td> <td>10</td> <td>4</td> </tr> <tr> <td>09 PM</td> <td>10</td> <td>6</td> </tr> <tr> <td>Tue 24</td> <td>11</td> <td>6</td> </tr> <tr> <td>03 AM</td> <td>11</td> <td>6</td> </tr> <tr> <td>06 AM</td> <td>12</td> <td>6</td> </tr> <tr> <td>09 AM</td> <td>13</td> <td>7</td> </tr> </tbody> </table>	Time	SUCCESS	FAULT	06 PM	10	4	09 PM	10	6	Tue 24	11	6	03 AM	11	6	06 AM	12	6	09 AM	13	7
Time	SUCCESS	FAULT																				
06 PM	10	4																				
09 PM	10	6																				
Tue 24	11	6																				
03 AM	11	6																				
06 AM	12	6																				
09 AM	13	7																				
<b>Description</b>	This provides a graphical view of the count for both successful and failed messages that have passed through a mediation sequence during a selected time interval.																					
<b>Purpose</b>	This allows you to identify any correlation that may exist between message failure rate, throughput and time for a specific sequence.																					
<b>Recommended action</b>	If the message failure is particularly high during a specific time, you can check whether any unusual occurrences have taken place during that time (e.g., system downtime, unavailability of the back-end service) and take appropriate action.																					

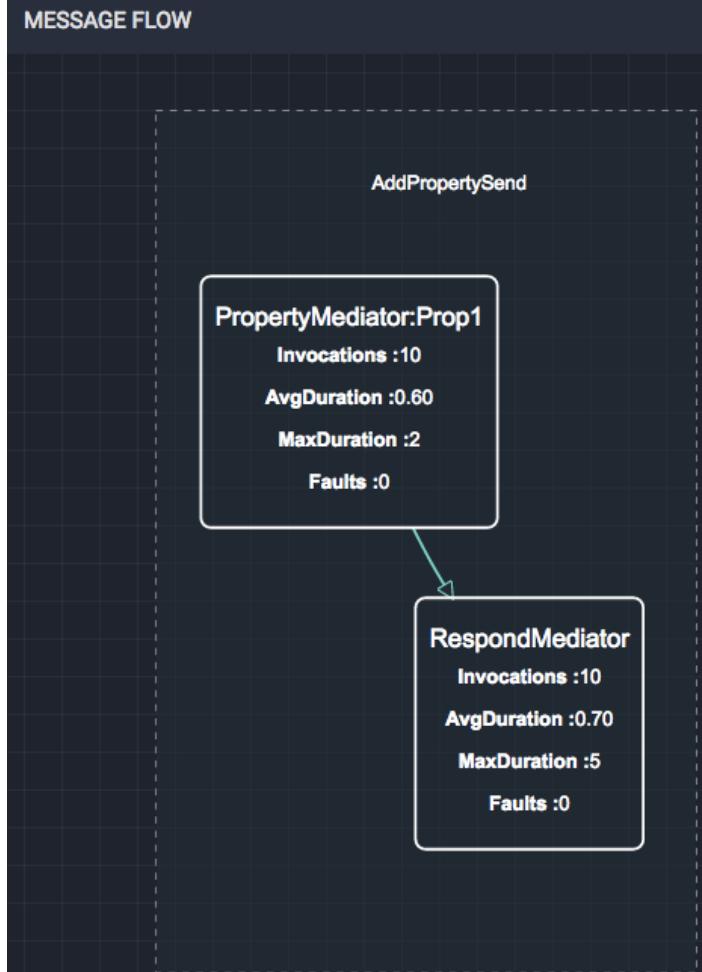
### Message Latency

<b>View (Example)</b>	
<b>Description</b>	The time taken by the selected sequence to complete the mediation it carries out per request.
<b>Purpose</b>	This enables you to understand the level of efficiency with which the mediation is carried out by sequences.
<b>Recommended action</b>	If the message latency of a sequence is too high, check its configuration and consider alternative configurations through which the same mediation can be done.

## Messages

<b>View (Example)</b>	
<b>Description</b>	This provides the list of message IDs that were processed by the selected sequence. Details including the port, time stamp and whether the message is successfully processed are displayed for each message ID. Messages can be sorted in an ascending or descending order based on the message ID, host, start time or status. You can search for specific message IDs, as well as click on a message ID to view more details about it in the <b>OVERVIEW/MESSAGE/&lt;MESSAGE_ID&gt;</b> page.
<b>Purpose</b>	This allows you to identify the individual message IDs that were not successfully processed and click on them to find more details in order to take corrective action.
<b>Recommended action</b>	<ul style="list-style-type: none"> <li>Sort the messages by the status and check whether message failure correlates with specific time intervals. If such a pattern is identified, check whether any unusual activity (e.g., system downtime, unavailability of the backend service) has occurred during that time.</li> <li>Sort the messages by the status and check whether message failure correlates with a specific host. If message failures mainly occur for a specific port(s) investigate further.</li> <li>Click on the message IDs that have failed for further information about them in the <b>OVERVIEW/MESSAGE/&lt;MESSAGE_ID&gt;</b> page.</li> </ul>

## Message Flow

<b>View (Example)</b>	
<b>Description</b>	<p>This provides a graphical view of the mediation flow through which the messages handled by a sequence have passed.</p>
<b>Purpose</b>	<p>This allows you to understand the different stages of mediation that a selected message ID has passed.</p>
<b>Recommended action</b>	<ul style="list-style-type: none"> <li>Check whether the message flow is correctly configured based on your requirement, and make modifications if required.</li> <li>Click on different mediators included in the message flow to view more details about them.</li> </ul>

## Analyzing Statistics for Endpoints

This section explains how to view and analyze statistics relating to WSO2 ESB endpoints using WSO2 Analytics.

Use one of the following methods to view information in this page.

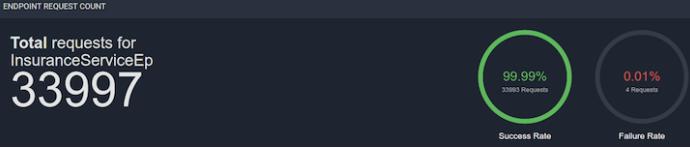
- When you open the **ESB Analytics** dashboard, the **Overview** page is displayed by default. Click on an endpoint in the **TOP ENDPOINTS BY REQUEST COUNT** gadget to view information relating to that endpoint.
- Click **ENDPOINT** in the left navigator to open the **OVERVIEW/ENDPOINT** page, and search for the endpoint for which you want to view information.
- Click on an endpoint that appears in the **MESSAGE FLOW** gadget in any of the following pages. This opens the **OVERVIEW/ENDPOINT** page with information relating to the endpoint you clicked on.
  - OVERVIEW/PROXY**
  - OVERVIEW/API**

- OVERVIEW/SEQUENCES

At any given time, this page displays the statistics for a selected time interval. Make sure you select the required time interval in the following bar displayed at the top of the page. If you want to define a custom time interval, click **Custom** and select the start and end dates of the required time interval in the calendar that appears.

[Last Hour](#)[Last 24 Hours](#)[Last 30 Days](#)[Last Year](#)[Custom](#) ▾
[Endpoint Request Count](#) | [Message Count](#) | [Message Latency](#) | [Messages](#)

### Endpoint Request Count

<b>View (Example)</b>	
<b>Description</b>	This indicates the total number of requests sent to the selected endpoint during a selected time interval. The successfully processed messages as well as the failed messages are displayed as a percentage of the total number of messages sent to the endpoint.
<b>Purpose</b>	This allows you to assess the extent to which an endpoint is used as well as the validity of the endpoint.
<b>Recommended action</b>	Compare the endpoint success rate for different time intervals. If the success rate is low during a specific time interval, check whether any unusual occurrences have taken place during that time interval (e.g., system downtime, unavailability of the back-end service).

### Message Count

<b>View (Example)</b>	
<b>Description</b>	This provides a graphical view of the count for both successful and failed messages that have been sent to an endpoint during a selected time interval.
<b>Purpose</b>	This allows you to identify any correlation that may exist between message failure rate, throughput and time. As a result, you can identify unusual occurrences that may have occurred during specific times (e.g., system downtime) as well as be aware if an endpoint cannot handle a high throughput.
<b>Recommended action</b>	If the message failure is particularly high during a specific time, you can check whether any unusual occurrences have taken place during that time (e.g., system downtime, unavailability of the back-end service) and take appropriate action.

## Message Latency

<b>View (Example)</b>	
<b>Description</b>	The time taken per request sent to the selected endpoint to pass through the complete mediation flow.
<b>Purpose</b>	This allows you to understand the efficiency with which messages received by the ESB are sent to the selected endpoint.
<b>Recommended action</b>	<ul style="list-style-type: none"> <li>If the message latency is high during a specific time interval, view the number of requests sent to the endpoint during the same time interval in order to check whether the endpoint has been available during that time interval.</li> <li>If the message latency is persistently low, check the back-end service for system errors.</li> </ul>

## Messages

<b>View (Example)</b>	<table border="1"> <thead> <tr> <th>Message ID</th><th>Host</th><th>Start Time</th><th>Status</th></tr> </thead> <tbody> <tr> <td>urn_uuid_ad24a7bf-d7f3-4ac9-8034-30827ff6bec8</td><td>10.100.5.73:9785</td><td>May 17, 2016 2:48:48 PM IST</td><td>Success</td></tr> <tr> <td>urn_uuid_89a4ec20-0202-493d-a7cc-07a67af58125</td><td>10.100.5.73:9785</td><td>May 17, 2016 2:13:47 PM IST</td><td>Success</td></tr> <tr> <td>urn_uuid_c1c3c809-dc1a-4825-876d-86c01d69f78e</td><td>10.100.5.73:9785</td><td>May 17, 2016 2:49:15 PM IST</td><td>Success</td></tr> <tr> <td>urn_uuid_25cd6854-5da0-4d69-8821-82a990ed2bd3</td><td>10.100.5.73:9785</td><td>May 17, 2016 2:56:15 PM IST</td><td>Success</td></tr> <tr> <td>urn_uuid_6847611e-d458-4c7e-83e6-b01e791eefaf3</td><td>10.100.5.73:9785</td><td>May 17, 2016 2:57:25 PM IST</td><td>Success</td></tr> <tr> <td>urn_uuid_745edd67-ba5e-4bf4-84a6-3d460b64e15d</td><td>10.100.5.73:9785</td><td>May 17, 2016 2:57:25 PM IST</td><td>Success</td></tr> <tr> <td>urn_uuid_80224397-2cd3-4c79-b12d-9fd3da3c0e4</td><td>10.100.5.73:9785</td><td>May 17, 2016 3:37:49 PM IST</td><td>Failed</td></tr> <tr> <td>urn_uuid_3b311e24-f965-4676-aa43-78823382b0f2</td><td>10.100.5.73:9785</td><td>May 18, 2016 9:44:17 AM IST</td><td>Success</td></tr> </tbody> </table>	Message ID	Host	Start Time	Status	urn_uuid_ad24a7bf-d7f3-4ac9-8034-30827ff6bec8	10.100.5.73:9785	May 17, 2016 2:48:48 PM IST	Success	urn_uuid_89a4ec20-0202-493d-a7cc-07a67af58125	10.100.5.73:9785	May 17, 2016 2:13:47 PM IST	Success	urn_uuid_c1c3c809-dc1a-4825-876d-86c01d69f78e	10.100.5.73:9785	May 17, 2016 2:49:15 PM IST	Success	urn_uuid_25cd6854-5da0-4d69-8821-82a990ed2bd3	10.100.5.73:9785	May 17, 2016 2:56:15 PM IST	Success	urn_uuid_6847611e-d458-4c7e-83e6-b01e791eefaf3	10.100.5.73:9785	May 17, 2016 2:57:25 PM IST	Success	urn_uuid_745edd67-ba5e-4bf4-84a6-3d460b64e15d	10.100.5.73:9785	May 17, 2016 2:57:25 PM IST	Success	urn_uuid_80224397-2cd3-4c79-b12d-9fd3da3c0e4	10.100.5.73:9785	May 17, 2016 3:37:49 PM IST	Failed	urn_uuid_3b311e24-f965-4676-aa43-78823382b0f2	10.100.5.73:9785	May 18, 2016 9:44:17 AM IST	Success
Message ID	Host	Start Time	Status																																		
urn_uuid_ad24a7bf-d7f3-4ac9-8034-30827ff6bec8	10.100.5.73:9785	May 17, 2016 2:48:48 PM IST	Success																																		
urn_uuid_89a4ec20-0202-493d-a7cc-07a67af58125	10.100.5.73:9785	May 17, 2016 2:13:47 PM IST	Success																																		
urn_uuid_c1c3c809-dc1a-4825-876d-86c01d69f78e	10.100.5.73:9785	May 17, 2016 2:49:15 PM IST	Success																																		
urn_uuid_25cd6854-5da0-4d69-8821-82a990ed2bd3	10.100.5.73:9785	May 17, 2016 2:56:15 PM IST	Success																																		
urn_uuid_6847611e-d458-4c7e-83e6-b01e791eefaf3	10.100.5.73:9785	May 17, 2016 2:57:25 PM IST	Success																																		
urn_uuid_745edd67-ba5e-4bf4-84a6-3d460b64e15d	10.100.5.73:9785	May 17, 2016 2:57:25 PM IST	Success																																		
urn_uuid_80224397-2cd3-4c79-b12d-9fd3da3c0e4	10.100.5.73:9785	May 17, 2016 3:37:49 PM IST	Failed																																		
urn_uuid_3b311e24-f965-4676-aa43-78823382b0f2	10.100.5.73:9785	May 18, 2016 9:44:17 AM IST	Success																																		
<b>Description</b>	This provides the list of message IDs that were processed by a proxy service. Details including the port, time stamp and whether the message is successfully processed are displayed for each message ID. Messages can be sorted in an ascending or descending order based on the message ID, host, start time or status. You can search for specific message IDs, as well as click on a message ID to view more details about it in the <b>OVERVIEW/MESSAGE/&lt;MESSAGE_ID&gt;</b> page.																																				
<b>Purpose</b>	This allows you to identify the individual message IDs that were not successfully processed and click on them to find more details in order to take corrective action.																																				

<b>Recommended action</b>	<ul style="list-style-type: none"> <li>Sort the messages by the status and check whether message failure correlates with specific time intervals. If such a pattern is identified, check whether any unusual activity (e.g., system downtime, unavailability of the backend service) has occurred during that time.</li> <li>Sort the messages by the status and check whether message failure correlates with a specific host. If message failures mainly occur for a specific port(s) investigate further.</li> <li>Click on the message IDs that have failed for further information about them in the <b>OVERVIEW/MESSAGE/&lt;MESSAGE_ID&gt;</b> page.</li> </ul>
---------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Analyzing Statistics for Inbound Endpoints

This section explains how to view and analyze statistics relating to inbound endpoints in WSO2 ESB using WSO2 Analytics.

Use one of the following methods to view information in this page.

- When you open the **ESB Analytics** dashboard, the **Overview** page is displayed by default. Click on an inbound endpoint in the **TOP INBOUND ENDPOINTS BY REQUEST COUNT** gadget to view information relating to that inbound endpoint.
- Click on the **INBOUND ENDPOINT** page and search for the inbound endpoint for which you want to view information.

At any given time, this page displays the statistics for a selected time interval. Make sure you select the required time interval in the following bar displayed at the top of the page. If you want to define a custom time interval, click **Custom** and select the start and end dates of the required time interval in the calendar that appears.



## | Inbound Endpoint Request Count | Message Count | Message Latency | Messages | Message Flow

### Inbound Endpoint Request Count

<b>View (Example)</b>	<div style="background-color: #2e3436; color: white; padding: 10px; text-align: center;"> <b>INBOUND ENDPOINT REQUEST COUNT</b>            Total requests for HttpListenerEP  <b>642</b> <div style="display: flex; justify-content: space-around; margin-top: 10px;"> <div style="text-align: center;">  <p>97.98%</p> <p>629 Requests</p> <p>Success Rate</p> </div> <div style="text-align: center;">  <p>2.02%</p> <p>13 Requests</p> <p>Failure Rate</p> </div> </div> </div>
<b>Description</b>	This indicates the total number of requests received by an inbound endpoint during a selected time interval, and the percentage of these requests that were successfully processed.
<b>Purpose</b>	This allows you to assess the extent to which an endpoint is used as well as the validity of the endpoint.
<b>Recommended action</b>	Check the success rate for an inbound endpoint at different time intervals. If the success rate was low only during particular time interval(s), check for unusual activities during that time interval(s) (e.g., system downtime, unavailability of the backend service etc.). If the success rate is low at all time intervals, check the configuration of the inbound endpoint as well as that of the mediation sequence used to call the inbound endpoint.

### Message Count

<b>View (Example)</b>	<p><b>MESSAGE COUNT</b></p> <table border="1"> <thead> <tr> <th>Time</th><th>Count</th></tr> </thead> <tbody> <tr><td>10:50</td><td>20</td></tr> <tr><td>10:51</td><td>110</td></tr> <tr><td>10:52</td><td>70</td></tr> <tr><td>10:53</td><td>120</td></tr> <tr><td>10:54</td><td>125</td></tr> <tr><td>10:55</td><td>100</td></tr> <tr><td>10:56</td><td>80</td></tr> <tr><td>10:57</td><td>130</td></tr> <tr><td>10:58</td><td>105</td></tr> <tr><td>10:59</td><td>85</td></tr> <tr><td>11:00</td><td>60</td></tr> <tr><td>11:01</td><td>10</td></tr> <tr><td>11:02</td><td>15</td></tr> <tr><td>11:03</td><td>20</td></tr> <tr><td>11:04</td><td>10</td></tr> <tr><td>11:05</td><td>5</td></tr> </tbody> </table>	Time	Count	10:50	20	10:51	110	10:52	70	10:53	120	10:54	125	10:55	100	10:56	80	10:57	130	10:58	105	10:59	85	11:00	60	11:01	10	11:02	15	11:03	20	11:04	10	11:05	5
Time	Count																																		
10:50	20																																		
10:51	110																																		
10:52	70																																		
10:53	120																																		
10:54	125																																		
10:55	100																																		
10:56	80																																		
10:57	130																																		
10:58	105																																		
10:59	85																																		
11:00	60																																		
11:01	10																																		
11:02	15																																		
11:03	20																																		
11:04	10																																		
11:05	5																																		
<b>Description</b>	This provides a graphical view of the count for both successful and failed messages that have been sent to an inbound endpoint during a selected time interval.																																		
<b>Purpose</b>	This allows you to identify any correlation that may exist between message failure rate, throughput and time. As a result, you can identify unusual occurrences that may have occurred during specific times (e.g., system downtime) as well as be aware if an inbound endpoint cannot handle a high throughput.																																		
<b>Recommended action</b>	If the message failure is particularly high during a specific time, you can check whether any unusual occurrences have taken place during that time (e.g., system downtime, unavailability of the back-end service) and take appropriate action.																																		

## Message Latency

<b>View (Example)</b>	<p><b>MESSAGE LATENCY</b></p> <table border="1"> <thead> <tr> <th>Time</th><th>Value (ms)</th></tr> </thead> <tbody> <tr><td>10:50</td><td>10</td></tr> <tr><td>10:51</td><td>10</td></tr> <tr><td>10:52</td><td>10</td></tr> <tr><td>10:53</td><td>10</td></tr> <tr><td>10:54</td><td>10</td></tr> <tr><td>10:55</td><td>10</td></tr> <tr><td>10:56</td><td>10</td></tr> <tr><td>10:57</td><td>10</td></tr> <tr><td>10:58</td><td>10</td></tr> <tr><td>10:59</td><td>10</td></tr> <tr><td>11:00</td><td>10</td></tr> <tr><td>11:01</td><td>10</td></tr> <tr><td>11:02</td><td>10</td></tr> <tr><td>11:03</td><td>10</td></tr> <tr><td>11:04</td><td>10</td></tr> <tr><td>11:05</td><td>10</td></tr> </tbody> </table>	Time	Value (ms)	10:50	10	10:51	10	10:52	10	10:53	10	10:54	10	10:55	10	10:56	10	10:57	10	10:58	10	10:59	10	11:00	10	11:01	10	11:02	10	11:03	10	11:04	10	11:05	10
Time	Value (ms)																																		
10:50	10																																		
10:51	10																																		
10:52	10																																		
10:53	10																																		
10:54	10																																		
10:55	10																																		
10:56	10																																		
10:57	10																																		
10:58	10																																		
10:59	10																																		
11:00	10																																		
11:01	10																																		
11:02	10																																		
11:03	10																																		
11:04	10																																		
11:05	10																																		
<b>Description</b>	This gadget provides a graphical illustration of the average time taken to process a request sent to the inbound endpoint at different times within the selected time interval.																																		
<b>Purpose</b>	This allows you to understand the efficiency with which requests were sent to an inbound endpoint within a specific time interval in terms of the time taken per request.																																		
<b>Recommended action</b>	If the message latency for an inbound endpoint is high during a specific time interval, check the number of requests sent to the inbound endpoint during the same time interval to identify whether the high latency was caused by an overload of requests.																																		

## Messages

<b>View (Example)</b>	<p>MESSAGES</p> <table border="1"> <thead> <tr> <th>Message ID</th><th>Host</th><th>Start Time</th><th>Status</th></tr> </thead> <tbody> <tr><td>um_uuid_003a84aa-b5fe-40f5-a9c8-b97086048ece</td><td>10.100.5.73:9763</td><td>June 13, 2016 10:53:32 AM IST</td><td>Success</td></tr> <tr><td>um_uuid_006a5fa0-2bcf-45bd-93ac-546fe97931b6</td><td>10.100.5.73:9763</td><td>June 13, 2016 10:55:07 AM IST</td><td>Success</td></tr> <tr><td>um_uuid_00757bd8299-4bde-8e1e-33d6272b1530</td><td>10.100.5.73:9763</td><td>June 13, 2016 11:07:23 AM IST</td><td>Success</td></tr> <tr><td>um_uuid_01b880d7-0153-4a76-af07-e000c260d50f</td><td>10.100.5.73:9763</td><td>June 13, 2016 10:55:09 AM IST</td><td>Success</td></tr> <tr><td>um_uuid_02040a62-894f-4a0f-ae94-664459dcdbab</td><td>10.100.5.73:9763</td><td>June 13, 2016 10:55:19 AM IST</td><td>Success</td></tr> <tr><td>um_uuid_029cb0be-acab-4ef1-b7fc-e7040951e3d9</td><td>10.100.5.73:9763</td><td>June 13, 2016 10:47:59 AM IST</td><td>Success</td></tr> <tr><td>um_uuid_031755db-2173-4535-bde1-4866abaeef310</td><td>10.100.5.73:9763</td><td>June 13, 2016 10:50:32 AM IST</td><td>Success</td></tr> <tr><td>um_uuid_03644ed5-52cb-49a9-be9e-8aab1e1cd4d</td><td>10.100.5.73:9763</td><td>June 13, 2016 10:53:36 AM IST</td><td>Success</td></tr> <tr><td>um_uuid_0405e414-96a9-42a1-a0ef-78947e8a178c</td><td>10.100.5.73:9763</td><td>June 13, 2016 10:55:20 AM IST</td><td>Success</td></tr> <tr><td>um_uuid_0470108a-75d4-414d-b476-dab313647d8</td><td>10.100.5.73:9763</td><td>June 13, 2016 10:53:32 AM IST</td><td>Success</td></tr> </tbody> </table>	Message ID	Host	Start Time	Status	um_uuid_003a84aa-b5fe-40f5-a9c8-b97086048ece	10.100.5.73:9763	June 13, 2016 10:53:32 AM IST	Success	um_uuid_006a5fa0-2bcf-45bd-93ac-546fe97931b6	10.100.5.73:9763	June 13, 2016 10:55:07 AM IST	Success	um_uuid_00757bd8299-4bde-8e1e-33d6272b1530	10.100.5.73:9763	June 13, 2016 11:07:23 AM IST	Success	um_uuid_01b880d7-0153-4a76-af07-e000c260d50f	10.100.5.73:9763	June 13, 2016 10:55:09 AM IST	Success	um_uuid_02040a62-894f-4a0f-ae94-664459dcdbab	10.100.5.73:9763	June 13, 2016 10:55:19 AM IST	Success	um_uuid_029cb0be-acab-4ef1-b7fc-e7040951e3d9	10.100.5.73:9763	June 13, 2016 10:47:59 AM IST	Success	um_uuid_031755db-2173-4535-bde1-4866abaeef310	10.100.5.73:9763	June 13, 2016 10:50:32 AM IST	Success	um_uuid_03644ed5-52cb-49a9-be9e-8aab1e1cd4d	10.100.5.73:9763	June 13, 2016 10:53:36 AM IST	Success	um_uuid_0405e414-96a9-42a1-a0ef-78947e8a178c	10.100.5.73:9763	June 13, 2016 10:55:20 AM IST	Success	um_uuid_0470108a-75d4-414d-b476-dab313647d8	10.100.5.73:9763	June 13, 2016 10:53:32 AM IST	Success
Message ID	Host	Start Time	Status																																										
um_uuid_003a84aa-b5fe-40f5-a9c8-b97086048ece	10.100.5.73:9763	June 13, 2016 10:53:32 AM IST	Success																																										
um_uuid_006a5fa0-2bcf-45bd-93ac-546fe97931b6	10.100.5.73:9763	June 13, 2016 10:55:07 AM IST	Success																																										
um_uuid_00757bd8299-4bde-8e1e-33d6272b1530	10.100.5.73:9763	June 13, 2016 11:07:23 AM IST	Success																																										
um_uuid_01b880d7-0153-4a76-af07-e000c260d50f	10.100.5.73:9763	June 13, 2016 10:55:09 AM IST	Success																																										
um_uuid_02040a62-894f-4a0f-ae94-664459dcdbab	10.100.5.73:9763	June 13, 2016 10:55:19 AM IST	Success																																										
um_uuid_029cb0be-acab-4ef1-b7fc-e7040951e3d9	10.100.5.73:9763	June 13, 2016 10:47:59 AM IST	Success																																										
um_uuid_031755db-2173-4535-bde1-4866abaeef310	10.100.5.73:9763	June 13, 2016 10:50:32 AM IST	Success																																										
um_uuid_03644ed5-52cb-49a9-be9e-8aab1e1cd4d	10.100.5.73:9763	June 13, 2016 10:53:36 AM IST	Success																																										
um_uuid_0405e414-96a9-42a1-a0ef-78947e8a178c	10.100.5.73:9763	June 13, 2016 10:55:20 AM IST	Success																																										
um_uuid_0470108a-75d4-414d-b476-dab313647d8	10.100.5.73:9763	June 13, 2016 10:53:32 AM IST	Success																																										
<b>Description</b>	This provides the list of message IDs that were sent to the inbound endpoint during the selected time interval. Details including the port, time stamp and whether the message is successfully processed are displayed for each message ID. Messages can be sorted in an ascending or descending order based on the message ID, host, start time or status. You can search for specific message IDs, as well as click on a message ID to view more details about it in the <b>OVERVIEW/MESSAGE/&lt;MESSAGE_ID&gt;</b> page.																																												
<b>Purpose</b>	This allows you to identify the individual message IDs that were not successfully processed and click on them to find more details in order to take corrective action.																																												
<b>Recommended action</b>	<ul style="list-style-type: none"> <li>Sort the messages by the status and check whether message failure correlates with specific time intervals. If such a pattern is identified, check whether any unusual activity (e.g., system downtime, unavailability of the backend service) has occurred during that time.</li> <li>Sort the messages by the status and check whether message failure correlates with a specific host. If message failures mainly occur for a specific port(s) investigate further.</li> <li>Click on the message IDs that have failed for further information about them in the <b>OVERVIEW/MESSAGE/&lt;MESSAGE_ID&gt;</b> page.</li> </ul>																																												

## Message Flow

<b>View (Example)</b>	<p>MESSAGE FLOW</p>
<b>Description</b>	This provides a graphical view of the mediation flow through which the messages sent to a specific inbound endpoint have passed.
<b>Purpose</b>	This allows you to identify the different stages of the message flow that all the messages sent to the inbound endpoint have passed through.
<b>Recommended action</b>	<ul style="list-style-type: none"> <li>Check whether the message flow is correctly configured based on your requirement, and make modifications if required.</li> <li>Click on different mediators included in the message flow to view more details about them.</li> </ul>

## Analyzing Statistics for Mediators

This section explains how to view statistics relating to WSO2 ESB mediators and analyze them using WSO2 Analytics.

This page displays information about a specific mediator configuration included in a specific proxy service/REST API/sequence. In order to view statistics for a specific mediator configuration, do the following.

- Click on the **PROXY SERVICE** page and search for the proxy service with the mediator you want to analyze. Then click on the relevant mediator displayed in the **MESSAGE FLOW** gadget.
- Click on the **API** page and search for the REST API with the mediator you want to analyze. Then click on the relevant mediator displayed in the **MESSAGE FLOW** gadget.
- Click on the **SEQUENCE** page and search for the sequence with the mediator you want to analyze. Then click on the relevant mediator displayed in the **MESSAGE FLOW** gadget.

At any given time, this page displays the statistics for a selected time interval. Make sure you select the required time interval in the following bar displayed at the top of the page. If you want to define a custom time interval, click **Custom** and select the start and end dates of the required time interval in the calendar that appears.

[Last Hour](#) [Last 24 Hours](#) [Last 30 Days](#) [Last Year](#)  [Custom](#)

[Mediator Request Count](#) | [Message Count](#) | [Message Latency](#) | [Messages](#)

### Mediator Request Count

<b>View (Example)</b>	<p>MEDIATOR REQUEST COUNT</p> <p>Total requests for LicenseServiceProxy@19:HeaderMediator:Action</p> <p><b>12</b></p> <p>100.00% 12 Requests</p> <p>0.00% 0 Requests</p> <p>Success Rate Failure Rate</p>
<b>Description</b>	<p>This indicates the total number requests that have been processed by a specific mediator during a selected time interval, the percentage of these requests that were successfully processed.</p>
<b>Purpose</b>	<ul style="list-style-type: none"> <li>• Viewing the request count for different mediators allows you to identify the nature of mediation performed for transactions processed for your applications and services.</li> <li>• By comparing message count patterns for all the mediators included in a specific proxy service/REST API/sequence, you can identify configuration errors that have caused message failure.</li> </ul>
<b>Recommended action</b>	<p>Compare the request count of a mediator with that of the preceding/subsequent mediators in the mediation sequence. If there is a difference, it indicates that an error has occurred in the mediation flow between the two mediators.</p>

### Message Count

<b>View (Example)</b>	
<b>Description</b>	This provides a graphical view of the count for both successful and failed messages that were processed by a specific mediator during a selected time interval.
<b>Purpose</b>	This allows you to identify any correlation that may exist between message failure rate, throughput and time.
<b>Recommended action</b>	If the message failure is particularly high during a specific time, you can check whether any unusual occurrences have taken place during that time (e.g., system downtime, unavailability of the back-end service) and take appropriate action.

## Message Latency

<b>View (Example)</b>	
<b>Description</b>	This provides a graphical illustration of the time spent per message to perform the mediation by the selected mediator over time.
<b>Purpose</b>	This enables you to understand the efficiency with which the mediation is performed by a selected mediator.
<b>Recommended action</b>	If the message latency is persistently high, check the mediator configuration for possible changes that can result in lower latency. You can also check the complete mediation flow of the relevant proxy service/API/sequence, and consider other combination of mediators that can be used to achieve the same mediation end result.

## Messages

<b>View (Example)</b>	
---------------------------	--------------------------------------------------------------------------------------

<b>Description</b>	This provides the list of message IDs that were processed using a mediator. Details including the port, time stamp and whether the message is successfully processed are displayed for each message ID.
<b>Purpose</b>	This allows you to identify the individual messages processed by a specific mediator that have failed, and check whether the mediator has been correctly used for that message (e.g., in combination with other mediators).
<b>Recommended action</b>	Click on a failed message ID to view the message in the <b>OVERVIEW/MESSAGE/&lt;MESSAGE_ID&gt;</b> page. Then click on the mediator (i.e. the one currently being analyzed) in the <b>MESSAGE FLOW</b> gadget. This updates the <b>MEDIATOR PROPERTIES</b> gadget with the payload, transport properties and context properties in the message before and after the mediation was carried out by the mediator. Analyze the differences to identify errors in the mediation performed.

## Analyzing Statistics for Messages

This section explains how to view and analyze statistics relating to WSO2 ESB messages using WSO2 Analytics.

Use one of the following methods to view information in this page.

- When you open the **ESB Analytics** dashboard, the **Overview** page is displayed by default. Click on the proxy service/REST API/sequence/endpoint/inbound endpoint with the message you want to view in the relevant gadget. Then click on the required message ID in the **MESSAGES** gadget.
- Open any of the following pages by clicking on the relevant link in the left navigator and search for the proxy service/REST API/sequence/endpoint/inbound endpoint with the message you want to view. Then click on the required message ID in the **MESSAGES** gadget.
  - **OVERVIEW/PROXY**
  - **OVERVIEW/API**
  - **OVERVIEW/SEQUENCES**
  - **OVERVIEW/ENDPOINT**
  - **OVERVIEW/INBOUNDEDPOINT**

At any given time, this page displays the statistics for a selected time interval. Make sure you select the required time interval in the following bar displayed at the top of the page. If you want to define a custom time interval, click **Custom** and select the start and end dates of the required time interval in the calendar that appears.

Last Hour   Last 24 Hours   **Last 30 Days**   Last Year   Custom ▾

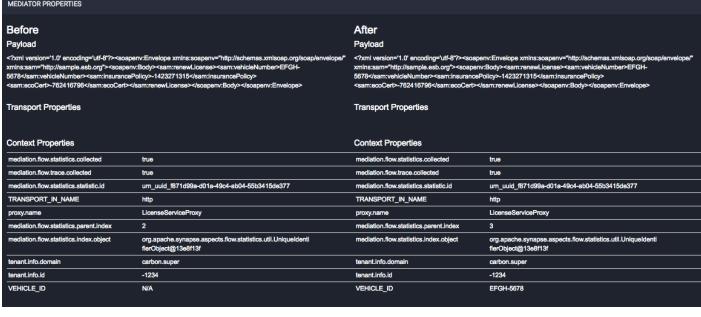
## | Message Flow | Mediator Properties

### Message Flow

<b>View (Example)</b>	
---------------------------	--

<b>Description</b>	This indicates the proxy service that processed a selected message and the mediators that were used to process the message.
<b>Purpose</b>	If a message has failed, this view allows you to understand whether the failure can be attributed to a specific mediator/endpoint in the message flow.
<b>Recommended action</b>	<ul style="list-style-type: none"> <li>Check whether the sequence in which the mediation is performed by different mediators, and the sequence in which the message is sent to different endpoints are correct.</li> <li>Click on each mediator in the message flow and view the <b>MEDIATOR PROPERTIES</b> gadget to identify the changes to the payload made by the mediator as well as the context and transport properties added to/removed from the mediation flow by the mediator. This allows you to evaluate whether the mediation is performed as expected.</li> <li>Click on an endpoint and view the <b>MEDIATOR PROPERTIES</b> gadget to identify the changes to the payload made before the endpoint and after the endpoint.</li> </ul>

## Mediator Properties

<b>View (Example)</b>	 <p>The screenshot shows the Mediator Properties gadget with the following sections:</p> <ul style="list-style-type: none"> <li><b>Before Payload:</b> XML representation of the message before it passes through the mediator.</li> <li><b>After Payload:</b> XML representation of the message after it passes through the mediator.</li> <li><b>Context Properties:</b> A table showing properties like mediation.flow.statistics.collected (true), mediation.flow.trace.collected (true), etc.</li> <li><b>Transport Properties:</b> A table showing properties like TRANSPORT_IN_NAME (Http), proxy.name (LicenseServiceProxy), etc.</li> </ul>
<b>Description</b>	This gadget displays the payload of the selected message before and after the selected mediator/sequence/endpoint. It also shows the context and transport properties in the mediation flow before and after the selected mediator/sequence/endpoint.
<b>Purpose</b>	This gadget allows you to analyze the nature of processing carried out by each ESB artifact included in the message flow that a message has passed through. If a message has failed, this gadget allows you to identify the exact stages in the message flow where the errors have occurred.
<b>Recommended action</b>	Click on the required mediator/endpoint/sequence in the <b>MESSAGE FLOW</b> gadget in order to update this gadget with information about it. Consider the <b>Before</b> section as the input of the selected artifact, and the <b>After</b> section as the output. Check whether the transformation that has been made is the same as what is required.

## Monitoring WSO2 ESB with WSO2 ESB Analytics

This section demonstrates a scenario where WSO2 ESB interacts with a simple Axis2 service when a client sends JSON messages. The ESB artifact used are then monitored by WSO2 Analytics - ESB.

In this scenario, a JSON client needs to connect with a Stockquote service. The client can send requests that contain company names in JSON format. However, the backend server can only serve SOAP messages. WSO2 ESB carries out the mediation so that the client can communicate with the backend server.

WSO2 Analytics collects the statistics for the REST API and the 5 mediators with which the mediation is performed, and presents them in the WSO2 Analytics Dashboard.

Follow the steps below to try this scenario.

- Prerequisites
- Step 1 - Configure WSO2 ESB to publish statistics

- Step 2 - Deploy the required ESB artifacts
- Step 3 - Set up the Stockquote service
- Step 4 - Send JSON requests
- Step 5 - Analyze statistics

#### **Prerequisites**

In order to try this tutorial, you need to download and install the following WSO2 products from here.

- **WSO2 Enterprise Service Bus:** To download this, click [DOWNLOAD](#).
- **WSO2 Analytics - ESB** - To download this, click [Analytics](#).

#### **Step 1 - Configure WSO2 ESB to publish statistics**

This step involves doing the required configurations to ensure that statistics are published in the WSO2 Analytics Dashboard when you invoke the StockQuote service.

Enable mediation statistics and message tracing by setting the following properties in the <ESB\_HOME>/repository/conf/synapse.properties file.

```
mediation.flow.statistics.enable=true
mediation.flow.statistics.tracer.collect.payloads=true
mediation.flow.statistics.tracer.collect.properties=true
```

#### **Step 2 - Deploy the required ESB artifacts**

This step involves uploading the ESB artifacts required for this scenario using a CAR file.

1. Download and save [ESB\\_Artifacts\\_1.0.0.car](#) in a preferred location in your computer.
2. Start the WSO2 ESB Analytics server with one of the following commands.
  - On Windows: <ESB\_ANALYTICS\_HOME>\bin\wso2server.bat --run
  - On Linux/Solaris/Mac OS: sh <ESB\_ANALYTICS\_HOME>/bin/wso2server.sh

The WSO2 ESB Analytics server needs to be started before the WSO2 ESB server in order to be updated with ESB statistics when they are published.
3. Start WSO2 ESB with one of the following commands.
  - On Windows: <ESB\_HOME>\bin\wso2server.bat --run
  - On Linux/Solaris/Mac OS: sh <ESB\_HOME>/bin/wso2server.sh
4. Log into the WSO2 ESB Management Console using the following URL.  
[https://<ESB\\_HOST>:<ESB\\_PORT>/carbon/](https://<ESB_HOST>:<ESB_PORT>/carbon/)
5. In the **Main** tab, click **Add** under **Carbon Applications**. This opens the **Add Carbon Applications** page.
6. Click **Browse**, and browse for the `ESB_Artifacts_1.0.0.car` file you saved in your computer. Then click **Upload**.
7. In the **Main** tab, click **APIs** to open the **Deployed APIs** page. An API named StockQuoteAPI. Enable statistics and tracing for this API by clicking **Enable Statistics** and **Enable Tracing**.

## Deployed APIs

[Add API](#)

Search API

Available defined APIs in the Synapse Configuration : 1

[Select all in this page](#) | [Select none](#)

Delete

Select	API Name	API Invocation URL	Action
<input type="checkbox"/>	StockQuoteAPI	http://10.100.5.73:8280/stockquote	<a href="#">Enable Statistics</a> <a href="#">Enable Tracing</a> Delete

[Select all in this page](#) | [Select none](#)

Delete

### Step 3 - Set up the Stockquote service

Follow the procedure below to start the axis2 server for the Stockquote service that is invoked by the client in this scenario.

1. Issue the following command from the <ESB\_HOME>/samples/axis2Server/src/SimpleStockQuotes service directory.  
ant
2. Issue the following command from the <ESB\_HOME>/samples/axis2Server directory.  
./axis2server.sh

### Step 4 - Send JSON requests

In this step, 40 requests are sent to the Stockquote service. For the purpose of demonstrating how successful messages and message failures are illustrated in the Analytics Dashboard, 10 of the requests are sent while the axis2 server is not running. This should generate a success rate of 75% for the StockQuoteAPI REST API.

1. Issue the following cURL command from your terminal to invoke the StockQuoteAPI REST API and get stock quotes from the Stockquote service. Send 30 requests.

```
curl -X POST http://localhost:8280/stockquote/getQuote -d '{"name": "WSO2"}' -H "Content-Type:application/json"
```

2. Stop the axis2 server that you previously started.
3. Send 10 more requests using the cURL command given above.

### Step 5 - Analyze statistics

The following assumptions are made in this section because the manner in which information in the Analytics Dashboard changes depending on the time gaps between requests sent to the ESB, and the time gap between sending requests and viewing the Analytics Dashboard.

- All 10 requests are sent within one hour, and the Analytics Dashboard is viewed within the same hour.
- The activities described in this use case are the only activities you have carried out in WSO2 ESB during the last hour before you view the Analytics Dashboard.

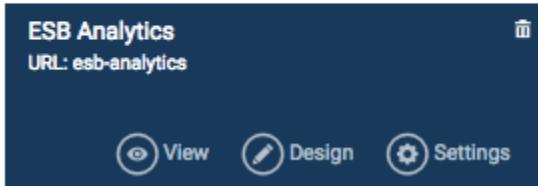
Follow the procedure below to access the WSO2 Analytics Dashboard and analyze the ESB statistics published in it.

1. Log into the WSO2 Analytics - ESB Management Console using the following URL.  
URL: [https://<ESB\\_Analytics\\_HOST>:<ESB\\_Analytics\\_PORT>/carbon/](https://<ESB_Analytics_HOST>:<ESB_Analytics_PORT>/carbon/)
2. In the **Main** tab, click **Analytics Dashboard**. Then log into the Analytics Dashboard by entering admin as

both the username and the password. This opens the **Dashboards** page where the **ESB Analytics** dashboard is displayed as follows.

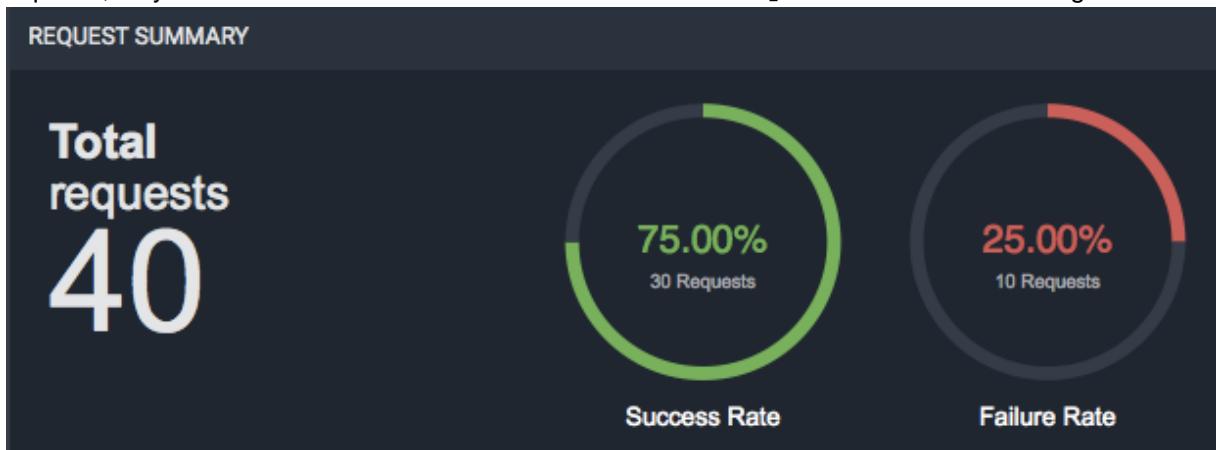
## Dashboards

\* View, Modify and Delete dashboards

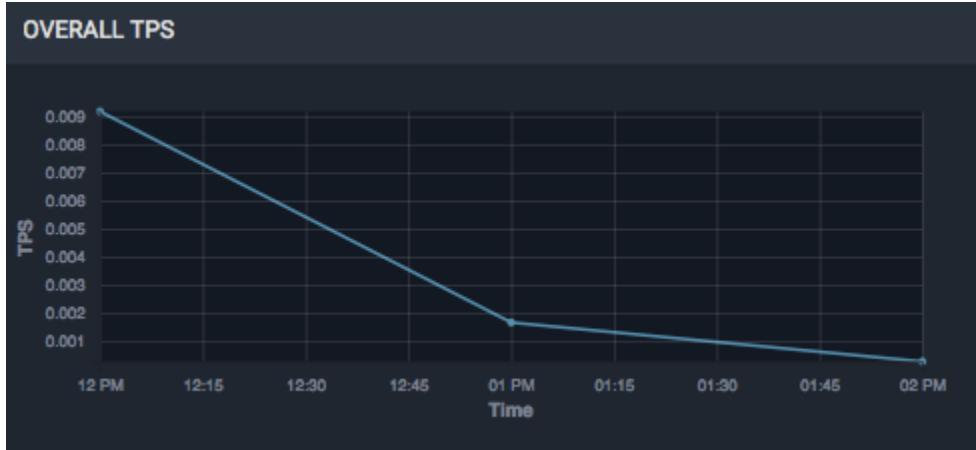


- Click **View** to open the dashboard. Then click **Last Hour**. The following is displayed.

- The request count is displayed as 40. The success rate of 75% is displayed because out of the 40 requests, only 30 were sent while the axis2 server for the Stockquote service was running.



- The number of transactions handled by the ESB per second is mapped on a graph as follows.



- The success rate and the failure rate of the messages received by the ESB during the last hour are mapped in a graph as follows.

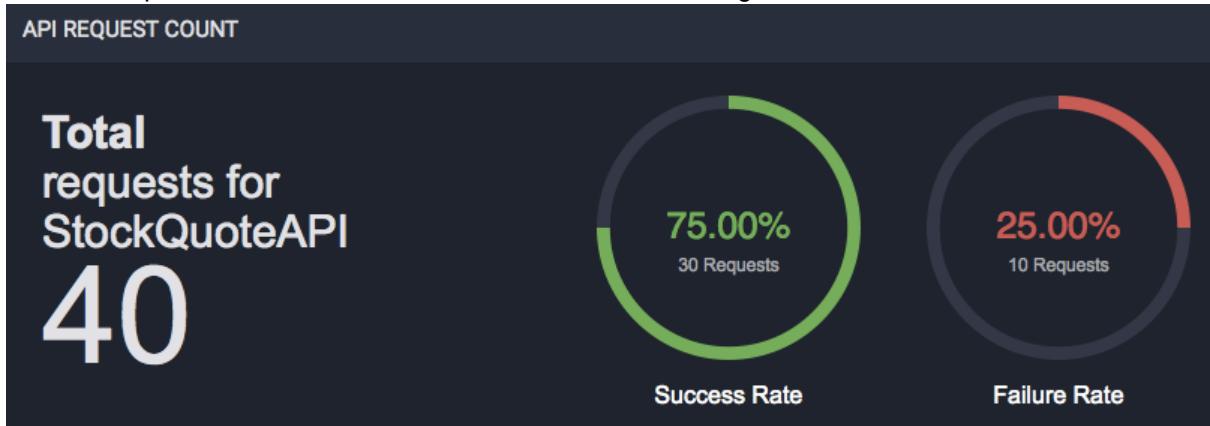


- The StockQuoteAPI REST API is displayed under **TOP APIS BY REQUEST COUNT** as follows.



- In the Top APIS BY Request COUNT gadget, click StockquoteAPI to open the **OVERVIEW/API/STOCKQUOTEAPI** page. The following is displayed.

- The **API REQUEST COUNT** gadget shows the total number of requests handled by the StockQuote API REST API during the last hour as 40. The success rate is displayed as 75% because only 30 out of the 40 requests were sent while the axis2 server was running.



- The **MESSAGE COUNT** gadget maps the number of successful messages as well as failed messages at different times within the last hour in a graph as shown below.



- The **MESSAGE LATENCY** gadget shows the speed with which the messages are processed by mapping the amount of time taken per message at different times within the last hour as shown below.

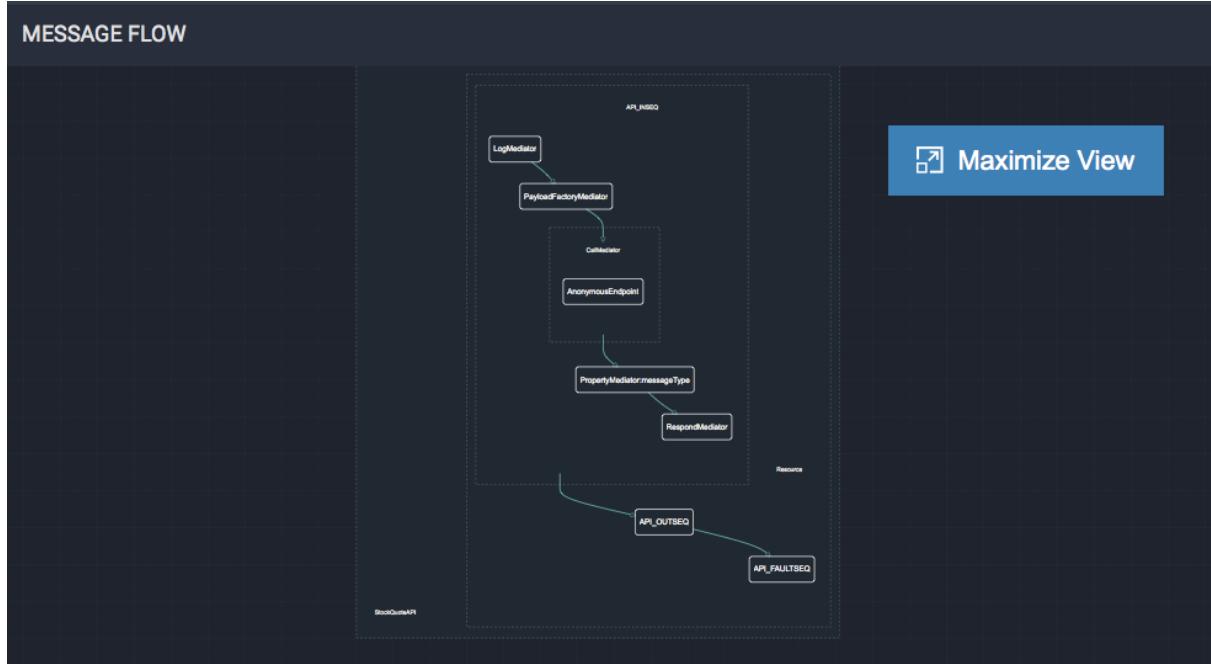


- The **MESSAGES** gadget lists all the the messages handled by the StockQuoteAPI REST API during the last hour as follows.

**MESSAGES**

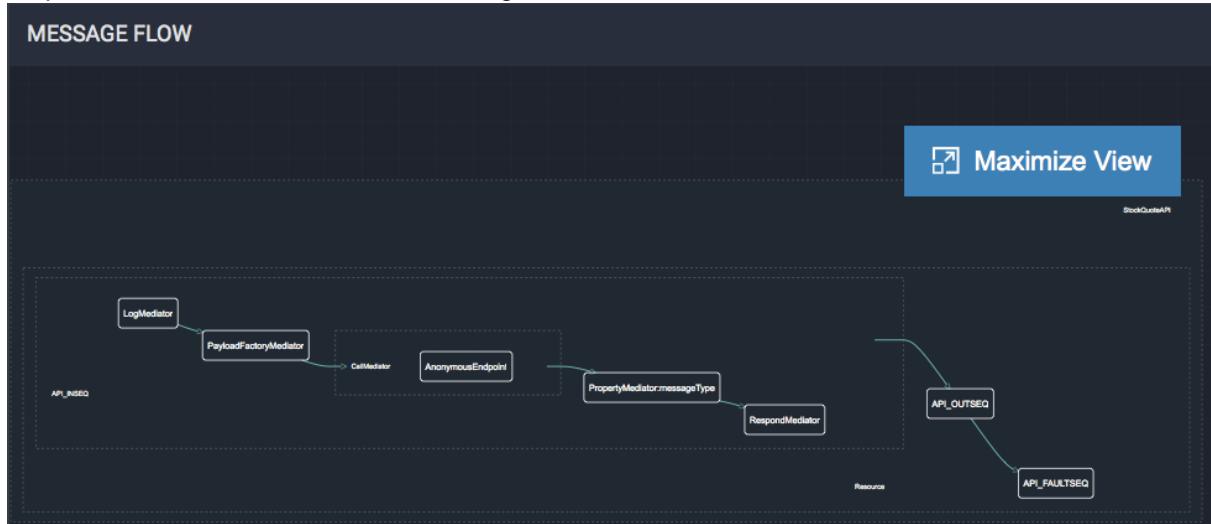
Message ID	Host	Start Time	Status
urn_uuid_00eb7180-e56d-4c0b-9bdc-5b6e3fc1272e	10.100.5.73:9763	June 9, 2016 10:11:05 AM IST	Failed
urn_uuid_0111a430-ca55-44d5-9158-b22a53028fe5	10.100.5.73:9763	June 9, 2016 10:05:42 AM IST	Success
urn_uuid_23bacdd1-a91e-4424-b6f1-19c78eb1e693	10.100.5.73:9763	June 9, 2016 10:03:36 AM IST	Success
urn_uuid_2b1a2ff8-a8df-4471-ac05-d8dd6e79c02e	10.100.5.73:9763	June 9, 2016 10:05:17 AM IST	Success
urn_uuid_3046875a-0ff7-4405-ac62-3ee08815e81f	10.100.5.73:9763	June 9, 2016 10:05:43 AM IST	Success
urn_uuid_322a84c1-ea8c-4f25-9755-3794cd7e9911	10.100.5.73:9763	June 9, 2016 10:05:44 AM IST	Success
urn_uuid_471053bf-f5ad-4136-beeb-4647eb698974	10.100.5.73:9763	June 9, 2016 10:05:47 AM IST	Success
urn_uuid_48003d47-ab13-4e9c-b5bb-06d40f885022	10.100.5.73:9763	June 9, 2016 10:05:10 AM IST	Success
urn_uuid_4cf83c3e-58e1-4e3b-86bb-93cd80d61bff	10.100.5.73:9763	June 9, 2016 10:35:07 AM IST	Failed
urn_uuid_4e248286-9c58-446c-85ca-85ef02503936	10.100.5.73:9763	June 9, 2016 10:05:16 AM IST	Success

- The **MESSAGE FLOW** gadget illustrates the order in which the messages handled by the StockQuote eAPI REST API within the last hour passed through all the mediation sequences, mediators and endpoints that were included in the message flow as shown below.



- Click on a message ID in the **MESSAGES** gadget to open the **OVERVIEW/MESSAGE/<MESSAGE\_ID>** page. The following is displayed.

- The **MESSAGE FLOW** gadget illustrates the order in which the messages handled by the StockQuote eAPI REST API within the last hour passed through all the mediation sequences, mediators and endpoints that were included in the message flow as shown below.



- Click **PayloadFactoryMediator** in the **MESSAGE FLOW** gadget. As a result, the **MEDIATOR PROPERTIES** gadget illustrates the processing carried out by the Payload Factory mediator selected by displaying the payload, transport properties and context properties in the message flow both before and after the mediator as shown below.

MEDIATOR PROPERTIES		
	Before	After
Payload	[{"name": "WSO2"}]	<?xml version='1.0' encoding='utf-8'?><soapenv:Envelope xmlns:soapenv='http://schemas.xmlsoap.org/soap/envelope'/><soapenv:Body><ser:getQuote xmlns:ser='http://services.samples' xmlns:xsd='http://services.samples/xsd'><ser:request><xsd:symbol>WSO2</xsd:symbol></ser:request></ser:getQuote></soapenv:Body></soapenv:Envelope>
Transport Properties	Accept: */*	Accept: */*
	User-Agent: curl/7.30.0	User-Agent: curl/7.30.0
	Host: localhost:8280	Host: localhost:8280
	To: /stockquote/getQuote	To: /stockquote/getQuote
	Content-Length: 15	Content-Length: 15
	Content-Type: application/json	Content-Type: application/xml
	MessageID: urn:uuid:0caf14f0-fe97-4f1d-b0ce-cf9009276e31	MessageID: urn:uuid:0caf14f0-fe97-4f1d-b0ce-cf9009276e31
Context Properties	mediation.flow.statistics.statistic.id: urn_uuid_0caf14f0-fe97-4f1d-b0ce-cf9009276e31	mediation.flow.statistics.statistic.id: urn_uuid_0caf14f0-fe97-4f1d-b0ce-cf9009276e31
	REST_METHOD: POST	REST_METHOD: POST
	SYNAPSE_REST_API: StockQuoteAPI	SYNAPSE_REST_API: StockQuoteAPI
	mediation.flow.statistics.parent.index: 4	mediation.flow.statistics.parent.index: 5
	REST_FULL_REQUEST_PATH: /stockquote/getQuote	REST_FULL_REQUEST_PATH: /stockquote/getQuote
	rest.url.pattern: /getQuote	rest.url.pattern: /getQuote
	mediation.flow.statistics.collected: true	mediation.flow.statistics.collected: true

## Extending ESB Analytics

The following topics cover the different ways in which the ESB Analytics feature can be extended.

- [Customizing Statistics Publishing](#)
- [Monitoring JMX Based Statistics](#)

### Customizing Statistics Publishing

The following needs to be done in order to implement a custom statistics observer.

- To disable operation of publishing statistics via the ESB Analytics server by setting the `AnalyticPublishingDisable` property in the `<ESB_HOME>/repository/conf/carbon.xml` file to `true` as shown below.

```
<MediationFlowStatisticConfig>
<AnalyticPublishingDisable>true</AnalyticPublishingDisable>
</MediationFlowStatisticConfig>
```

This is useful when you want to collect statistics using a product other than ESB Analytics. In such scenarios, preventing the ESB Analytics server from publishing statistics allows you to avoid incurring an unnecessary system overhead.

- Create the JAR file with the that contains the required statistics observer implementation, and copy it to the `<ESB_HOME>/repository/components/lib` directory. This implementation should have the following.
  - The `org.wso2.carbon.das.messageflow.data.publisher.observer.MessageFlowObserver` interface.
  - If the observer is tenant aware, it should also implement `org.wso2.carbon.das.messageflow.data.publisher.observer.TenantInformation`.
- Add all the observers as a comma separated list in the `<ESB_HOME>/repository/conf/carbon.xml` file as shown in the example below.

```
<MediationFlowStatisticConfig>
<Observers>
 org.wso2.custom.Observer1,org.wso2.custom.Observer2
</Observers>
</MediationFlowStatisticConfig>
```

### Monitoring JMX Based Statistics

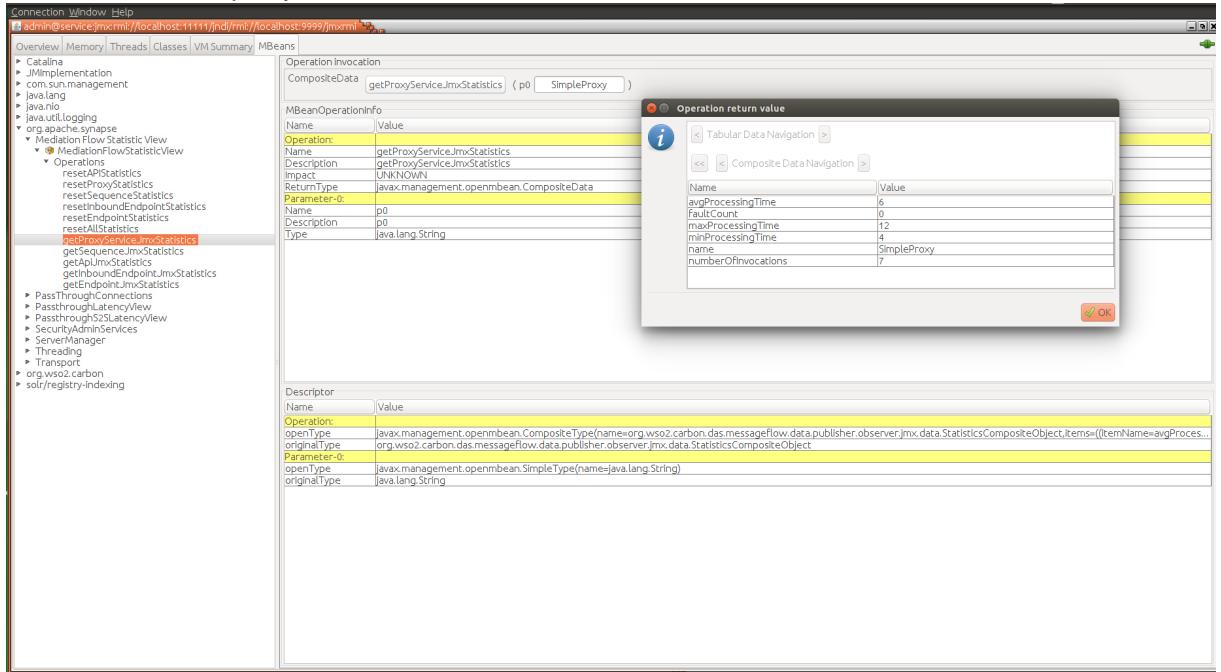
WSO2 ESB is shipped with inbuilt support for JMX (Java Management Extension) monitoring. JMX is a common method to manage and monitor the runtime parameters of a remote server. Follow the steps below to view JMX based statistics for WSO2 ESB.

1. Enable mediation flow statistics in the <ESB\_HOME>/repository/conf/synapse.properties file as shown below.

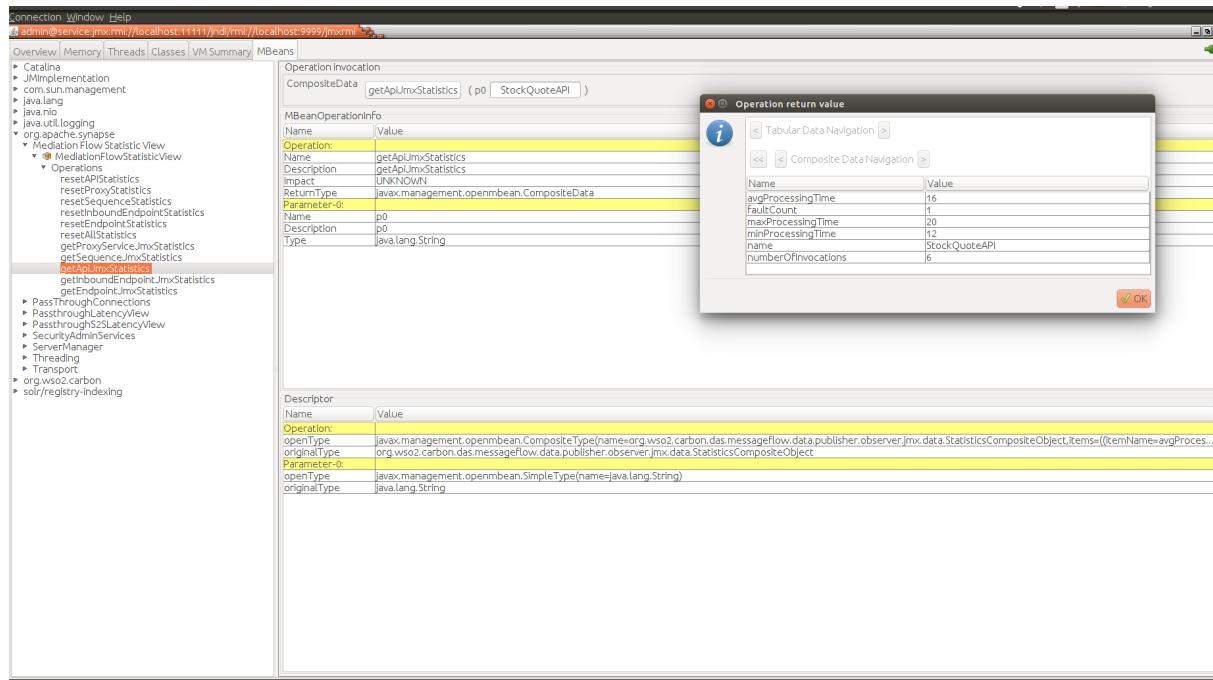
```
mediation.flow.statistics.enable=true
mediation.flow.statistics.tracer.collect.payloads=true
mediation.flow.statistics.tracer.collect.properties=true
```

2. Start the WSO2 ESB server. For more information, see [Running the Product](#).
3. Open the J Console in your computer and connect to the ESB server.
4. Navigate to org.apache.synapse => Mediation Flow Statistics View => Operation. Click on the required ESB artifact and do the relevant operation invocation. JMX statistics relating to proxy services, REST APIs, inbound endpoints, endpoints and sequences can be viewed as shown in the examples below.

- JMX statistics for a proxy service



- JMX statistics for a REST API



## Disabling the JMX observer

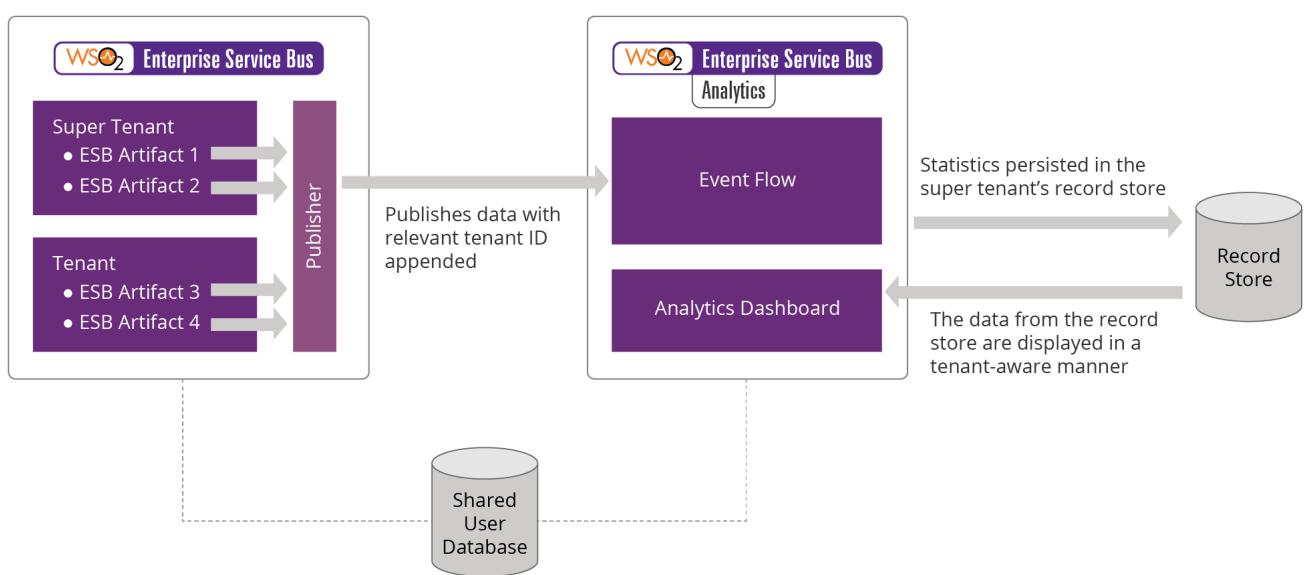
The JMX observer can be disabled by adding the following property to the <ESB\_HOME>/repository/conf/carbon.xml file.

```
<MediationFlowStatisticConfig>
 <JmxPublishingDisable>true</JmxPublishingDisable>
</MediationFlowStatisticConfig>
```

## Monitoring WSO2 ESB with WSO2 Analytics in a Multi-tenant Environment

This section demonstrates how ESB Analytics is used to analyze statistics relating to ESB artifacts in a multi-tenanted ESB environment.

## Architecture



- In a multi tenanted environment, all the tenants share a single event publisher that publishes data to the WSO2 ESB Analytics server. This publisher uses the credentials of the super tenant for authentication purposes.
- A single event receiver collects all the statistics from the ESB artifacts of all tenants, and persists them in the database configured as the super tenant's record store for ESB Analytics.
- The user database can be shared between the ESB Analytics and the ESB server to avoid defining the same user IDs in both environments.
- When you log into the WSO2 ESB Analytics Dashboard with the super tenant credentials, you can view statistics relating to the super tenant.
- When you log into the WSO2 ESB Analytics Dashboard with the credentials of a tenant other than the super tenant, you can only view statistics relating to the ESB artifacts defined in the environment of that tenant.

Before you follow the steps given below, enable analytics for WSO2 ESB following the instructions given in [Prerequisites to Publish Statistics](#).

1. Configure a MySQL database as the user database for WSO2 ESB WSO2 ESB Analytics following the procedure given below. As a result, the user database is shared between WSO2 ESB Analytics and WSO2 ESB. Therefore, once you define the `tenant1` tenant in WSO2 ESB, that tenant can also access WSO2 ESB Analytics with the same credentials used to access WSO2 ESB.
  - a. Download and install [MySQL Server](#).
  - b. Download the [MySQL JDBC driver](#).
  - c. Unzip the downloaded MySQL driver zipped archive, and copy the MySQL JDBC driver JAR (`mysql-connector-java-x.x.xx-bin.jar`) into the `<PRODUCT_HOME>/repository/components/lib` directory of both WSO2 ESB Analytics and WSO2 ESB.
  - d. Define the host name for configuring permissions for the new database by opening the `/etc/hosts` file and adding the following line:  
`<MYSQL-DB-SERVER-IP> carbondb.mysql-wso2.com`

This step is required only if your database is not on your local machine and on a separate server.

- e. Enter the following command in a terminal/command window, where `username` is the username you want to use to access the databases:  
`mysql -u username -p`

- f. When prompted, specify the password that should be used to access the databases with the username you specified.
- g. Create the databases using the following commands, where <PRODUCT\_HOME> is the path to any of the product instances you installed, and `username` and `password` are the same as those you specified in the previous steps.

## About using MySQL in different operating systems

For users of Microsoft Windows, when creating the database in MySQL, it is important to specify the character set as latin1. Failure to do this may result in an error (error code: 1709) when starting your cluster. This error occurs in certain versions of MySQL (5.6.x) and is related to the UTF-8 encoding. MySQL originally used the latin1 character set by default, which stored characters in a 2-byte sequence. However, in recent versions, MySQL defaults to UTF-8 to be friendlier to international users. Hence, you must use latin1 as the character set as indicated below in the database creation commands to avoid this problem. Note that this may result in issues with non-latin characters (like Hebrew, Japanese, etc.). The database creation command should look as follows.

```
mysql> create database <DATABASE_NAME> character set latin1;
```

For users of other operating systems, the standard database creation commands can be used. For these operating systems, your database creation command should look as follows.

```
mysql> create database <DATABASE_NAME>;
```

```
mysql> create database WSO2_CARBON_DB;
mysql> use WSO2_CARBON_DB;
mysql> source <PRODUCT_HOME>/dbscripts/mysql.sql;
mysql> grant all on WSO2_CARBON_DB.* TO
wso2carbon@"carbondb.mysql-wso2.com" identified by "wso2carbon";
```

- h. Add the URL of the `WSO2_CARBON_DB` with the URL of the newly created database in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file of both WSO2 ESB and WSO2 ESB Analytics as shown in the extract below.

```

<datasources-configuration
xmlns:svns="http://org.wso2.securevault/configuration">
 <providers>

 <provider>org.wso2.carbon.ndatasource.rdbms.RDBMSDataSourceReader</provider>
 </providers>
 <datasources>
 <datasource>
 <name>WSO2_CARBON_DB</name>
 <description>The datasource used for registry and user manager</description>
 <jndiConfig>
 <name>jdbc/WSO2CarbonDB</name>
 </jndiConfig>
 <definition type="RDBMS">
 <configuration>

 <url>jdbc:mysql://carbondb.mysql-wso2.com:3306/WSO2_CARBON_DB?autoReconnect=true</url>
 <username>wso2carbon</username>
 <password>wso2carbon</password>

 <driverClassName>com.mysql.jdbc.Driver</driverClassName>
 <maxActive>50</maxActive>
 <maxWait>60000</maxWait>
 <testOnBorrow>true</testOnBorrow>
 <validationQuery>SELECT 1</validationQuery>
 <validationInterval>30000</validationInterval>
 </configuration>
 </definition>
 </datasource>
 </datasources>
 </datasources-configuration>

```

2. Start the WSO2 ESB Analytics server, and then start the WSO2 ESB server. The following command can be issued from the <PRODUCT\_HOME>/bin directory to start each server.
  - a. On Windows: wso2server.bat --run
  - b. On Linux/Mac OS: sh wso2server.sh
3. Access the WSO2 ESB Management Console with the following URL, and log in by entering admin as both the user name and the password.  
[https://<ESB\\_HOST>:<ESB\\_PORT>/carbon](https://<ESB_HOST>:<ESB_PORT>/carbon)
4. In the **Configure** tab, click **Add New Tenant** to open the **Register a New Organization** page. Then enter information as follows and save.

Parameter	Value
<b>Domain</b>	def.com
<b>Select Usage Plan for Tenant</b>	Demo
<b>First Name</b>	DEF
<b>Last Name</b>	Networks

<b>Admin Username</b>	tenant1
<b>New Admin Password</b>	tenant1
<b>Email</b>	deftenant1@gmail.com

5. Deploy a proxy service to WSO2 ESB for the super tenant following the procedure.
  - a. Download and save [this file](#) in a preferred location in your machine.
  - b. Click on the **Main** tab. Under **Carbon Applications**, click **Add** to open the **Add Carbon Applications** page. Click **Choose File** and browse for the file you downloaded and saved in the previous sub step. Then click **Upload**.
6. Click on the **Main** tab. Under **Services**, click **List**. A proxy service named **SimpleProxy** is listed as follows.

Home > Manage > Services > List      [Help](#)

**Deployed Services**

4 active services. 4 deployed service group(s).

Service Type	ALL	Service	Search														
<a href="#">Select all in this page</a>   <a href="#">Select none</a> <a href="#">Delete</a>																	
<b>Services</b>																	
<input type="checkbox"/>	echo		axis2		Unsecured		WSDL1.1		WSDL2.0		Try this service		Download		Design View		Source View
<input type="checkbox"/>	<b>SimpleProxy</b>		proxy		Unsecured		WSDL1.1		WSDL2.0		Try this service						
<input type="checkbox"/>	Version		axis2		Unsecured		WSDL1.1		WSDL2.0		Try this service		Download				
<input type="checkbox"/>	wso2carbon-sts		sts		Unsecured		WSDL1.1		WSDL2.0								

[Select all in this page](#) | [Select none](#)      [Delete](#)

- a. Click on this proxy service, and then click on **Enable Tracing** to enable both statistics and tracing.

Home > Manage > Services > List > Service Dashboard      [Help](#)

**Service Dashboard (SimpleProxy)**

<b>Service Details</b>	<b>Client Operations</b>
Service Name SimpleProxy	<a href="#">Try this service</a>
Service Description echo	<a href="#">Generate Axis2 Client</a>
Service Group Name SimpleProxy	WSDL1.1       WSDL2.0
Deployment Scope request	
Service Type proxy	
<b>Operations</b>	
<input checked="" type="checkbox"/> Active <a href="#">Deactivate</a>	
<b>Quality of Service Configuration</b>	<b>Statistics</b>
<b>Specific Configuration</b>	Request Count 0
<a href="#">Edit</a> <a href="#">Enable Statistics</a>	Response Count 0
<a href="#">Redeploy</a> <a href="#">Enable Tracing</a>	Fault Count 0
	Maximum Response Time 0 ms
	Minimum Response Time 0 ms
	Average Response Time 0.0 ms

- b. Click **Try this service** to open the **Try it Tool**, and send the following request 5 times.

```
<body>
<p:echoInt xmlns:p="http://echo.services.core.carbon.wso2.org">
 <!-- 0 to 1 occurrence-->
 <in>1234</in>
</p:echoInt>
</body>
```

- Log out of the ESB Management Console, and log in again with the following credentials.

**User Name:** tenant1@def.com

**Password:** tenant1

- Deploy a proxy service to WSO2 ESB for the tenant1 tenant following the procedure.

- Download and save [this file](#) in a preferred location in your machine.
- Click on the **Main** tab. Under **Carbon Applications**, click **Add** to open the **Add Carbon Applications** page. Click **Choose File** and browse for the file you downloaded and saved in the previous sub step. Then click **Upload**.

- Click on the **Main** tab. Under **Services**, click **List**. A proxy service named EchoService is listed as follows.

Services	
<input checked="" type="checkbox"/>	EchoService
<input type="checkbox"/>	wso2carbon-sts
	proxy
	sts
	Unsecured
	WSDL1.1
	WSDL2.0
	<a href="#">Try this service</a>
	<a href="#">Design View</a>
	<a href="#">Source View</a>

- Click on this proxy service, and then click on **Enable Tracing** to enable both statistics and tracing.

Service Details	
Service Name	EchoService
Service Description	echo
Service Group Name	EchoService
Deployment Scope	request
Service Type	proxy

Client Operations	
<a href="#">Try this service</a>	<a href="#">Generate Axis2 Client</a>
<a href="#">WSDL1.1</a>	<a href="#">WSDL2.0</a>

Endpoints	
<a href="https://Rukshanis-MacBook-Air.local:8243/services/t/def.com/EchoService">https://Rukshanis-MacBook-Air.local:8243/services/t/def.com/EchoService</a>	
<a href="http://Rukshanis-MacBook-Air.local:8280/services/t/def.com/EchoService">http://Rukshanis-MacBook-Air.local:8280/services/t/def.com/EchoService</a>	

Operations	
<input checked="" type="checkbox"/> Active	<a href="#">[ Deactivate ]</a>

Quality of Service Configuration	
----------------------------------	--

Specific Configuration	
<a href="#">Edit</a>	<a href="#">Enable Statistics</a>
<a href="#">Redeploy</a>	<a href="#">Enable Tracing</a>

Statistics	
Request Count	0
Response Count	0
Fault Count	0
Maximum Response Time	0 ms
Minimum Response Time	0 ms
Average Response Time	0.0 ms

- Click **Try this service** to open the **Try it Tool**, and send the following request 5 times.

```
<body>
<p:echoInt xmlns:p="http://echo.services.core.carbon.wso2.org">
 <!--0 to 1 occurrence-->
 <in>1234</in>
</p:echoInt>
</body>
```

- Access the WSO2 ESB Analytics Dashboard with the following URL, and log in entering admin as both the username and the password.

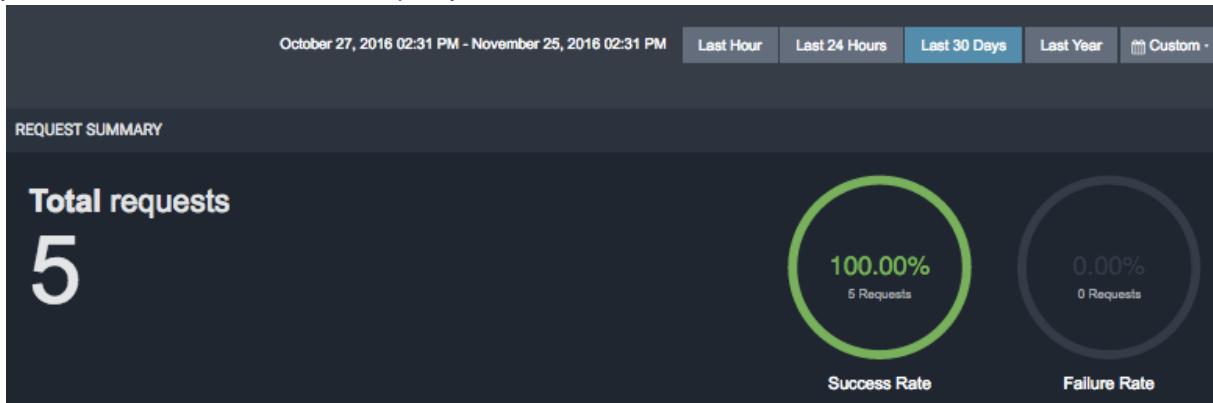
[https://<ANALYTICS\\_HOST>:<ANALYTICS\\_PORT>/portal/dashboards/esb-analytics/](https://<ANALYTICS_HOST>:<ANALYTICS_PORT>/portal/dashboards/esb-analytics/)

- In the **ESB Analytics** dashboard, click **View** to open the pages displaying ESB statistics.

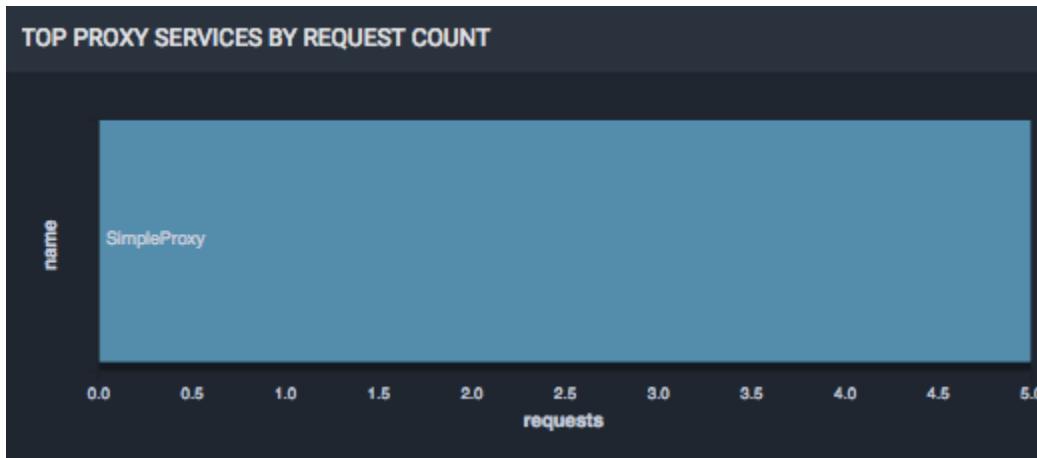


The following can be observed.

- The request summary displays a request count of 5 as shown below. This represents the 5 requests you sent from the SimpleProxy proxy service.



- The SimpleProxy service is displayed in the **TOP PROXY SERVICES BY REQUEST COUNT** gadget.



- Log out of the ESB Analytics Dashboard and log in again with the following credentials.

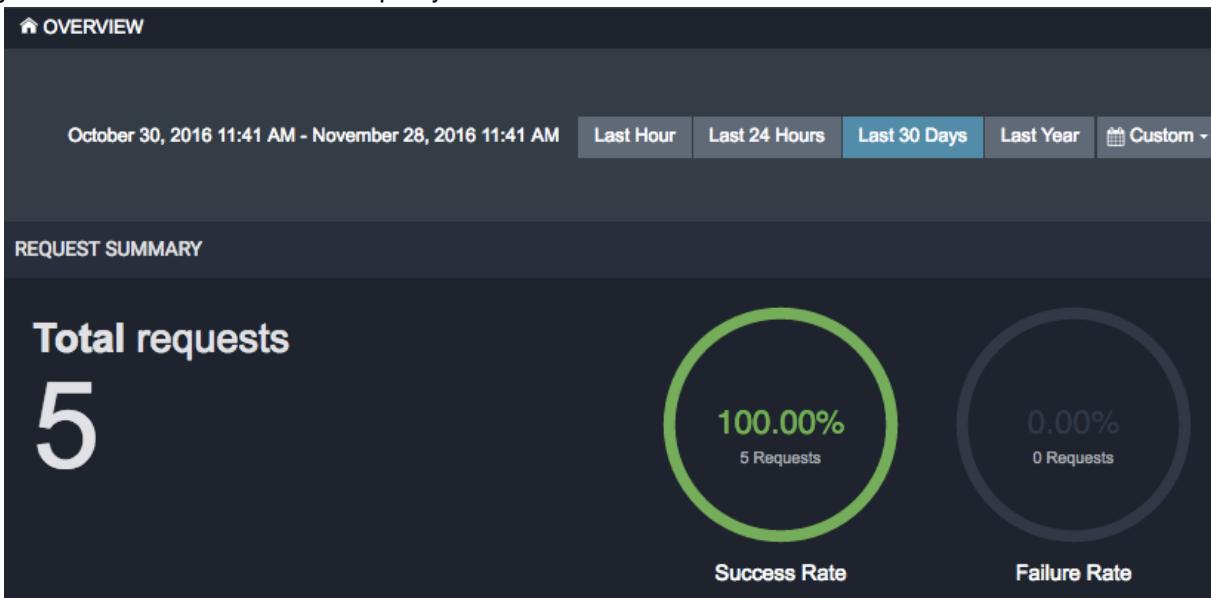
**User Name:** tenant1@def.com

**Password:** tenant1

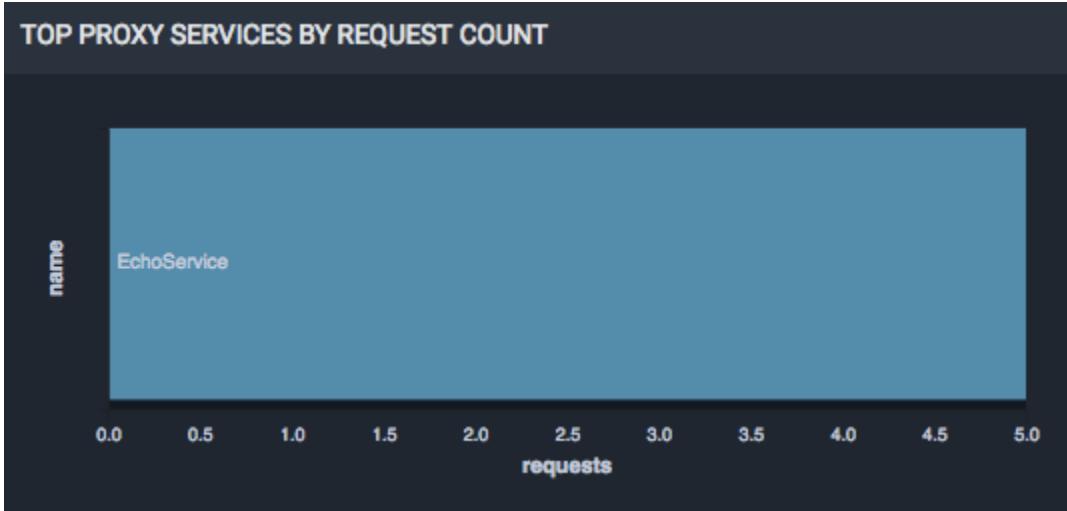
13. In the **ESB Analytics** dashboard, click **View** to open the pages displaying ESB statistics.

The following can be observed.

- The request summary displays a request count of 5 as shown below. This represents the 5 requests you sent from the EchoService proxy service.



- The EchoService proxy service is displayed in the **TOP PROXY SERVICES BY REQUEST COUNT** gadget.



## Cloud Services Gateway (CSG)

**Cloud Services Gateway (CSG)** can be used to expose a private service to the public through a cooperative firewall ([New in version 4.0](#)). CSG consists of two parts:

- CSG Agent** - This feature lets you add/edit CSG servers and publish/unpublished services to a remote CSG server. This feature should be installed in WSO2 Application server.
- CSG** - This feature runs on WSO2 ESB. It requires Qpid component to run on the system.

### Tip

WSO2 ESB is delivered with an built-in Qpid server.

Once installed successfully, you can see the following log line on the WSO2 ESB log:

[ 2011-04-28 12:51:49 ,831 ] INFO - CSGComponent Activated the CSG Component

## Note

If the component starting fails, you will get a detail message why it has happened (most of the time because of missing [Qpid broker](#) component).

Most of the cooperate networks are secured with cooperate firewalls. There are services deployed in the private network that can only be accessed within the organization. If the user wants to access one of these services outside the organization, the firewall will not allow to do that. However, CSG can be useful in this situation.

When the CSG component is installed within ESB, the user can deploy a [proxy service](#) which will act as a front end for the private services. This minimal Proxy has a very specific feature. The Proxy will have a JMS endpoint and the private service (when published) will start listen to this JMS queue. This queue will use the CSG as the communication medium for the front end proxy and the back end private service.

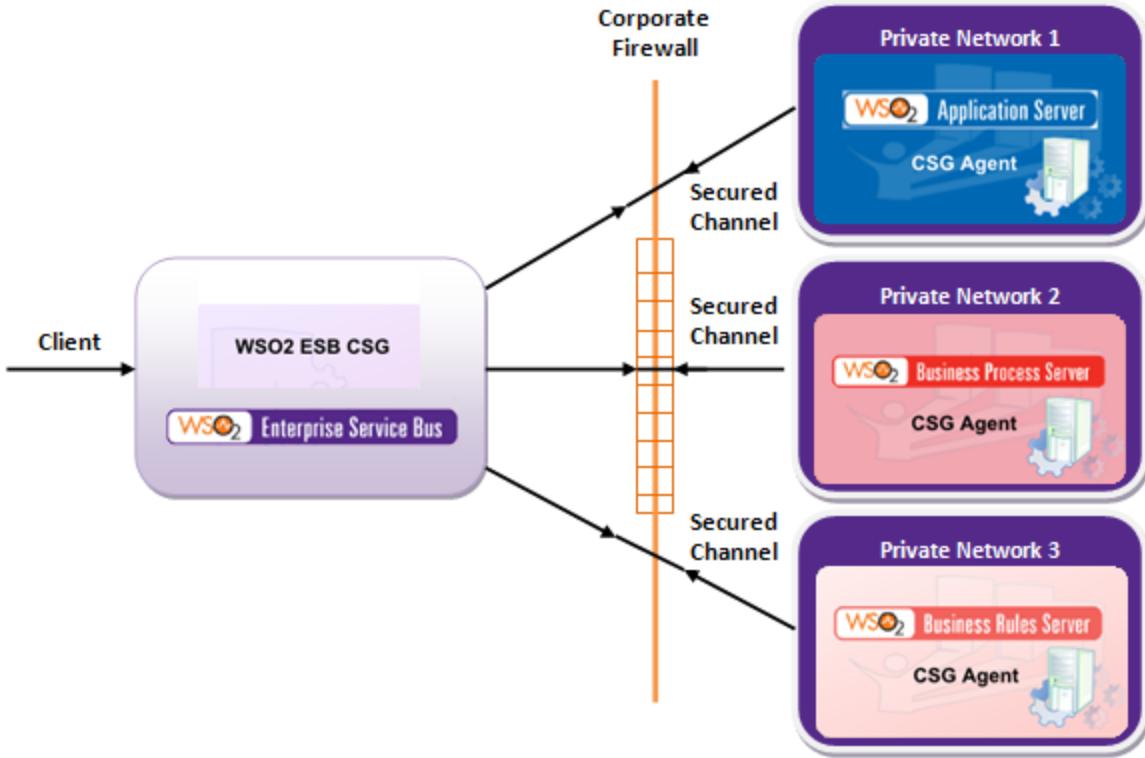
The technique that is used to publish private to public is very simple. When you are within the firewall, you can access the outside but not from outside to inside. When you publish a private service, that service will act as the client and will always make the connection to internet from the cooperate firewall. Since the firewall allows connection from inside to outside, there is no problem to publish services and use them outside. Once published, a Proxy Service will be deployed on ESB that you can use.

The deploying Proxy has the following name: "private service nameProxy". For example, a back end echo service with the service name "Echo" will have Proxy name "EchoProxy."

### Deployment of CSG

When installing CSG will be installed on WSO2 ESB and CSG agent will be available in all service hosting products.

When deploying CSG the private services which will be hosted on one of the service hosting products (such as AS, BPS, BRS, MS etc..) and will be deployed behind the corporate firewall. The ESB which contains the CSG component will be deployed in a place where publicly accessible.

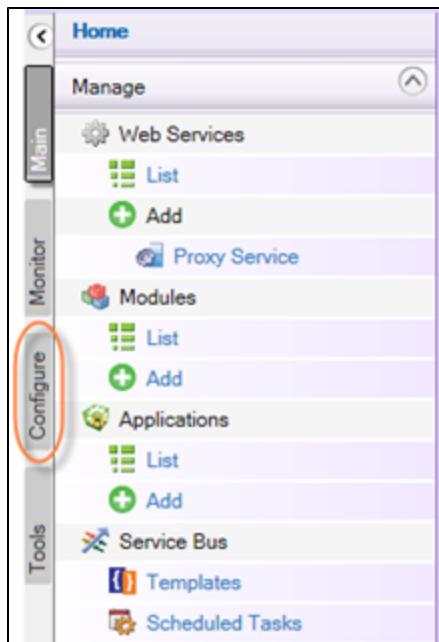


The user can [add a new CSG server](#), publish and [unpublish CSG services](#).

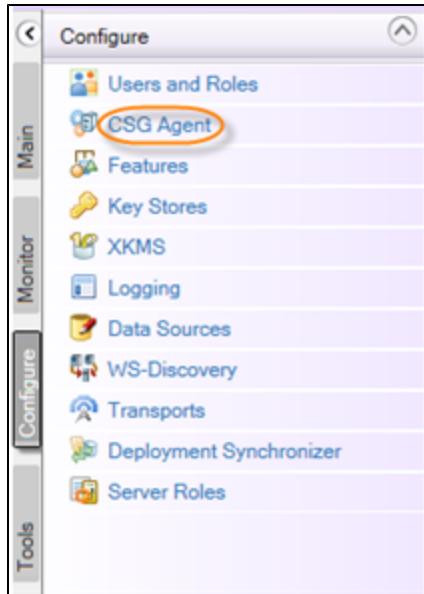
### Adding a New CSG Server

Follow the instructions below to add a new CSG server to WSO2 ESB.

1. Sign in. Enter your user name and password to log on to the ESB Management Console.
2. Select the "Configure" tab on the left vertical menu to access the "Configure" menu.



3. In the "Configure" menu, click "CSG Agent."



4. The "Cloud Services Gateway Agent" page appears. Click on the "Add New CSG Server" link.

The screenshot shows the 'Cloud Services Gateway Agent' page. The URL in the address bar is 'Home > Configure > CSG Agent'. The main content area displays the message 'No CSG server is currently defined.' Below this, there is a button labeled '+ Add New CSG Server' which is highlighted with a red circle.

5. The "Add New CSG Server" window appears.

The screenshot shows the 'Add New CSG Server' configuration window. It has a header 'Add New CSG Server' and a section titled 'CSG Server Information'. The form contains the following fields:

- CSG Server Name \* (input field)
- CSG Host Name \* (input field)
- CSG Host Port \* (input field)
- Domain Name \* (input field)
- User Name \* (input field)
- Password \* (input field)

At the bottom are 'Save' and 'Cancel' buttons.

6. Fill in the required fields:

- **CSG Server Name** - A unique name for the server so that later you can select the server to publish the service.
- **CSG Host Name** - The host name of the remote WSO2 ESB server, for example, "localhost."
- **CSG Host Port** - The port of the remote WSO2 ESB server, for example, "9443."
- **Domain Name** - The domain to which this private service will be exposed, for example, "test.org," if the domain name is not given a standalone product deployment will be carried out.

- **User Name** - The user name of the remote WSO2 ESB user, for example, "test" (tenant login will look like test@test.org)
- **Password** - The password of the remote WSO2 ESB user, for example, "test123."

7. Click "Save."

Once this server is added, it will be listed on the CSG server listing page. From there, you can [edit](#) the already added servers or to remove any servers.

Once a server is added, you can [publish](#) the already deployed services to those servers.

### Configuring CSG Properties

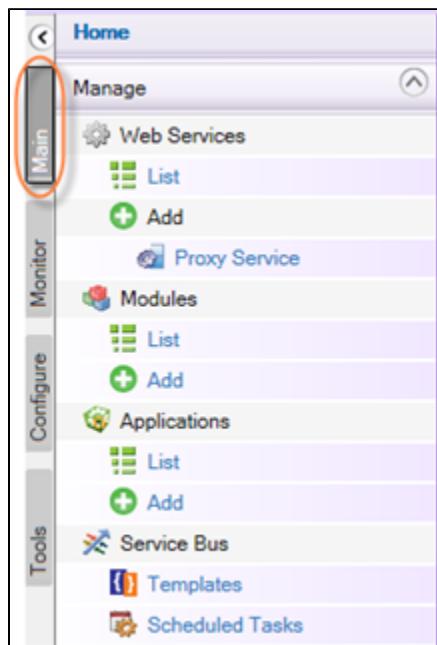
You can configure several properties of **CSG** (server and agent components) using a property file called `csg.properties` which should be placed either in CARBON classpath or inside `$CARBON_HOME/repository/conf`. Following properties can be used:

- `csg.agent.connection.read_timeout` - When CSG Agent component downloading the WSDL the read time out.Default is 100000 milliseconds.
- `csg.agent.connection.connect_timeout` - When CSG Agent component downloading the WSDL the connect time out.Default is 200000 milliseconds.
- `csg.jms.wait_reply` - How long CSG server component Proxy should wait for a reply listing on the JMS queue.Default is 30000 milliseconds.

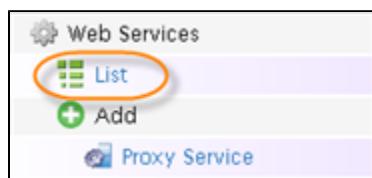
### Publishing a Service to CSG Server

Follow the instructions below to publish a service to [CSG Server](#).

1. Sign in. Enter your user name and password to log on to the ESB Management Console.
2. Click on "Main" in the left menu to access the "Manage" menu.



3. In the "Manage" menu, click on "List" under "Web Services."



4. The "Deployed Services" page appears. Click on the service you want to add to CSG Server.

## Deployed Services

6 active services. 6 deployed service group(s).

Service Type ALL Service

Select all in this page | Select none

Services						
<input type="checkbox"/>	echo		Unsecured			
<input type="checkbox"/>	Proxy Service		Unsecured			
<input type="checkbox"/>	PS2		Unsecured			
<input type="checkbox"/>	Version		Unsecured			
	wso2carbon-sts		Unsecured			
	XKMS		Unsecured			

5. Under the "CSG Configuration," select the server that you want to publish this service to. The drop-down menu lists the added CSG servers.

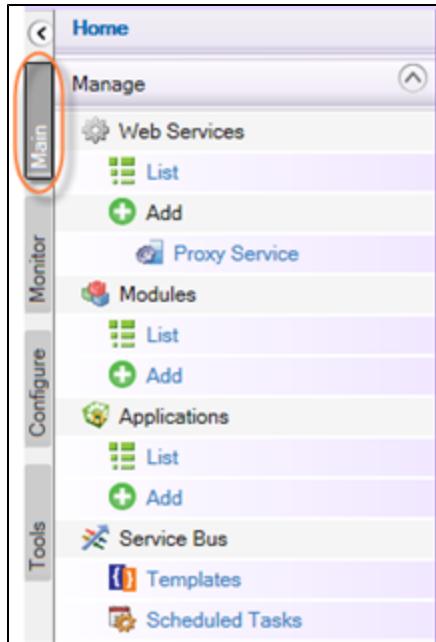
The screenshot shows the 'Manage' section of the ESB Management Console. On the left, there's a sidebar with icons for Active, Security, Reliable Messaging, Response Caching, Access Throttling, MTOM, Policies, Transports, Modules, Operations, and Parameters. Below this is a 'CSG Configuration' section. In the 'Publish To CSG Server' field, a dropdown menu is open with the option 'Select Server' highlighted by a red oval.

## Unpublishing a Service from CSG Server

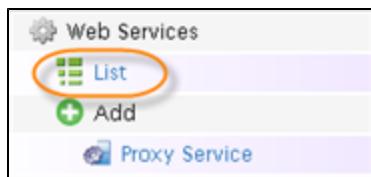
You can unpublish a published service from the **CSG Server**. An unpublished service can be published again if required.

Follow the instructions below to unpublish a published service from the CSG Server.

1. Sign in. Enter your user name and password to log on to the ESB Management Console.
2. Click on "Main" in the left menu to access the "Manage" menu.



3. In the "Manage" menu, click on "List" under "Web Services."

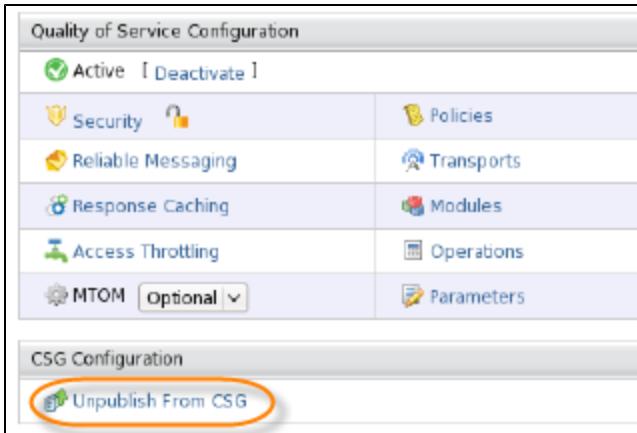


4. The "Deployed Services" page appears. Click on the service you want to unpublish from CSG Server.

The screenshot shows the 'Deployed Services' page. At the top, it displays '6 active services. 6 deployed service group(s.)'. Below this are search and filter controls: 'Service Type ALL' dropdown, 'Service' input field, and a magnifying glass icon. There are also links for 'Select all in this page' and 'Delete'. The main area is a table titled 'Services' with the following data:

	Service Name	Type	Status	WSDL1.1	WSDL2.0	Action
<input type="checkbox"/>	echo	axis2	Unsecured	WSDL1.1	WSDL2.0	<a href="#">Try this service</a>
<input type="checkbox"/>	Proxy Service	proxy	Unsecured	WSDL1.1	WSDL2.0	<a href="#">Try this service</a>
<input type="checkbox"/>	PS2	proxy	Unsecured	WSDL1.1	WSDL2.0	<a href="#">Try this service</a>
<input type="checkbox"/>	Version	axis2	Unsecured	WSDL1.1	WSDL2.0	<a href="#">Try this service</a>
	wso2carbon-sts	sts	Unsecured	WSDL1.1	WSDL2.0	
	XKMS	axis2	Unsecured	WSDL1.1	WSDL2.0	

5. Under "CSG Configuration," click on the "Unpublish from CSG" link.



## Error Handling

The main role of an Enterprise Service Bus (ESB) is to act as the backbone of an organization's service-oriented architecture. It is the spine through which all the systems and applications within the enterprise (and external applications that integrate with the enterprise) communicate with each other. As such, an ESB often has to deal with many wire level protocols, messaging standards, and remote APIs. But applications and networks can be full of errors. Applications crash. Network routers and links get into states where they cannot pass messages through with the expected efficiency. These error conditions are very likely to cause a fault or trigger a runtime exception in the ESB.

### Using fault sequences

WSO2 ESB provides fault sequences for dealing with errors. A fault sequence is a collection of mediators just like any other sequence, and it can be associated with another sequence or a proxy service. When the sequence or the proxy service encounters an error during mediation or while forwarding a message, the message that triggered the error is delegated to the specified fault sequence. Using the available mediators it is possible to log the erroneous message, forward it to a special error-tracking service, and send a SOAP fault back to the client indicating the error or even send an email to the system admin.

It is not mandatory to associate each sequence and proxy service with a fault sequence. In situations where a fault sequence is not specified explicitly, a default fault sequence will be used to handle errors. [Sample 4: Specifying a Fault Sequence with a Regular Mediation Sequence](#) shows how to specify a fault sequence with a regular mediation sequence.

Whenever an error occurs in WSO2 ESB, the mediation engine attempts to provide as much information as possible on the error to the user by initializing the following properties on the erroneous message:

- ERROR\_CODE
- ERROR\_MESSAGE
- ERROR\_DETAIL
- ERROR\_EXCEPTION

Within the fault sequence, you can access these property values using the `get-property` XPath function. Sample 4 uses the log mediator as follows to log the actual error message:

```
<log level="custom">
 <property name="text" value="An unexpected error occurred"/>
 <property name="message" expression="get-property('ERROR_MESSAGE')"/>
</log>
```

Note how the `ERROR_MESSAGE` property is being used to get the error message text. If you want to customize the error message that is sent back to the client, you can use the `Fault mediator` as demonstrated in [Sample 5: Creating SOAP Fault Messages and Changing the Direction of a Message](#).

### Error codes

This section describes error codes and their meanings.

#### ***Transport error codes***

Error Code	Detail
101000	Receiver input/output error sending
101001	Receiver input/output error receiving
101500	Sender input/output error sending
101501	Sender input/output error receiving
101503	Connection failed
101504	Connection timed out (no input was detected on this connection over the maximum period of inactivity)
101505	Connection closed
101506	NHTTP protocol violation
101507	Connection canceled
101508	Request to establish new connection timed out
101509	Send abort
101510	Response processing failed

If the HTTP PassThrough transport is used, and a connection level error occurs, the error code is calculated using the following equation:

$$\text{Error code} = \text{Base error code} + \text{Protocol State}$$

There is a state machine in the transport sender side, where the protocol state changes according to the phase of the message.

Following are the possible protocol states and the description for each:

Protocol State	Description
REQUEST_READY(0)	Connection is at the initial stage ready to send a request
REQUEST_HEAD(1)	Sending the request headers through the connection
REQUEST_BODY(2)	Sending the request body
REQUEST_DONE(3)	Request is completely sent
RESPONSE_HEAD(4)	The connection is reading the response headers
RESPONSE_BODY(5)	The connection is reading the response body
RESPONSE_DONE(6)	The response is completed

CLOSING(7)	The connection is closing
CLOSED(8)	The connection is closed

Since there are several possible protocol states in which a request can time out, you can calculate the error code accordingly using the values in the table above.

For example, in a scenario where you send a request and the request is completely sent to the backend, but a timeout happens before the response headers are received, the error code is calculated as follows:

In this scenario, the base error code is CONNECTION\_TIMEOUT(101504) and the protocol state is REQUEST\_DONE (3).

Therefore,

$$\text{Error code} = 101504 + 3 = 101507$$

### ***Endpoint failures***

This section describes the error codes for endpoint failures. For more information on handling endpoint errors, see [Endpoint Error Handling](#).

#### **General errors**

Error Code	Detail
303000	Load Balance endpoint is not ready to connect
303000	Recipient List Endpoint is not ready
303000	Failover endpoint is not ready to connect
303001	Address Endpoint is not ready to connect
303002	WSDL Address is not ready to connect

#### **Failure on endpoint in the session**

Error Code	Detail
309001	Session aware load balance endpoint, No ready child endpoints
309002	Session aware load balance endpoint, Invalid reference
309003	Session aware load balance endpoint, Failed session

#### **Non-fatal warnings**

Error Code	Detail
303100	A failover occurred in a Load balance endpoint
304100	A failover occurred in a Failover endpoint

#### **Referring real endpoint is null**

Error Code	Detail
305100	Indirect endpoint not ready

## Callout operation failed

Error Code	Detail
401000	Callout operation failed (from the callout mediator)

### Custom error codes

Error Code	Detail
500000	Endpoint Custom Error - This error is triggered when the endpoint is prefixed by <property name="FORCE_ERROR_ON_SOAPFAULT" value="true" />, which enhances the failover logic by marking an endpoint as suspended when the response is a SOAP fault.

## Java Message Service (JMS) Support

Java Message Service (JMS) is a widely used API in Java-based Message Oriented Middleware(MOM) applications. It facilitates loosely coupled, reliable, and asynchronous communication between different components of a distributed application.

JMS supports two asynchronous communication models for messaging as follows:

- Point-to-point model - In this model message communication happens from one JMS client to another JMS client through a dedicated queue.
- Publish and subscribe model - In this model message communication happens from one JMS client(publisher) to many JMS clients(subscribers) through a topic.

This section provides information on JMS support in WSO2 ESB along with implementation details of common use cases, a configuration guide as well as a troubleshooting guide. In the [JMS Usecases](#) section you will also see use cases for the new messaging features introduced with JMS 2.0.

WSO2 ESB supports the following messaging features introduced with JMS 2.0:

- Shared Topic Subscription
- JMSX Delivery Count
- JMS Message Delivery Delay

- [JMS Usecases](#)
  - ESB as a JMS Consumer
  - ESB as a JMS Producer
  - ESB as Both a JMS Producer and Consumer
  - JMS Synchronous Invocations : Dual Channel HTTP-to-JMS
  - JMS Synchronous Invocations : Quad Channel JMS-to-JMS
  - Store and Forward Using JMS Message Stores
  - Publish-Subscribe with JMS
  - Shared Topic Subscription with WSO2 ESB
  - Detecting Repeatedly Redelivered Messages via WSO2 ESB Using the JMSXDeliveryCount Property
  - Using WSO2 ESB as a JMS Producer and Specifying a Delivery Delay on Messages
- [JMS Samples](#)
- [Advanced Topics](#)
  - [JMS Security Management](#)
    - Security in WSO2 Message Broker/Apache Qpid
    - Security in Apache ActiveMQ
  - [JMS Transactions](#)
  - [Working with Multiple Types of Brokers](#)

- JMS MapMessage Support
- JMS Troubleshooting Guide
- JMS FAQ

## JMS Usecases

JMS supports two models for messaging as follows:

- Queues : point-to-point
- Topics : publish and subscribe

This section provides information on how to implement common business use cases of both these patterns where WSO2 ESB acts as both a producer and consumer for a JMS broker.

- [ESB as a JMS Consumer](#)
- [ESB as a JMS Producer](#)
- [ESB as Both a JMS Producer and Consumer](#)
- [JMS Synchronous Invocations : Dual Channel HTTP-to-JMS](#)
- [JMS Synchronous Invocations : Quad Channel JMS-to-JMS](#)
- [Store and Forward Using JMS Message Stores](#)
- [Publish-Subscribe with JMS](#)
- [Shared Topic Subscription with WSO2 ESB](#)
- [Detecting Repeatedly Redelivered Messages via WSO2 ESB Using the JMSXDeliveryCount Property](#)
- [Using WSO2 ESB as a JMS Producer and Specifying a Delivery Delay on Messages](#)

## ESB as a JMS Consumer

This section describes how to configure WSO2 ESB to listen to a JMS Queue.

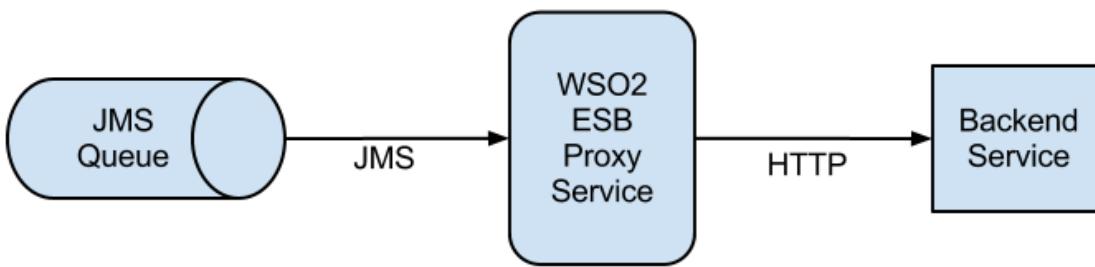


Diagram 1 : Simple JMS to HTTP proxy service

The following example code shows the configuration of WSO2 ESB to listen to a JMS queue, consume messages, and send them to a HTTP back-end service.

1. Configure WSO2 ESB with Apache ActiveMQ and set up the JMS listener. For instructions, see [Configure with ActiveMQ](#).
2. Start the Apache ActiveMQ server by executing the following command:  
`./activemq console`
3. Start the WSO2 ESB server by executing the following command:  
`./wso2server.sh`
4. Create a proxy service with the following configuration. To create a proxy service using ESB Tooling, see [Working with Proxy Services via ESB Tooling](#).

```

<proxy xmlns="http://ws.apache.org/ns/synapse" name="JMSToHTTPStockQuoteProxy"
transports="jms">
 <target>
 <inSequence>
 <property action="set" name="OUT_ONLY" value="true" />
 </inSequence>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 <outSequence>
 <send/>
 </outSequence>
 </target>
</proxy>

```

5. To test this you will need an HTTP back-end service. [Deploy](#) the SimpleStockQuoteService and start the [Axis2 server](#).
6. Place a message into the Apache ActiveMQ queue by executing the following command from <ESB\_HOME>/samples/axis2Client folder.

```

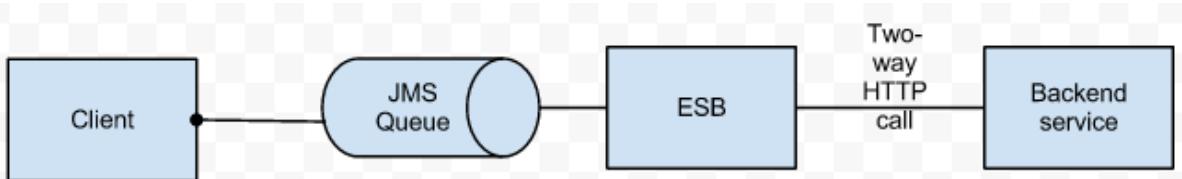
ant stockquote -Dmode=placeorder
-Dtrpurl="jms:/JMSToHTTPStockQuoteProxy?transport.jms.ConnectionFactoryJNDIName=Q
ueueConnectionFactory&java.naming.factory.initial=org.apache.activemq.jndi.Active
MQInitialContextFactory&java.naming.provider.url=tcp://localhost:61616&transport.
jms.ContentTypeProperty=Content-Type&transport.jms.DestinationType=queue"

```

You can make the proxy service a JMS listener by setting its transport as `jms`. Once the JMS transport is enabled for a proxy service, ESB will start listening on a JMS queue with the same name as the proxy service. In the [sample code above](#), ESB listens to a JMS queue named `JMSToHTTPStockQuoteProxy`. To make the proxy service listen to a different JMS queue, define the `transport.jms.Destination` parameter with the name of the destination queue.

### Two-way HTTP Back-end Call

In addition to one-way invocations, WSO2 ESB proxy service can listen to the queue, pick up a message and do a two-way HTTP call as well. This is done when the client specifies a `replyDestination` element when placing a request message to a JMS queue. It allows the response to be delivered to the `replyDestination` queue specified by the client. The scenario is depicted in the diagram below.



We can have a proxy service similar to the following to simulate a two-way invocation.

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="Proxy1"
 transports="jms"
 startOnLoad="true"
 trace="disable">
 <description/>
 <target>
 <inSequence>
 <send>
 <endpoint>
 <address
uri="http://localhost:9765/services/t/superqa.com/Axis2Service"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <parameter name="transport.jms.ContentType">
 <rules>
 <jmsProperty>contentType</jmsProperty>
 <default>text/xml</default>
 </rules>
 </parameter>
 </proxy>

```

### **Defining Content Type of Incoming JMS Messages**

By default, WSO2 ESB considers all messages consumed from a queue as a SOAP message. To consider messages consumed from a queue as a different format, define the **transport.jms.ContentType** parameter with the respective content type as a proxy service parameter. The sample code below

To demonstrate this, we have modified the [sample code 1](#) as follows to connect to **MyJMSQueue** queue and read messages as POX messages.

## Sample Code 2

```

<proxy xmlns="http://ws.apache.org/ns/synapse" name="JMSToHTTPStockQuoteProxy"
transports="jms">
 <target>
 <inSequence>
 <property action="set" name="OUT_ONLY" value="true" />
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <parameter name="transport.jms.ContentType">
 <rules>
 <jmsProperty>contentType</jmsProperty>
 <default>application/xml</default>
 </rules>
 </parameter>
 <parameter name="transport.jms.Destination">MyJMSQueue</parameter>
</proxy>

```

## ESB as a JMS Producer

This section describes how to configure WSO2 ESB to send messages to a JMS Queue.

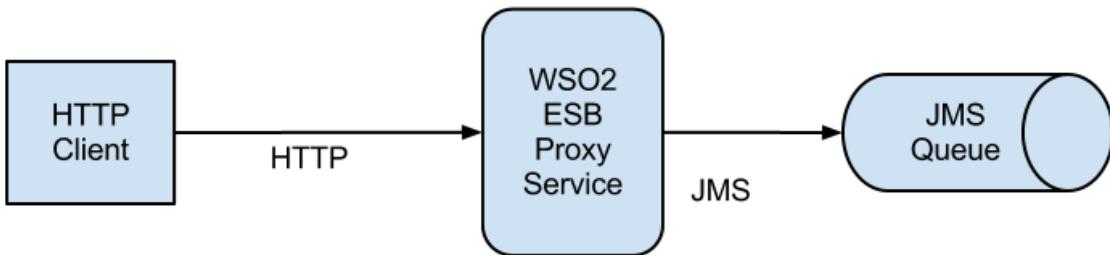


Diagram 1 : Simple HTTP to JMS proxy service

The following example code shows the configuration of ESB proxy service to accept messages via HTTP and send them to JMS queue.

## Sample code 2

```
<proxy xmlns="http://ws.apache.org/ns/synapse" name="StockQuoteProxy"
transports="http">
 <target>
 <inSequence>
 <property action="set" name="OUT_ONLY" value="true"/>
 <send>
 <endpoint>
 <address uri="" /> <!-- Specify the JMS connection URL here -->
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
</proxy>
```

During inSequence, the OUT\_ONLY property is set to true to indicate that message exchange is one-way.

### JMS Connection URL

To send a message to a JMS queue, we define a JMS connection URL (line numbers 7 in the configuration above) as the URL of the endpoint, which will be invoked via the send mediator.

You can define a JMS queue name and [connection factory parameters](#) in the JMS connection URL. Values of connection factory parameters depend on the type of the JMS broker. Listed below are examples of how the JMS connection URL can be defined for WSO2 Message Broker and for ActiveMQ.

Example JMS connection URL for ActiveMQ

```
jms:/SimpleStockQuoteService?transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory&java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory&java.naming.provider.url=tcp://localhost:61616&transport.jms.DestinationType=queue
```

Example JMS connection URL for WSO2 Message Broker

```
jms:/StockQuotesQueue?transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory&java.naming.factory.initial=org.wso2.andes.jndi.PropertiesFileInitialContextFactory&java.naming.provider.url=repository/conf/jndi.properties&transport.jms.DestinationType=queue
```

## Note

When entering these URLs in the management console, replace 'amp;' character in the endpoint URL with '&'

## Other Configurations

In addition to configuring the ESB proxy service as shown above, follow the steps below.

- To enable JMS sender in the ESB, un-comment following line in <ESB\_HOME>/repository/conf/axis2/axis2.xml.

```
<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender" />
```

- To enable JMS listeners in back-end axis2 services, edit <ESB\_HOME>/samples/axis2Server/repository/conf/axis2.xml file and configure JMS listener as explained section [Setting up the JMS Listener](#).
- To send an HTTP message to ESB proxy service, execute the following command from <ESB\_HOME>/sample/axis2Client.

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
-Dmode=placeorder -Dsymbol=MSFT
```

- During inSequence, the OUT\_ONLY property is set to true to indicate that message exchange is one-way.

## ESB as Both a JMS Producer and Consumer

This section describes how to configure WSO2 ESB to work as a JMS-to-JMS proxy service.

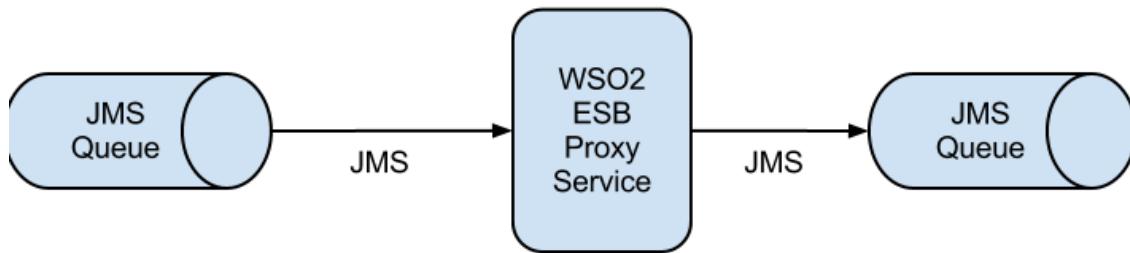


Diagram 3 : Simple JMS to JMS proxy service

The following example code shows the configuration of ESB to both listen to a JMS queue and consume messages as well as to send messages to a JMS queue.

### Example code 3

```
<proxy name="StockQuoteProxy" transports="jms">
 <target>
 <inSequence>
 <property action="set" name="OUT_ONLY" value="true" />
 <send>
 <endpoint>
 <address
uri="jms:/SimpleStockQuoteService?transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory&
java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory&
;java.naming.provider.url=tcp://localhost:61616"/>
 </endpoint>
 </send>
 </inSequence>
 </target>
</proxy>
```

To place a message into a JMS queue, execute following command from <ESB\_HOME>/samples/axis2Client directory.

```
ant stockquote -Dmode=placeorder
-Dtrpurl="jms:/StockQuoteProxy?transport.jms.ConnectionFactoryJNDIName=QueueConnection
Factory&java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFac
tory&java.naming.provider.url=tcp://localhost:61616&transport.jms.ContentTypeProperty=
Content-Type&transport.jms.DestinationType=queue"
```

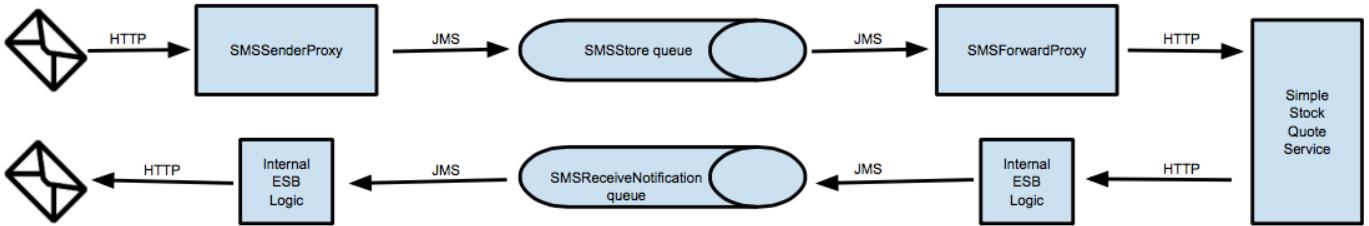
Generally, JMS is used for one-way, asynchronous message exchange. However you can perform synchronous messaging also with JMS. For more information, see [JMS Synchronous Invocations : Dual Channel HTTP-to-JMS](#) and [JMS Synchronous Invocations : Quad Channel JMS-to-JMS](#).

#### JMS Synchronous Invocations : Dual Channel HTTP-to-JMS

A JMS synchronous invocation takes place when a JMS producer receives a response to a JMS request produced by it when invoked. WSO2 ESB uses an internal **JMS correlation ID** to correlate the request and the response. See [JMS Request/Reply Example](#) for more information.

JMS synchronous invocations are further explained in the following use case.

#### Use case



When the proxy service named **SMSenderProxy** receives an HTTP request, it publishes that request in a JMS queue named **SMSStore**. Another proxy service named **SMSForwardProxy** subscribes to messages published in this queue and forwards them to a back-end service named **SimpleStockQuoteService**. When this back-end service returns an HTTP response, internal ESB logic is used to save that message as a JMS message in a JMS

queue named `SMSReceiveNotification`. Then this response is taken from the `SMSReceiveNotification` queue and delivered to the client as an HTTP message using internal ESB logic.

The following sub sections explain how to execute this use case. WSO2 Message Broker is used as the JMS broker.

- Prerequisites
- Configuring the JMS publisher
- Configuring the JMS consumer
- Start the back-end service
- Invoke the JMS publisher

### **Prerequisites**

Before executing this use case, the following steps need to be carried out. See [Integrating WSO2 ESB in MB Documentation](#) for detailed instructions.

- WSO2 MB should be installed and set up. See [Setting up WSO2 Message Broker](#).
- WSO2 ESB should be installed and set up. See [Setting up WSO2 ESB](#). Specific entries that are required to be added to the `<ESB_HOME>/repository/conf/jndi.properties` file for this use case are as follows.

Item	Value
Queue	<code>queue.SMSStore=SMSStore</code>
Queue	<code>queue.SMSReceiveNotificationStore=SMSReceiveNotificationStore</code>
Connection Factory	<code>connectionfactory.QueueConnectionFactory = amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5673'</code>

### **Configuring the JMS publisher**

Configure a proxy service named `SMSenderProxy` as shown below to accept messages sent via the HTTP transport, and to place those messages in the `SMSStore` queue in WSO2 MB.

#### Example code 4

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="SMSenderProxy"
 transports="https,http"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
 <target>
 <inSequence>
 <property name="transport.jms.ContentTypeProperty"
 value="Content-Type"
 scope="axis2"/>
 </inSequence>
 <outSequence>
 <property name="TRANSPORT_HEADERS" scope="axis2" action="remove" />
 <send/>
 </outSequence>
 <endpoint>
 <address
uri="jms:/SMSStore?transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory&java.naming.factory.initial=org.wso2.andes.jndi.PropertiesFileInitialContextFactory&java.naming.provider.url=repository/conf/jndi.properties&transport.jms.DestinationType=queue&transport.jms.ReplyDestination=SMSReceiveNotificationStore" />
 </endpoint>
 </target>
 <description/>
 </proxy>

```

The endpoint of this proxy service uses the following properties to map the proxy service with WSO2 MB.

Property	Value for this use case	Description
<b>address uri</b>	jms:/SMSStore	The destination in which the proxy service is stored.
<b>java.naming.factory.initial</b>	org.wso2.andes.jndi.PropertiesFileInitialContextFactory	This property specifies the initial context factory implementation. The value specified here is the same as the one specified in <ESB_HOME>/repository/conf/axis2.xml for the JNDI provider.
<b>java.naming.provider.url</b>	repository/conf/jndi.properties	The location of the JNDI provider configuration file.
<b>transport.jms.DestinationType</b>	queue	The destination type can be either queue or topic, generated by the proxy service.
<b>transport.jms.ReplyDestination</b>	SMSReceiveNotificationStore	The destination in which the proxy service sends back-end service is stored.

Since this is a two-way invocation, the **OUT\_ONLY** property is not set in the In sequence.

#### Configuring the JMS consumer

Configure a proxy service named **SMSForwardProxy** to consume messages from the **SMSStore** queue in WSO2 MB and forward them to the back-end service.

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="SMSForwardProxy"
 transports="jms"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
 <target>
 <inSequence>
 <send>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <parameter name="transport.jms.ContentType">
 <rules>
 <jmsProperty>contentType</jmsProperty>
 <default>text/xml</default>
 </rules>
 </parameter>
 <parameter
 name="transport.jms.ConnectionFactory">myQueueConnectionFactory</parameter>
 <parameter name="transport.jms.DestinationType">queue</parameter>
 <parameter name="transport.jms.Destination">SMSStore</parameter>
 <description/>
 </proxy>

```

The `transport.jms.ConnectionFactory`, `transport.jms.DestinationType` and `transport.jms.Destination` properties map this proxy service to the `SMSStore` queue.

The `SimpleStockQuoteService` sample shipped with WSO2 ESB is used as the back-end service in this example. To invoke this service, the address URI of this proxy service is defined as `http://localhost:9000/services/SimpleStockQuoteService`.

### **Start the back-end service**

In this example, the `SimpleStockQuoteService` serving as the back-end receives the message from the `SMSForwardProxy` proxy service via the JMS transport. The response is sent by `SimpleStockQuoteService` is published in the `SMSReceiveNotificationStore` queue which was set as the value for the `transport.jms.ReplyDestination` parameter of the `SMSenderProxy` proxy service. This allows the `SMSenderProxy` to pick the response and deliver it to the client.

The back-end service is started as follows.

1. Execute the following command from the `<ESB_HOME>/samples/axis2Server` directory.  
 For Windows: `axis2server.bat`  
 For Linux: `axis2server.sh`
2. Execute the `ant` command from `<ESB_HOME>/samples/axis2Server/src/SimpleStockQuoteService` directory.

### **Invoke the JMS publisher**

Execute the following command from the `<ESB_HOME>/sample/axis2Client` directory to invoke the `SMSSenderProxy` proxy service you defined as the JMS publisher.

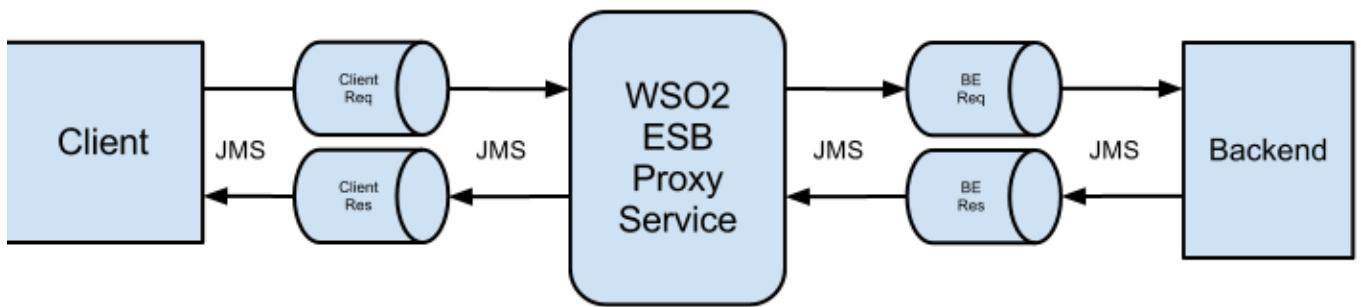
```
ant stockquote -Daddurl=http://localhost:8280/services/SMSSenderProxy -Dsymbol=IBM
```

You will get the following response.

```
Standard :: Stock price = $149.43669233447662
```

### JMS Synchronous Invocations : Quad Channel JMS-to-JMS

The following diagram depicts quad-channel JMS synchronous invocations of the WSO2 ESB.



The following example code shows configuration of WSO2 ESB for quad-channel JMS synchronous invocations.

### Example Code 5

```

<proxy name="QuadJMS" transports="jms">
 <target>
 <inSequence>
 <property action="set" name="transport.jms.ContentTypeProperty"
value="Content-Type" scope="axis2"/>
 <send>
 <endpoint>
 <address
uri="jms:/BEReq?transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory&
java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory&
;java.naming.provider.url=tcp://localhost:61616&transport.jms.DestinationType=queue&transport.jms.ReplyDestination=BERes"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <parameter name="transport.jms.ContentType">
 <rules>
 <jmsProperty>contentType</jmsProperty>
 <default>text/xml</default>
 </rules>
 </parameter>
 <parameter name="transport.jms.Destination">ClientReq</parameter>
</proxy>

```

The message flow of the sample code is as follows:

1. **JMSReplyTo** property of JMS message is set to **ClientRes**. Therefore, the client sends a JMS message to **ClientReq** queue.
2. Next, the **transport.jms.ReplyDestination** value is set to **BERes**. This enables the ESB proxy to pick messages from **ClientReq** queue, and send to **BEReq** queue.
3. Next, the back-end picks messages from the **BEReq** queue, processes and places response messages to **B ERes** queue.
4. Once a response is available in **BERes** queue, the proxy service picks it and sends back to **ClientRes** queue
5. Finally, the client picks it as the response message.

### Store and Forward Using JMS Message Stores

A message store is a storage in the ESB for messages. WSO2 ESB come with two types of message store implementations such as [In Memory Message Store](#) and [JMS Message Store](#). Users also have the option to create a [Custom Message Store](#) with their own message store implementation. For more information, refer to section [Message Stores](#).

This section specifically describes how to configure JMS Message stores for different message brokers. Given below is the generic configuration for a JMS message store.

```

<messageStore xmlns="http://ws.apache.org/ns/synapse"
 class="org.apache.synapse.message.store.impl.jms.JmsStore"
 name="JMSMS">
 <parameter
 name="java.naming.factory.initial">INITIAL_CONTEXT_FACTORY_NAME_HERE</parameter>
 <parameter name="java.naming.provider.url">URL_OF_THE_NAMING_PROVIDER</parameter>
</messageStore>

```

The configuration above only includes the mandatory elements. But, for some brokers you may have to specify extra parameters. Following table shows the list of configurable parameters.

Parameter Name	Value	Required
java.naming.factory.initial	Initial Context Factory used to connect to the JMS broker.	YES
java.naming.provider.url	Url of the naming provider to be used by the context factory.	YES
store.jms.destination	JNDI Name of the Queue Name that the message store is connecting to.	NO but for some JMS clients this will be needed
store.jms.connection.factory	JNDI name of the Connection factory, which is used to create jms connections.	NO but for some JMS clients this will be needed
store.jms.username	User Name that is used to create the connection with the broker.	NO
store.jms.password	Password that is used to create the connection with the broker.	NO
store.jms.JMSSpecVersion	1.1 or 1.0 JMS API specification to be used (Default 1.1)	NO
store.jms.cache.connection	true/false Enable Connection caching	NO

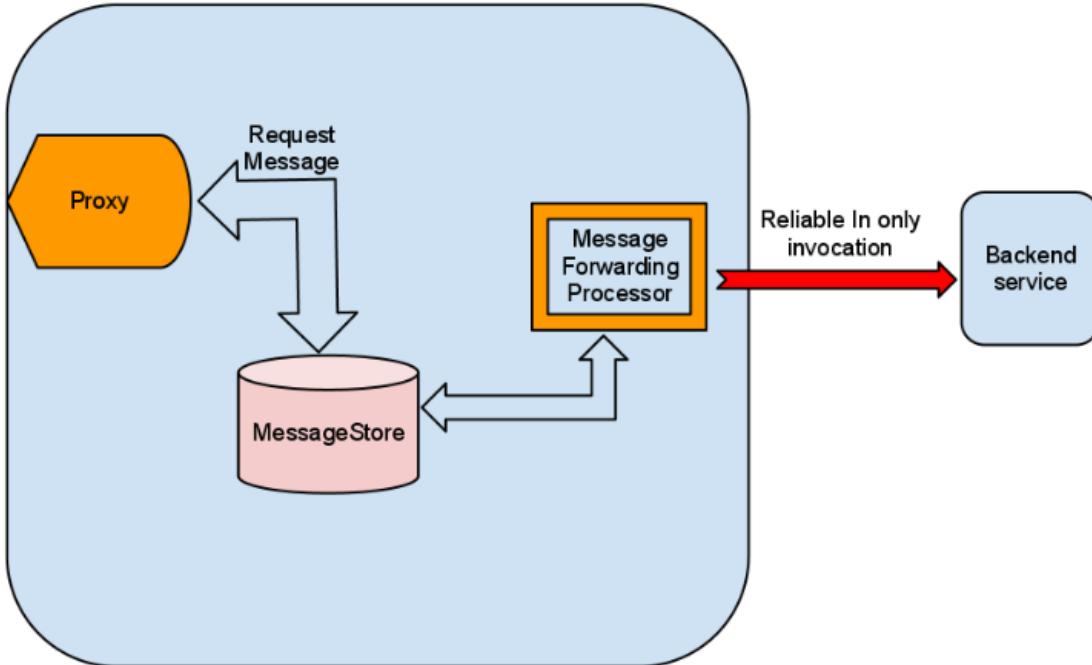
Next, let's see a few real-life business use cases of JMS message stores.

- [Use Case Scenario 1](#)
- [Use Case Scenario 2](#)

### Use Case Scenario 1

In this sample:

- The client sends requests to a proxy service.
- The proxy service stores the messages to a JMS message store.
- The back-end service is invoked by a message forwarding processor, which picks stored messages in the JMS message store.



Let's proceed to configuring this sample scenario.

#### **Configure the Broker Server**

1. Refer to [Configure with WSO2 Message Broker](#) or [Configure with ActiveMQ](#) for details on setting up a broker server for this sample. Note that you only need to set up the broker server itself and do **not** need to configure the listeners and senders in the ESB configuration (i.e., just perform steps 1 through 4 in the Message Broker instructions or 1 through 3 in the ActiveMQ instructions).
2. Create a JMS message store for the broker set up in step 1. To create proxy services, sequences, endpoints, message stores, processors etc. in ESB, you can either use the management console or copy the XML configuration to the source view. You can find the source view under menu **Manage > Service Bus > Source View** in the left navigation pane of the WSO2 ESB management console. Alternatively, you can add an XML file to `<ESB_HOME>/repository/deployment/server/synapse-configs/default/message-stores`.

For example,

- If you use ActiveMQ, the sample message store configuration is as follows:

Set the value of the the `<java.naming.provider.url>` property to point to the provider URL.

```

<messageStore xmlns="http://ws.apache.org/ns/synapse"
 class="org.apache.synapse.message.store.impl.jms.JmsStore"
 name="JMSMS">
 <parameter name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
</messageStore>

```

- If you use WSO2 Message Broker as the broker server, the sample message store configuration is as follows:

Set the value of the the <java.naming.provider.url> property to point to the jndi.properties file. In this case, <store.jms.destination> is a mandatory parameter.

```
<messageStore name="JMSMS"
class="org.apache.synapse.message.store.impl.jms.JmsStore"
xmlns="http://ws.apache.org/ns/synapse">
<parameter
name="java.naming.factory.initial">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
<parameter
name="java.naming.provider.url">repository/conf/jndi.properties</parameter>
<parameter name="store.jms.destination">JMSMS</parameter>
</messageStore>
```

### Configuring Back-end Service

1. Set up the prerequisites given in the **Prerequisites** section in [Setting Up the ESB Samples](#). Then, deploy the **SimpleStockQuoteService** client by navigating to <ESB\_HOME>/samples/axis2Server/src/SimpleStockQuoteService, and running the **ant** command in the command prompt or shell script. This will build the sample and deploy the service for you.
2. WSO2 ESB comes with a default Axis2 server, which you can use as the back-end service for this sample. To start the Axis2 server, navigate to <ESB\_HOME>/samples/axis2server and run axis2Server.sh (on Linux) or axis2Server.bat (on Windows).
3. Point your browser to <http://localhost:9000/services/SimpleStockQuoteService?wsdl> and verify that the service is running.

You now have a JMS message store configured in the ESB. Next, configure the ESB for the specific message broker you use.

### Configuring the ESB

1. Define an endpoint which is used to send the message to the back-end service.

```
<endpoint name="SimpleStockQuoteService">
<address uri="http://127.0.0.1:9000/services/SimpleStockQuoteService"/>
</endpoint>
```

2. Create a proxy service which stores messages to the created Message Store.

```
<proxy name="Proxy1" transports="https http" startOnLoad="true" trace="disable">
<target>
<inSequence>
<property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
<property name="OUT_ONLY" value="true"/>
<log level="full"/>
<store messageStore="JMSMS" />
</inSequence>
</target>
</proxy>
```

Note the following in the configuration:

- We use the property FORCE\_SC\_ACCEPTED in the message flow to send an Http 202 status to the client after ESB accepts a message. If this property is not specified, the the client which sends the request to the proxy service will timeout since it is not getting any response back from the proxy.

3. Create a message forwarding processor using the below configuration. Message forwarding processor consumes the messages stored in the message store.

```
<messageProcessor
class="org.apache.synapse.message.processor.impl.forwarder.ScheduledMessageForwardingP
rocessor" name="Processor1" targetEndpoint="SimpleStockQuoteService"
messageStore="JMSMS">
<parameter name="max.delivery.attempts">4</parameter>
<parameter name="interval">4000</parameter>
<parameter name="is.active">true</parameter>
</messageProcessor>
```

Once the back-end service and the ESB are configured, proceed to invoking the sample as follows.

### ***Executing the Sample***

1. To invoke the proxy service, we use the sample axis2 client shipped with the ESB. Navigate to <ESB\_HOME>/repository/samples/axis2client/ directory, and execute the following command to invoke the proxy service.

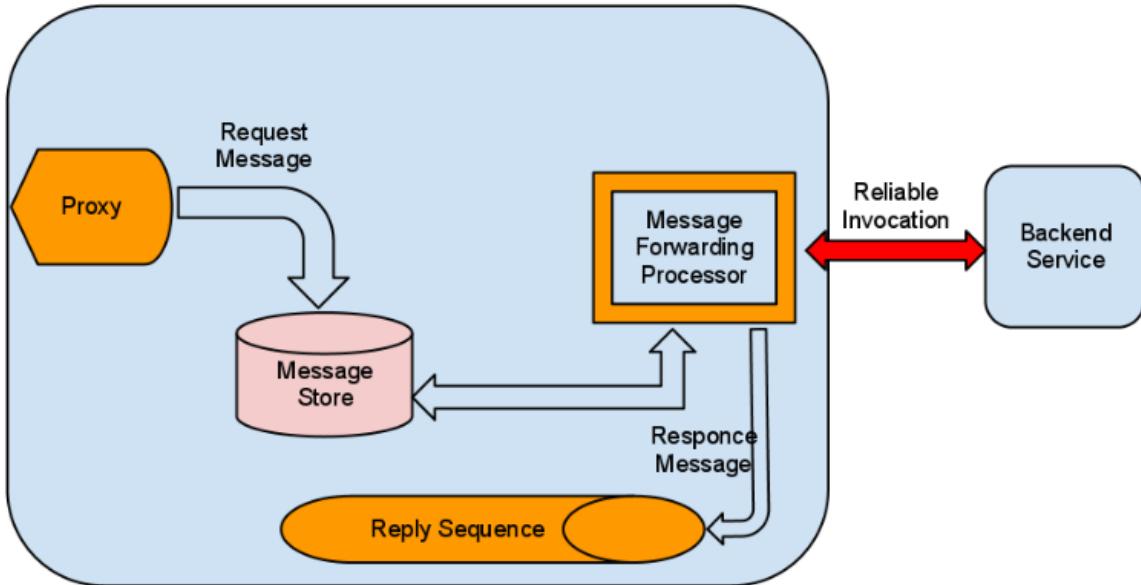
```
ant stockquote -Daddurl=http://localhost:8280/services/Proxy1 -Dmode=placeorder
```

2. Note a message similar to the following example printed in the Axis2 Server console.

```
SimpleStockQuoteService :: Accepted order for : 7482 stocks of IBM at $ 169.27205579038733
```

### **Use Case Scenario 2**

In the sample, when the message forwarding processor receives a response from the back-end service, it forwards it to a **replySequence** to process the response message.



Let's proceed to configuring this sample scenario.

#### **Configure the Broker Server**

1. Refer to [Configure with WSO2 Message Broker](#) or [Configure with ActiveMQ](#) for details on setting up a broker server for this sample. Note that you only need to set up the broker server itself and do **not** need to configure the listeners and senders in the ESB configuration (i.e., just perform steps 1 through 4 in the Message Broker instructions or 1 through 3 in the ActiveMQ instructions).
2. Create a JMS message store for the broker set up in step 1. To create proxy services, sequences, endpoints, message stores, processors etc. in ESB, you can either use the management console or copy the XML configuration to the source view. You can find the source view under menu **Manage > Service Bus > Source View** in the left navigation pane of the WSO2 ESB management console. Alternatively, you can add an XML file to `<ESB_HOME>/repository/deployment/server/synapse-configs/default/message-stores`.

For example,

- If you use ActiveMQ, the sample message store configuration is as follows:

Set the value of the the `<java.naming.provider.url>` property to point to the provider URL.

```

<messageStore xmlns="http://ws.apache.org/ns/synapse"
 class="org.apache.synapse.message.store.impl.jms.JmsStore"
 name="JMSMS">
 <parameter
 name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
</messageStore>

```

- If you use WSO2 Message Broker as the broker server, the sample message store configuration is as follows:

Set the value of the the `<java.naming.provider.url>` property to point to the `jndi.properties` file. In this case, `<store.jms.destination>` is a mandatory parameter.

```

<messageStore name="JMSMS"
class="org.apache.synapse.message.store.impl.jms.JmsStore"
xmlns="http://ws.apache.org/ns/synapse">
 <parameter
name="java.naming.factory.initial">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 <parameter
name="java.naming.provider.url">repository/conf/jndi.properties</parameter>
 <parameter name="store.jms.destination">JMSMS</parameter>
</messageStore>

```

### Configuring Back-end Service

1. Set up the prerequisites given in the **Prerequisites** section in [Setting Up the ESB Samples](#). Then, deploy the **SimpleStockQuoteService** client by navigating to <ESB\_HOME>/samples/axis2Server/src/SimpleStockQuoteService, and running the **ant** command in the command prompt or shell script. This will build the sample and deploy the service for you.
2. WSO2 ESB comes with a default Axis2 server, which you can use as the back-end service for this sample. To start the Axis2 server, navigate to <ESB\_HOME>/samples/axis2server and run axis2Server.sh (on Linux) or axis2Server.bat (on Windows).
3. Point your browser to <http://localhost:9000/services/SimpleStockQuoteService?wsdl> and verify that the service is running.

You now have a JMS message store configured in the ESB. Next, configure the ESB for the specific message broker you use.

### Configuring the ESB

1. Define an endpoint, which is used to send the message to the back-end service.

```

<endpoint name="SimpleStockQuoteService">
 <address uri="http://127.0.0.1:9000/services/SimpleStockQuoteService"/>
</endpoint>

```

2. Create a proxy service which stores messages to the created Message Store.

```

<proxy name="Proxy2" transports="https,http"
statistics="disable" trace="disable" startOnLoad="true">
 <target>
 <inSequence>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2" />
 <log level="full" />
 <store messageStore="JMSMS" />
 </inSequence>
 </target>
</proxy>

```

3. Create a sequence to handle the response received from the back-end service.

```
<sequence name="replySequence">
 <log level="full">
 <property name="REPLY" value="MESSAGE" />
 </log>
 <drop/>
</sequence>
```

4. Create a message forwarding processor using the below configuration. Message forwarding processor consumes the messages stored in the message store. Compared to the message processor in [sample 1](#), this has an additional parameter **message.processor.reply.sequence** to point to a sequence to handle the response message.

```
<messageProcessor
 class="org.apache.synapse.message.processor.impl.forwarder.ScheduledMessageForwardingProcessor"
 name="Processor2" messageStore="JMSMS">
 <parameter name="interval">4000</parameter>
 <parameter name="max.delivery.attempts">4</parameter>
 <parameter name="message.processor.reply.sequence">replySequence</parameter>
</messageProcessor>
```

Once the back-end service and the ESB are configured, proceed to invoking the sample as follows.

### ***Executing the Sample***

1. To invoke the proxy service, we use the sample axis2 client shipped with the ESB. Navigate to <ESB\_HOME>/repository/samples/axis2client/ directory, and execute the following command to invoke the proxy service.

```
ant stockquote -Daddurl=http://localhost:8280/services/Proxy2
```

2. Note the service being invoked and then the response being logged from the replySequence. For example,

```

INFO - LogMediator To: /services/InOutProxy, WSAction: urn:getSimpleQuote, SOAPAction:
urn:getSimpleQuote,
MessageID: urn:uuid:dec12d9c-5289-476c-9d9a-b7bb7ebc7be4, Direction: request, REPLY =
MESSAGE,
Envelope:
<?xml version='1.0' encoding='utf-8'?>
<soapenv:envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:body><ns:getsimplequoteresponse xmlns:ns="http://services.samples">
<ns:return xmlns:ax21="http://services.samples/xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="ax21:GetQuoteResponse">
<ax21:change>-2.3141856129298564</ax21:change><ax21:earnings>12.877155014054368</ax21
:earnings>
<ax21:high>172.73334579339183</ax21:high><ax21:last>165.31090559096748</ax21:last>
<ax21:lasttradetimestamp>Thu Dec 29 07:48:42 IST 2011</ax21:lasttradetimestamp>
<ax21:low>-164.80767926468306</ax21:low><ax21:marketcap>9451314.231029626</ax21:marke
tcap>
<ax21:name>IBM Company</ax21:name><ax21:open>-161.41234152690964</ax21:open>
<ax21:peratio>25.74977555860659</ax21:peratio><ax21:percentagechange>-1.22140363581356
63</ax21:percentagechange>
<ax21:prevclose>189.46935681818218</ax21:prevclose><ax21:symbol>IBM</ax21:symbol><ax21
:volume>8611</ax21:volume>
</ns:return></ns:getsimplequoteresponse></soapenv:body></soapenv:envelope>

```

## Publish-Subscribe with JMS

JMS supports two models for messaging as follows:

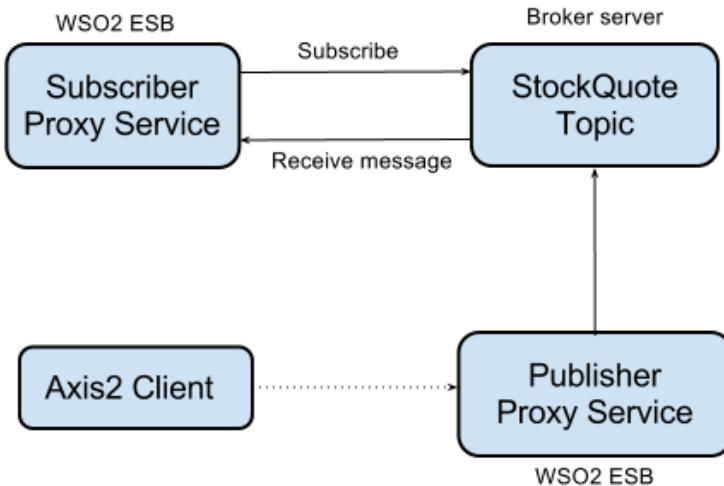
- Queues: point-to-point
- Topics: publish and subscribe

There are many business use cases that can be implemented using the publisher-subscriber (pub-sub) pattern. For example, consider a blog with subscribed readers. The blog author posts a blog entry, which the subscribers of that blog can view. In other words, the blog author publishes a message (the blog post content) and the subscribers (the blog readers) receive that message. Popular publisher/subscriber patterns like these can be implemented using JMS topics, as described in the following sections.

- Scenario overview
- Configuring the broker server
- Configuring the publisher
- Configuring the subscribers
- Publishing to the topic

### Scenario overview

In this sample scenario, we create a JMS Topic in a broker server such as ActiveMQ or the WSO2 Message Broker and then add proxy services that act as the publisher and subscribers in WSO2 ESB. This example assumes you have already downloaded and installed WSO2 ESB (see [Installation Guide](#)).



### Configuring the broker server

For this example, we will use ActiveMQ as our broker server. Follow the instructions in [Configure with ActiveMQ](#) to set up ActiveMQ for use with WSO2 ESB.

### Configuring the publisher

1. Open the `<ESB_HOME>/repository/conf/JNDI.properties` file and specify the JNDI designation of the topic (in this example, `SimpleStockQuoteService`). For example:

```

register some queues in JNDI using the form
queue.[jndiName] = [physicalName]
queue.MyQueue = example.MyQueue

register some topics in JNDI using the form
topic.[jndiName] = [physicalName]
topic.MyTopic = example.MyTopic
topic.SimpleStockQuoteService = SimpleStockQuoteService

```

2. Next, [add a proxy service named StockQuoteProxy](#) and configure it to publish to the topic `SimpleStockQuoteService`. You can add the proxy service to the ESB using the management console, either by building the proxy service in the design view or by copying the XML configuration into the source view. Alternatively, you can add an XML file named `StockQuoteProxy.xml` to `<ESB_HOME>/repository/deployment/server/synapse-configs/default/proxy-services`. A sample XML code segment that defines the proxy service is given below. Notice that the address URI specifies properties for configuring the JMS transport.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy"
 transports="http"
 startOnLoad="true"
 trace="disable">
 <target>
 <endpoint>

 <address
uri="jms:/SimpleStockQuoteService?transport.jms.ConnectionFactoryJNDIName=TopicConnectionFactory&java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory&java.naming.provider.url=tcp://localhost:61616&transport.jms.DestinationType=topic"/>
 </endpoint>
 <inSequence>
 <property name="OUT_ONLY" value="true"/>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 </proxy>
 </definitions>

```

If you are using the source view in the management console, you must use '&' instead of '=' in endpoint URLs, as shown in the example above. If you are saving this proxy service configuration as an XML file in the proxy-services directory, use '=' instead.

## Configuring the subscribers

Next, you configure two proxy services that subscribe to the JMS topic SimpleStockQuoteService, so that whenever this topic receives a message, it is sent to these subscribing proxy services. Following is the sample configuration for these proxy services.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="SimpleStockQuoteService1"
 transports="jms"
 startOnLoad="true"
 trace="disable">
 <description/>
 <target>
 <inSequence>
 <property name="OUT_ONLY" value="true"/>
 <log level="custom">
 <property name="Subscriber1" value="I am Subscriber1"/>
 </log>
 <drop/>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <parameter name="transport.jms.ContentType">
 <rules>

```

```
<jmsProperty>contentType</jmsProperty>
 <default>application/xml</default>
</rules>
</parameter>
<parameter
name="transport.jms.ConnectionFactory">myTopicConnectionFactory</parameter>
 <parameter name="transport.jms.DestinationType">topic</parameter>
 <parameter name="transport.jms.Destination">SimpleStockQuoteService</parameter>
</proxy>

<proxy name="SimpleStockQuoteService2"
 transports="jms"
 startOnLoad="true"
 trace="disable">
 <description/>
 <target>
 <inSequence>
 <property name="OUT_ONLY" value="true"/>
 <log level="custom">
 <property name="Subscriber2" value="I am Subscriber2"/>
 </log>
 <drop/>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <parameter name="transport.jms.ContentType">
 <rules>
 <jmsProperty>contentType</jmsProperty>
 <default>application/xml</default>
 </rules>
 </parameter>
 <parameter
name="transport.jms.ConnectionFactory">myTopicConnectionFactory</parameter>
 <parameter name="transport.jms.DestinationType">topic</parameter>
```

```

<parameter name="transport.jms.Destination">SimpleStockQuoteService</parameter>
</proxy>
</definitions>

```

## Publishing to the topic

1. Start the ESB with one of the following commands:
  - <ESB\_HOME>/bin/wso2server.sh (on Linux)
  - <ESB\_HOME>/bin/wso2server.bat (on Windows)

A log message similar to the following will appear:

```

INFO {org.wso2.andes.server.store.CassandraMessageStore} - Created Topic :
SimpleStockQuoteService
INFO {org.wso2.andes.server.store.CassandraMessageStore} - Registered
Subscription tmp_127_0_0_1_44759_1 for Topic SimpleStockQuoteService

```

2. To invoke the publisher, use the sample stockquote client service by navigating to <ESB\_HOME>/samples/axis2Client and running the following command:

```

ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
-Dmode=placeorder -Dsymbol=MSFT

```

The message flow is executed as follows:

- When the stockquote client sends the message to the StockQuoteProxy service, the publisher is invoked and sends the message to the JMS topic.
- The topic delivers the message to all the subscribers of that topic. In this case, the subscribers are ESB proxy services.

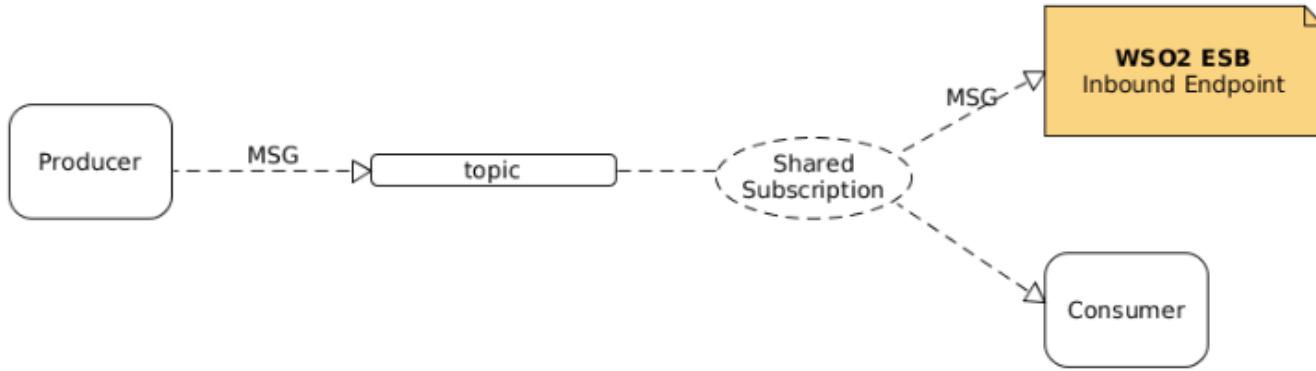
There can be many types of publishers and subscribers for a given JMS topic. The following article in the WSO2 library provides more information on different types of publishers and subscribers: <http://wso2.org/library/articles/2011/12/wso2-esb-example-pubsub-soa>.

## Shared Topic Subscription with WSO2 ESB

With JMS 1.1, a subscription on a topic is not permitted to have more than one consumer at a time. That is, if multiple JMS consumers subscribe to a JMS topic, and if a message comes to that topic, multiple copies of the message is forwarded to each consumer. There is no way of sharing messages between consumers that come to the topic.

With the shared subscription feature in JMS 2.0 you can overcome this restriction. When shared subscription is used, a message that comes to a topic is forwarded to only one of the consumers. That is, if multiple JMS consumers subscribe to a JMS topic, consumers can share the messages that come to the topic. The advantage of shared topic subscription is that it allows to share the workload between consumers.

WSO2 ESB can be configured as a shared topic listener so that it can connect to a shared topic subscription as a message consumer (subscriber) to share workload between other consumers of the subscription. The following diagram illustrates this sample scenario:



To demonstrate the sample scenario, we will configure WSO2 ESB's JMS inbound endpoint as a shared topic listener using HornetQ as the message broker. This sample scenario includes the following sections:

- Prerequisites
- Configuring and starting the HornetQ message broker
- Building the sample scenario
- Executing the sample scenario
- Analyzing the output

## Prerequisites

- Download the `hornetq-all-new.jar` and copy it to the `<ESB_HOME>/repository/components/lib/` directory.
- Replace the `geronimo-jms_1.1_spec-1.1.0.wso2v1.jar` file in the `<ESB_HOME>/lib/endorsed/` directory with `javax.jms-api-2.0.1.jar`
- Download HornetQ from <http://hornetq.jboss.org/downloads> and extract the tar.gz.

## Configuring and starting the HornetQ message broker

1. Open the `<HornetQ_HOME>/config/stand-alone/non-clustered/hornetq-jms.xml` file in a text editor and add the following configuration:

```

<connection-factory name="QueueConnectionFactory">
 <xa>false</xa>
 <connectors>
 <connector-ref connector-name="netty" />
 </connectors>
 <entries>
 <entry name="/QueueConnectionFactory" />
 </entries>
</connection-factory>

<connection-factory name="TopicConnectionFactory">
 <xa>false</xa>
 <connectors>
 <connector-ref connector-name="netty" />
 </connectors>
 <entries>
 <entry name="/TopicConnectionFactory" />
 </entries>
</connection-factory>

<queue name="wso2">
 <entry name="/queue/mySampleQueue" />
</queue>

<topic name="sampleTopic">
 <entry name="/topic/exampleTopic" />
</topic>

```

## 2. Start the HornetQ message broker

- To start the message broker on Linux, navigate to the <HornetQ\_HOME>/bin/ directory and execute the run.sh command with root privileges.

### **Building the sample scenario**

The XML configuration for this sample scenario is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <registry provider="org.wso2.carbon.mediation.registry.WSO2Registry">
 <parameter name="cachableDuration">15000</parameter>
 </registry>
 <taskManager provider="org.wso2.carbon.mediation.ntask.NTaskTaskManager">
 <parameter name="cachableDuration">15000</parameter>
 </taskManager>
 <sequence name="request" onError="fault">
 <log level="full"/>
 <drop/>
 </sequence>
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default "fault" sequence"/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
 </sequence>
 <sequence name="main">
 <log level="full"/>
 <drop/>
 </sequence>
 <inboundEndpoint name="jms_inbound">
 <sequence>
 <onError>fault</onError>
 <protocol>jms</protocol>
 <suspend>false</suspend>
 </sequence>
 <parameters>
 <parameter name="interval">1000</parameter>
 <parameter name="transport.jms.Destination">/topic/exampleTopic</parameter>
 <parameter name="transport.jms.CacheLevel">3</parameter>
 <parameter
 name="transport.jms.ConnectionFactoryJNDIName">TopicConnectionFactory</parameter>
 <parameter name="sequential">true</parameter>
 <parameter
 name="java.naming.factory.initial">org.jnp.interfaces.NamingContextFactory</parameter>
 <parameter name="java.naming.provider.url">jnp://localhost:1099</parameter>
 <parameter
 name="transport.jms.SessionAcknowledgement">AUTO_ACKNOWLEDGE</parameter>
 <parameter name="transport.jms.SessionTransacted">false</parameter>
 <parameter name="transport.jms.ConnectionFactoryType">topic</parameter>
 <parameter name="transport.jms.JMSSpecVersion">2.0</parameter>
 <parameter name="transport.jms.SharedSubscription">true</parameter>
 <parameter
 name="transport.jms.DurableSubscriberName">mySubscription</parameter>
 </parameters>
 </inboundEndpoint>
</definitions>

```

## To build the sample

- Start the ESB with the sample configuration.

## Executing the sample scenario

- Run the following java file:

### TopicConsumer.java

```

package SharedTopicSubscribe;

import java.util.Properties;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.Topic;
import javax.naming.Context;
import javax.naming.InitialContext;

public class TopicConsumer {
 private static final String DEFAULT_CONNECTION_FACTORY =
 "TopicConnectionFactory";
 private static final String DEFAULT_DESTINATION = "/topic/exampleTopic";
 private static final String INITIAL_CONTEXT_FACTORY =
 "org.jnp.interfaces.NamingContextFactory";
 private static final String PROVIDER_URL = "jnp://localhost:1099";
 private static final String SUBSCRIPTION_NAME = "mySubscription";

 public static void main(final String[] args) {
 try {
 runExample();
 } catch (Exception e) {
 e.printStackTrace();
 }
 }

 public static void runExample() throws Exception {
 Connection connection = null;
 Context initialContext = null;
 try {
 // /Step 1. Create an initial context to perform the JNDI lookup.
 final Properties env = new Properties();
 env.put(Context.INITIAL_CONTEXT_FACTORY, INITIAL_CONTEXT_FACTORY);
 env.put(Context.PROVIDER_URL, System.getProperty(Context.PROVIDER_URL,
PROVIDER_URL));
 initialContext = new InitialContext(env);

 // Step 2. perform a lookup on the topic
 Topic topic = (Topic) initialContext.lookup(DEFAULT_DESTINATION);

 // Step 3. perform a lookup on the Connection Factory
 ConnectionFactory cf =
 (ConnectionFactory)
 initialContext.lookup(DEFAULT_CONNECTION_FACTORY);

 // Step 4. Create a JMS Connection
 connection = cf.createConnection();

 // Step 5. Create a JMS Session
 Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
 }
 }
}

```

```
// Step 6. Create a JMS Message Consumer
MessageConsumer messageConsumer =
 session.createSharedConsumer(topic,
SUBSCRIPTION_NAME);

// Step 7. Start the Connection
connection.start();
System.out.println("Shared message consumer started on topic: " +
DEFAULT_DESTINATION +
"\n");

// Step 8. Receive the message
TextMessage messageReceived = null;
while (true) {
 messageReceived = (TextMessage) messageConsumer.receive();
 System.out.println("Consumer received message: " + messageReceived.getText()
+ "\n");
}

} finally {

// Step 9. Close JMS resources
if (connection != null) {
 connection.close();
}

// Also the initialContext
if (initialContext != null) {
 initialContext.close();
}
```

```

 }
}
}
```

This acts as the shared topic subscriber with WSO2 ESB's inbound endpoint.

- Run the following java file to publish 5 messages to the HornetQ topic:

### TopicPublisher.java

```

package SharedTopicSubscribe;

import java.util.Properties;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.Topic;
import javax.naming.Context;
import javax.naming.InitialContext;

public class TopicPublisher {
 private static final String DEFAULT_CONNECTION_FACTORY =
 "TopicConnectionFactory";
 private static final String DEFAULT_DESTINATION = "/topic/exampleTopic";
 private static final String INITIAL_CONTEXT_FACTORY =
 "org.jnp.interfaces.NamingContextFactory";
 private static final String PROVIDER_URL = "jnp://localhost:1099";
 // Set up all the default values
 private static final String param = "IBM";

 public static void main(final String[] args) {
 try {
 runExample();
 } catch (Exception e) {
 e.printStackTrace();
 }
 }

 public static boolean runExample() throws Exception {
 Connection connection = null;
 Context initialContext = null;
 try {
 // Step 1. Create an initial context to perform the JNDI lookup.
 // Set up the namingContext for the JNDI lookup
 final Properties env = new Properties();
 env.put(Context.INITIAL_CONTEXT_FACTORY, INITIAL_CONTEXT_FACTORY);
 env.put(Context.PROVIDER_URL, System.getProperty(Context.PROVIDER_URL,
PROVIDER_URL));
 initialContext = new InitialContext(env);

 // Step 2. perform a lookup on the topic
 Topic topic = (Topic) initialContext.lookup(DEFAULT_DESTINATION);

 // Step 3. perform a lookup on the Connection Factory
 }
```

```

ConnectionFactory cf =
 (ConnectionFactory) initialContext.lookup(DEFAULT_CONNECTION_FACTORY);

// Step 4. Create a JMS Connection
connection = cf.createConnection();

// Step 5. Create a JMS Session
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

// Step 6. Create a Message Producer
MessageProducer producer = session.createProducer(topic);
System.out.println("Publishing 5 messages to topic/exampleTopic");
for (int i = 0; i < 5; i++) {

 // Step 7. Create a Text Message
 TextMessage message = session.createTextMessage(getMessage());

 // Step 8. Send the Message
 producer.send(message);
}
return true;
} finally {

 // Step 9. Close JMS resources
 if (connection != null) {
 connection.close();
 }

 // Also the initialContext
 if (initialContext != null) {
 initialContext.close();
 }
}

private static double getRandom(double base, double varience, boolean onlypositive) {
 double rand = Math.random();
 return (base + (rand > 0.5 ? 1 : -1) * varience * base * rand) *
 (onlypositive ? 1 : rand > 0.5 ? 1 : -1);
}

private static String getMessage() {
 return "<soapenv:Envelope
xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope/\">\n" +
 " <soapenv:Header/>\n" + "<soapenv:Body>\n" +
 " <m:placeOrder xmlns:m=\"http://services.samples\">\n" + " <m:order>\n" +
 " <m:price>" + getRandom(100, 0.9, true) + "</m:price>\n" +
 " <m:quantity>" + (int) getRandom(10000, 1.0, true) +
 "</m:quantity>\n" +
 " <m:symbol>" + param + "</m:symbol>\n" + " </m:order>\n" +

```

```

 " + "
```

## Analyzing the output

You will see that 5 messages are shared between the inbound listener and `TopicConsumer.java`. This is because both the ESB inbound listener and `TopicConsumer.java` are configured as shared subscribers.

The total number of consumed messages between the inbound listener and `TopicConsumer.java` will be equal to the number messages published by `TopicPublisher.java`.

## Detecting Repeatedly Redelivered Messages via WSO2 ESB Using the `JMSXDeliveryCount` Property

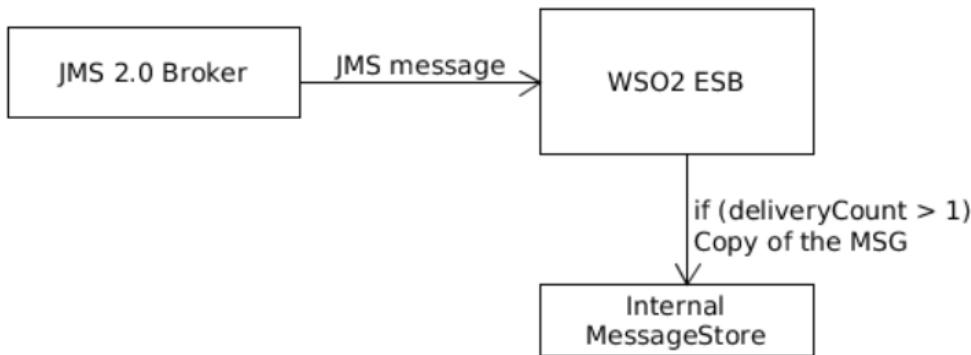
In JMS 2.0, it is mandatory for JMS providers to set the `JMSXDeliveryCount` property, which allows an application that receive a message to determine how many times the message is redelivered.

If a message is being redelivered, it means that a previous attempt to deliver the message failed due to some reason. If a message is being redelivered multiple times, it can be because the message is *bad* in some way. When such a message is being redelivered over and over again, it wastes resources and prevents subsequent *good* messages from being processed.

WSO2 ESB allows you to detect such repeatedly redelivered messages using the `JMSXDeliveryCount` property that is set in messages.

Detecting repeatedly redelivered messages is particularly useful since it makes it possible to handle such messages in a proper manner. For example, consume such message and send the messages to a separate queue.

The following diagram illustrates how WSO2 ESB can be used to detect repeatedly redelivered messages and store such messages in an internal message store.



To demonstrate the scenario illustrated above, we will configure WSO2 ESB's JMS inbound endpoint using HornetQ as the message broker. This sample scenario includes the following sections:

- Prerequisites
- Configuring and starting the HornetQ message broker
- Building the sample scenario
- Executing the sample scenario
- Analyzing the output

### Prerequisites

- Download the `hornetq-all-new.jar` and copy it to the `<ESB_HOME>/repository/components/lib/` directory.

- Replace the geronimo-jms\_1.1\_spec-1.1.0.wso2v1.jar file in the <ESB\_HOME>/lib/endorsed/ directory with javax.jms-api-2.0.1.jar
- Download HornetQ from <http://hornetq.jboss.org/downloads> and extract the tar.gz.

## Configuring and starting the HornetQ message broker

1. Open the <HornetQ\_HOME>/config/stand-alone/non-clustered/hornetq-jms.xml file in a text editor and add the following configuration:

```

<connection-factory name="QueueConnectionFactory">
 <xa>false</xa>
 <connectors>
 <connector-ref connector-name="netty" />
 </connectors>
 <entries>
 <entry name="/QueueConnectionFactory" />
 </entries>
</connection-factory>

<connection-factory name="TopicConnectionFactory">
 <xa>false</xa>
 <connectors>
 <connector-ref connector-name="netty" />
 </connectors>
 <entries>
 <entry name="/TopicConnectionFactory" />
 </entries>
</connection-factory>

<queue name="wso2">
 <entry name="/queue/mySampleQueue" />
</queue>

<topic name="sampleTopic">
 <entry name="/topic/exampleTopic" />
</topic>

```

2. Start the HornetQ message broker

- To start the message broker on Linux, navigate to the <HornetQ\_HOME>/bin/ directory and execute the run.sh command with root privileges.

## Building the sample scenario

The XML configuration for this sample scenario is as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <registry provider="org.wso2.carbon.mediation.registry.WSO2Registry">
 <parameter name="cachableDuration">15000</parameter>
 </registry>
 <taskManager provider="org.wso2.carbon.mediation.ntask.NTaskTaskManager">
 <parameter name="cachableDuration">15000</parameter>
 </taskManager>
 <sequence name="request" onError="fault">
 <log level="full" />
 <filter regex="1"
 source="get-property('default','jms.message.delivery.count')"

```

```

xmlns:ns="http://org.apache.synapse/xsd">
 <then>
 <log>
 <property name="DeliveryCounter" value="1"/>
 </log>
 </then>
 <else>
 <store messageStore="JMS-Redelivered-Store" />
 <log>
 <property name="DeliveryCounter" value="more than 1"/>
 </log>
 </else>
</filter>
<drop/>
</sequence>
<sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default "fault" sequence" />
 <property expression="get-property('ERROR_CODE')" name="ERROR_CODE" />
 <property expression="get-property('ERROR_MESSAGE')" name="ERROR_MESSAGE" />
 </log>
 <drop/>
</sequence>
<sequence name="main">
 <log level="full"/>
 <drop/>
</sequence>
<messageStore name="JMS-Redelivered-Store"/>
<inboundEndpoint name="jms_inbound" onError="fault" protocol="jms" sequence="request" suspend="false">
 <parameters>
 <parameter name="interval">1000</parameter>
 <parameter name="transport.jms.Destination">queue/mySampleQueue</parameter>
 <parameter name="transport.jms.CacheLevel">1</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName">QueueConnectionFactory</parameter>
 <parameter name="sequential">true</parameter>
 <parameter name="java.naming.factory.initial">org.jnp.interfaces.NamingContextFactory</parameter>
 <parameter name="java.naming.provider.url">jnp://localhost:1099</parameter>
 <parameter name="transport.jms.SessionAcknowledgement">AUTO_ACKNOWLEDGE</parameter>
 <parameter name="transport.jms.SessionTransacted">false</parameter>
 <parameter name="transport.jms.ConnectionFactoryType">queue</parameter>
 </parameters>
</inboundEndpoint>

```

```

 </parameters>
</inboundEndpoint>
</definitions>

```

This configuration creates an inbound endpoint to the JMS broker and has a simple sequence that logs the message status using the `JMSXDeliveryCount` value.

To build the sample

- Start the ESB with the sample configuration.

### Executing the sample scenario

- Run the following java file to publish a message to the JMS queue:

#### SOAPPublisher.java

```

package JMSXDeliveryCount;

import java.util.Properties;
import java.util.logging.Logger;

import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSContext;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class SOAPPublisher {
 private static final Logger log =
Logger.getLogger(SOAPPublisher.class.getName());

 // Set up all the default values
 private static final String param = "IBM";

 // with header for inbounds
 private static final String MESSAGE_WITH_HEADER =
 "<soapenv:Envelope
xmlns:soapenv=\\"http://schemas.xmlsoap.org/soap/envelope/\\">\n" +
 " <soapenv:Header>\n" +
 " <soapenv:Body>\n" +
 " <m:placeOrder xmlns:m=\\"http://services.samples\\">\n" +
 " <m:order>\n" +
 " <m:price>" +
 getRandom(100, 0.9, true) +
 "</m:price>\n" +
 " <m:quantity>" +
 (int) getRandom(10000, 1.0, true) +
 "</m:quantity>\n" +
 " <m:symbol>" +
 param +
 "</m:symbol>\n" +
 " </m:order>\n" +
 " </m:placeOrder>" +
 " </soapenv:Body>\n" +
 "</soapenv:Envelope>";

```

```

private static final String DEFAULT_CONNECTION_FACTORY =
"QueueConnectionFactory";
private static final String DEFAULT_DESTINATION = "queue/mySampleQueue";
private static final String INITIAL_CONTEXT_FACTORY =
"org.jnp.interfaces.NamingContextFactory";
private static final String PROVIDER_URL = "jnp://localhost:1099";

public static void main(String[] args) {

 Context namingContext = null;

 try {

 // Set up the namingContext for the JNDI lookup
 final Properties env = new Properties();
 env.put(Context.INITIAL_CONTEXT_FACTORY, INITIAL_CONTEXT_FACTORY);
 env.put(Context.PROVIDER_URL, System.getProperty(Context.PROVIDER_URL,
PROVIDER_URL));
 namingContext = new InitialContext(env);

 // Perform the JNDI lookups
 String connectionFactoryString =
 System.getProperty("connection.factory",
 DEFAULT_CONNECTION_FACTORY);
 log.info("Attempting to acquire connection factory \\" + +
connectionFactoryString + "\\");
 ConnectionFactory connectionFactory =
 (ConnectionFactory) namingContext.lookup(connectionFactoryString);
 log.info("Found connection factory \\" + connectionFactoryString + "\\ in
JNDI");

 String destinationString = System.getProperty("destination",
DEFAULT_DESTINATION);
 log.info("Attempting to acquire destination \\" + destinationString + "\\");
 Destination destination = (Destination)
namingContext.lookup(destinationString);
 log.info("Found destination \\" + destinationString + "\\ in JNDI");

 // String content = System.getProperty("message.content",
 // DEFAULT_MESSAGE);
 String content = System.getProperty("message.content", MESSAGE_WITH_HEADER);

 try (JMSContext context = connectionFactory.createContext()) {
 log.info("Sending message");
 // Send the message
 context.createProducer().send(destination, content);
 }

 } catch (NamingException e) {
 log.severe(e.getMessage());
 } finally {
 if (namingContext != null) {
 try {
 namingContext.close();
 } catch (NamingException e) {
 log.severe(e.getMessage());
 }
 }
 }
}

```

```
}
```

```
private static double getRandom(double base, double variance, boolean
onlypositive) {
 double rand = Math.random();
 return (base + (rand > 0.5 ? 1 : -1) * variance * base * rand) *
```

```

 (onlypositive ? 1 : rand > 0.5 ? 1 : -1);
 }
}

```

## Analyzing the output

When you analyze the output on the ESB console, you will see an entry similar to the following:

```

INFO - LogMediator To: , MessageID: ID:60868ca5-d174-11e5-b7de-f9743c9bcc9e,
Direction: request, DeliveryCounter = 1

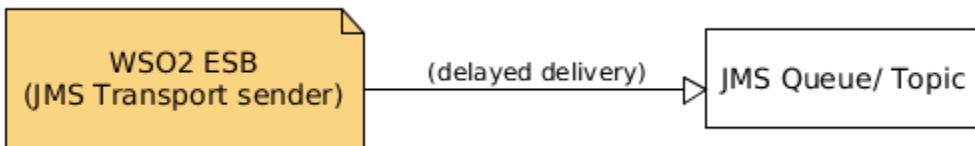
```

## Using WSO2 ESB as a JMS Producer and Specifying a Delivery Delay on Messages

In a normal message flow, JMS messages that are sent by the JMS producer to the JMS broker are forwarded to the respective JMS consumer without any delay.

With the delivery delay messaging feature introduced with JMS 2.0, you can specify a delivery delay time value in each JMS message so that the JMS broker will not deliver the message until after the specified delivery delay has elapsed. Specifying a delivery delay is useful if there is a scenario where you do not want a message consumer to receive a message that is sent until a specified time duration has elapsed. To implement this, you need to add a delivery delay to the JMS producer so that the publisher does not deliver a message until the specified delivery delay time interval is elapsed.

The following diagram illustrates how you can use WSO2 ESB as a JMS producer and specify a delivery delay on messages if you do not want the message consumer to receive a message until a specified time duration has elapsed.



To demonstrate the scenario illustrated above, we will configure WSO2 ESB's JMS transport sender using HornetQ as the message broker. This sample scenario includes the following sections:

- Prerequisites
- Configuring and starting the HornetQ message broker
- Configuring WSO2 ESB's JMS transport sender
- Building the sample scenario
- Executing the sample scenario
- Analyzing the output

## Prerequisites

- Download the `hornetq-all-new.jar` and copy it to the `<ESB_HOME>/repository/components/lib/` directory.
- Replace the `geronimo-jms_1.1_spec-1.1.0.wso2v1.jar` file in the `<ESB_HOME>/lib/endorsed/` directory with `javax.jms-api-2.0.1.jar`
- Download HornetQ from <http://hornetq.jboss.org/downloads> and extract the tar.gz.

## Configuring and starting the HornetQ message broker

1. Open the `<HornetQ_HOME>/config/stand-alone/non-clustered/hornetq-jms.xml` file in a text editor and add the following configuration:

```

<connection-factory name="QueueConnectionFactory">
 <xa>false</xa>
 <connectors>
 <connector-ref connector-name="netty" />
 </connectors>
 <entries>
 <entry name="/QueueConnectionFactory" />
 </entries>
</connection-factory>

<connection-factory name="TopicConnectionFactory">
 <xa>false</xa>
 <connectors>
 <connector-ref connector-name="netty" />
 </connectors>
 <entries>
 <entry name="/TopicConnectionFactory" />
 </entries>
</connection-factory>

<queue name="wso2">
 <entry name="/queue/mySampleQueue" />
</queue>

<topic name="sampleTopic">
 <entry name="/topic/exampleTopic" />
</topic>

```

## 2. Start the HornetQ message broker

- To start the message broker on Linux, navigate to the <HornetQ\_HOME>/bin/ directory and execute the run.sh command with root privileges.

### Configuring WSO2 ESB's JMS transport sender

- To enable the JMS transport sender, un-comment the JMS transport sender configuration in the <ESB\_HOME>/repository/conf/axis2/axis2.xml file and update the sender configuration so that it is as follows:

```

<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender">
 <parameter name="myQueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial" locked =
 "false">org.jnp.interfaces.NamingContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">jnp://localhost:1099</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter
name="transport.jms.Destination">queue/mySampleQueue</parameter>
 <parameter name="transport.jms.JMSSpecVersion">2.0</parameter>
 </parameter>

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial" locked =
 "false">org.jnp.interfaces.NamingContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">jnp://localhost:1099</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter
name="transport.jms.Destination">queue/mySampleQueue</parameter>
 <parameter name="transport.jms.JMSSpecVersion">2.0</parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
</transportSender>

```

## Building the sample scenario

The XML configuration for this sample scenario is as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <registry provider="org.wso2.carbon.mediation.registry.WSO2Registry">
 <parameter name="cachableDuration">15000</parameter>
 </registry>
 <taskManager provider="org.wso2.carbon.mediation.ntask.NTaskTaskManager"/>
 <proxy name="JMSDelivery" startOnLoad="true" trace="disable" transports="https
http">
 <description/>
 <target>
 <inSequence>
 <property name="OUT_ONLY" value="true" />
 <property name="FORCE_SC_ACCEPTED" scope="axis2" value="true" />
 <property action="remove" name="Content-Length" scope="transport" />
 <property action="remove" name="MIME-Version" scope="transport" />
 <property action="remove" name="Transfer-Encoding" scope="transport" />
 <property action="remove" name="User-Agent" scope="transport" />
 <property action="remove" name="Accept-Encoding" scope="transport" />
 <property name="messageType" scope="axis2" value="application/xml" />
 <property action="remove" name="Content-Type" scope="transport" />
 <log level="full" />

```

```

 <send>
 <endpoint>
 <address
uri="jms:transport.jms.ConnectionFactory=myQueueConnectionFactory" />
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
</target>
<publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl" />
</proxy>
<proxy name="JMSDeliveryDelayed" startOnLoad="true" trace="disable"
transports="https http">
 <description/>
 <target>
 <inSequence>
 <property name="OUT_ONLY" value="true" />
 <property name="FORCE_SC_ACCEPTED" scope="axis2" value="true" />
 <property action="remove" name="Content-Length" scope="transport" />
 <property action="remove" name="MIME-Version" scope="transport" />
 <property action="remove" name="Transfer-Encoding" scope="transport" />
 <property action="remove" name="User-Agent" scope="transport" />
 <property action="remove" name="Accept-Encoding" scope="transport" />
 <property name="messageType" scope="axis2" value="application/xml" />
 <property action="remove" name="Content-Type" scope="transport" />
 <property name="JMS_MESSAGE_DELAY" scope="axis2" value="10000" />
 <log level="full" />
 <send>
 <endpoint>
 <address
uri="jms:/transport.jms.ConnectionFactory=myQueueConnectionFactory" />
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl" />
</proxy>
<sequence name="fault">
 <!-- Log the message at the full log level with the ERROR_MESSAGE and the
ERROR_CODE-->
 <log level="full">
 <property name="MESSAGE" value="Executing default 'fault' sequence" />
 <property expression="get-property('ERROR_CODE')" name="ERROR_CODE" />
 <property expression="get-property('ERROR_MESSAGE') "
name="ERROR_MESSAGE" />
 </log>
 <!-- Drops the messages by default if there is a fault -->
 <drop/>
</sequence>
<sequence name="main">
 <in>
 <!-- Log all messages passing through -->

```

```
<log level="full"/>
<!-- ensure that the default configuration only sends if it is one of
samples -->
 <!-- Otherwise Synapse would be an open proxy by default (BAD!) -->
</sequence>
<filter regex="http://localhost:9000.*" source="get-property('To')">
 <!-- Send the messages where they have been sent (i.e. implicit "To"
EPR) -->
 <send/>
 </filter>
</in>
<out>
 <send/>
</out>
<description>The main sequence for the message mediation</description>
</sequence>
<!-- You can add any flat sequences, endpoints, etc.. to this synapse.xml file if
```

```

you do
 not want to keep the artifacts in several files -->
</definitions>

```

This configuration creates two proxy services `JMSDeliveryDelayed` and `JMSDelivery`. The `JMSDeliveryDelayed` proxy service sets a delivery delay of 10 seconds on the message that it forwards, whereas the `JMSDelivery` proxy service does not set a delivery delay on the message.

To build the sample

- Start the ESB with the sample configuration.

### Executing the sample scenario

- Run the following java file, which acts as the JMS consumer that consumes messages from the queue:

```

QueueConsumer.java

package DeliveryDelay;

import java.sql.Timestamp;
import java.util.Date;
import java.util.Properties;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.Queue;
import javax.naming.Context;
import javax.naming.InitialContext;

/**
 * Sample consumer to demonstrate JMS 2.0 feature :
 * Message Delivery Delay
 * Classic API is used
 */

public class QueueConsumer {
private static final String DEFAULT_CONNECTION_FACTORY =
"QueueConnectionFactory";
private static final String DEFAULT_DESTINATION = "queue/mySampleQueue";
private static final String INITIAL_CONTEXT_FACTORY =
"org.jnp.interfaces.NamingContextFactory";
private static final String PROVIDER_URL = "jnp://localhost:1099";

public static void main(final String[] args) {
try {
 runExample();
} catch (Exception e) {
 e.printStackTrace();
}
}

public static void runExample() throws Exception {
Connection connection = null;

```

```

Context initialContext = null;
try {

 // Step 1. Create an initial context to perform the JNDI lookup.
 final Properties env = new Properties();
 env.put(Context.INITIAL_CONTEXT_FACTORY, INITIAL_CONTEXT_FACTORY);
 env.put(Context.PROVIDER_URL, System.getProperty(Context.PROVIDER_URL,
PROVIDER_URL));
 initialContext = new InitialContext(env);

 // Step 2. perform a lookup on the Queue
 Queue queue = (Queue) initialContext.lookup(DEFAULT_DESTINATION);

 // Step 3. perform a lookup on the Connection Factory
 ConnectionFactory cf = (ConnectionFactory)
initialContext.lookup(DEFAULT_CONNECTION_FACTORY);

 // Step 4. Create a JMS Connection
 connection = cf.createConnection();

 // Step 5. Create a JMS Session
 Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

 // Step 6. Create a JMS Message Consumer
 MessageConsumer messageConsumer = session.createConsumer(queue);

 // Step 7. Start the Connection
 connection.start();
 System.out.println("JMS consumer stated on the queue " + DEFAULT_DESTINATION +
"\n");

 //Clear the queue, if there is any previous messages in the queue
 TextMessage tempMessage;
 do{
 tempMessage = (TextMessage) messageConsumer.receive(1);
 } while(tempMessage != null);

 // Step 8.1. Receive the message one
 TextMessage firstMessage = (TextMessage) messageConsumer.receive();
 long first = System.currentTimeMillis();
 System.out.println("Consumer received message: ["+new Timestamp(new
Date(first).getTime())+"] " + firstMessage.getText() + "\n");

 // Step 8.2. Receive delayed
 TextMessage secondMessage = (TextMessage) messageConsumer.receive();
 long second = System.currentTimeMillis();
 System.out.println("Consumer received dealyed message: ["+new Timestamp(new
Date(second).getTime())+"] " + secondMessage.getText() + "\n");
 System.out.println("Time difference between two messages :
"+(second-first)/1000+"s");
 } finally {

 // Step 9. Close JMS resources
 if (connection != null) {
 connection.close();
}

```

```
}

// Also the initialContext
if (initialContext != null) {
 initialContext.close();
}
```

}

- Run the following command from <ESB\_HOME>/sample/axis2Client to invoke the two proxy services:

```
ant stockquote -Daddurl=http://localhost:8280/services/JMSDelivery
-Dmode=placeorder -Dsymbol=MSFT && ant stockquote
-Daddurl=http://localhost:8280/services/JMSDeliveryDelayed -Dmode=placeorder
-Dssymbol=MSFT
```

## Analyzing the output

You will see that two messages are received by the Java consumer with a time difference of more than 10s.

This is because the `JMSDeliveryDelayed` proxy service sets a delivery delay of 10 seconds on the message that it forwards, whereas the `JMSDelivery` proxy service does not set a delivery delay on the message.

## JMS Samples

The [Samples](#) section of the ESB documentation contains several JMS integration sample scenarios. Given below are references to them.

- Sample 250: Introduction to Switching Transports
  - Sample 251: Switching from HTTP(S) to JMS
  - Sample 252: Pure Text (Binary) and POX Message Support with JMS
  - Sample 253: Bridging from JMS to HTTP and Replying with a 202 Accepted Response
  - Sample 260: Switch from FIX to AMQP
  - Sample 263: Transport Switching - JMS to http/s Using JBoss Messaging(JBM)
  - Sample 264: Sending Two-Way Messages Using JMS transport
  - Sample 381: Class Mediator to CBR Binary Messages

## Advanced Topics

This section covers few advanced JMS configurations as follows.

- JMS Security Management
  - JMS Transactions
  - Working with Multiple Types of Brokers
  - JMS MapMessage Support

JMS Security Management

JMS is an integral part of enterprise integration solutions that are highly-reliable, loosely-coupled and asynchronous. As a result, implementing proper security to your JMS deployments is vital. This document discusses some of the best practices of an effective JMS security implementation, when used in combination with WSO2 ESB.

Let's see how some of the key concepts of system security such as authentication, authorization and availability are implemented in different types of broker servers.

- Security in WSO2 Message Broker/Apache Qpid
  - Security in Apache ActiveMQ

Security in WSO2 Message Broker/Apache Qpid

Given below is an overview of how some common security concepts are implemented in WSO2 Message Broker.

Security Concept	How it is Implemented in WSO2 MB
Authentication	Andes Authenticator connected entities to authenticate.
Authorization	Creation and use of role-based permissions.
Availability	Clustering using Apache Zookeeper.
Integrity	Message-level encryption using WS-Security.

Let's see how each concept in the table above is implemented in WSO2 MB.

To set up WSO2 MB with WSO2 ESB, refer to section [Configure with WSO2 Message Broker](#). Also, open <MB\_HOME>/repository/conf/advanced/qpid-config.xml file and add the following line as a child element of <tuning>

```
<messageBatchSizeForBrowserSubscriptions>100000</messageBatchSizeForBrowserSubscriptions>
```

Authentication: Plain Text

WSO2 MB requires all its incoming connections to be authenticated. The <ESB\_HOME>/repository/conf/jndi.properties file contains lines similar to the following. They contain the username and password credentials used to authenticate connections made to the WSO2 MB. This is plain text authentication.

```
connectionfactory.TopicConnectionFactory =
amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5672'
connectionfactory.QueueConnectionFactory =
amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5672'
```

In the WSO2 MB authentication example below, we send a request to the proxy service **testJMSProxy**, which adds a message to the **example.MyQueue** queue.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <registry provider="org.wso2.carbon.mediation.registry.WSO2Registry">
 <parameter name="cachableDuration">15000</parameter>
 </registry>
 <proxy name="testJMSProxy"
 transports="https http"
 startOnLoad="true"
 trace="disable">
 <target>
 <inSequence>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
 <property name="target.endpoint" value="jmsEP" scope="default"/>
 <store messageStore="testMsgStore"/>
 </inSequence>
 </target>
 </proxy>
 <endpoint name="jmsEP">
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default 'fault' sequence"/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
 </sequence>
 <sequence name="main">
 <in>
 <log level="full"/>
 <filter source="get-property('To')" regex="http://localhost:9000.*">
 <send/>
 </filter>
 </in>
 <out>
 <send/>
 </out>
 <description>The main sequence for the message mediation</description>
 </sequence>
 <messageStore class="org.wso2.carbon.message.store.persistence.jms.JMSMessageStore"
 name="testMsgStore">
 <parameter
name="java.naming.factory.initial">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 <parameter
name="java.naming.provider.url">repository/conf/jndi.properties</parameter>
 <parameter name="store.jms.destination">MyQueue</parameter>
 </messageStore>
</definitions>

```

If you change the authentication credentials of the jndi.properties file, the connection will not be authenticated. You will see an error similar to:

```
ERROR - AMQConnection Throwabe Received but no listener set:
org.wso2.andes.AMQDisconnectedException: Server closed connection and reconnection not
permitted.
```

Authentication: Encrypted

In the [previous authentication example](#), the user names and passwords are stored in plain text inside the WSO2 ESB's jndi.properties file. These credentials can be [stored in an encrypted manner](#) for added security.

Authorization

WSO2 MB allows user-based authorization as seen in the [example on WSO2 MB Authentication](#). To set up users, follow the instructions in [User Management section](#) of the WSO2 MB documentation.

WSO2 MB provides role-based authorization for topics, where public/subscribe access can be assigned to user groups. For more information on setting up role-based authorization for topics, refer to section [Managing Topics and Subscriptions section](#) of the WSO2 MB documentation.

Integrity

Integrity is part of message-level security, and can be implemented using a standard like WS-Security. Refer to the section on [Integrity in ActiveMQ](#) to see how message-level security works over JMS.

Next, let's see how security is implemented in Apache ActiveMQ: [Security in Apache ActiveMQ](#).

#### Security in Apache ActiveMQ

Given below is an overview of how some common security concepts are implemented in Apache ActiveMQ.

Security Concept	How it is Implemented
Authentication	Simple authentication and JAAS plugins.
Authorization	Built-in authorization mechanism using XML configuration.
Availability	Master/Slave configurations using fail-over transport in ActiveMQ (not to be confused with WSO2 ESB transports).
Integrity	WS-Security

Authentication

Simple Authentication: ActiveMQ comes with an authentication plugin, which provides basic authentication between the ActiveMQ JMS and the WSO2 ESB. The steps below describe how to configure.

1. Add the following configuration in <ACTIVEMQ\_HOME>/conf/activemq-security.xml file.

```
<simpleAuthenticationPlugin anonymousAccessAllowed="true">
 <users>
 <authenticationUser username="system" password="${activemq.password}" groups="users,admins"/>
 <authenticationUser username="user" password="${guest.password}" groups="users"/>
 <authenticationUser username="guest" password="${guest.password}" groups="guests"/>
 </users>
</simpleAuthenticationPlugin>
```

2. Edit <ACTIVEMQ\_HOME>/conf/credentials.properties file for plain-text version or <ACTIVEMQ\_HOME>/conf/credentials-enc.properties file for encrypted version to define the username and password lists referenced in the configuration above.

The **anonymousAccessAllowed** attribute defines whether or not to allow anonymous access. The groups and users defined in step 1 are used to provide authorization schemes. Refer to section [Authorization](#) for more information.

3. Ensure that the <transportReceiver> element below is added in < ESB\_HOME>/repository/conf/axis2/axis2.xml file.

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
 <parameter name="transport.jms.UserName">system</parameter>
 <parameter name="transport.jms.Password">manager</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">TopicConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 </parameter>

 <parameter name="myQueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
 <parameter name="transport.jms.UserName">system</parameter>
 <parameter name="transport.jms.Password">manager</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
 <parameter name="transport.jms.UserName">system</parameter>
 <parameter name="transport.jms.Password">manager</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>
</transportReceiver>
```

Lines similar to the following contain the username and password configured in ActiveMQ.

```

<parameter name="transport.jms.UserName">system</parameter>
<parameter name="transport.jms.Password">manager</parameter>
```

## Infor

For more advanced authentication schemes that use JAAS which are supported in ActiveMQ, refer to the official ActiveMQ documentation here: <http://activemq.apache.org/security.html>

### Authorization

ActiveMQ provides authorization schemes using simple XML configurations, which you can apply to the users defined in the [authentication plugin](#). To setup authorization, ensure you have the following configuration in <ACTIVEMQ\_HOME>/conf/activemq-security.xml file.

```
<authorizationPlugin>
 <map>
 <authorizationMap>
 <authorizationEntries>
 <authorizationEntry queue="" read="admins" write="admins" admin="admins" />
 <authorizationEntry queue="USERS.>" read="users" write="users" admin="users" />
 <authorizationEntry queue="GUEST.>" read="guests" write="guests,users" admin="guests,users" />

 <authorizationEntry queue="TEST.Q" read="guests" write="guests" />

 <authorizationEntry topic="" read="admins" write="admins" admin="admins" />
 <authorizationEntry topic="USERS.>" read="users" write="users" admin="users" />
 <authorizationEntry topic="GUEST.>" read="guests" write="guests,users" admin="guests,users" />

 <authorizationEntry topic="ActiveMQ.Advisory.>" read="guests,users" write="guests,users" admin="guests,users" />
 </authorizationEntries>
 </authorizationMap>
 </map>
</authorizationPlugin>
```

## Infor

This configuration defines role-based authorization on queues and topics, and uses ActiveMQ wildcards. For information on wildcards , refer to ActiveMQ documentation here: <http://activemq.apache.org/wildcards.html>.

### Availability

ActiveMQ supports the use of master/slave and fail-over transport to provide high-availability. ActiveMQ supports two types of master/slave configurations as follows:

- Master/slave using shared file systems
- Master/slave using JDBC

## Infor

For more information on either model, refer to ActiveMQ documentation on master/slave here: <http://activemq.apache.org/master-slave.html>.

We explore the second option here.

Master/slave using JDBC

ActiveMQ uses a special URI similar to the following to facilitate fail-over functionality: `failover://(tcp://127.0.0.1:61616,tcp://127.0.0.1:61617,tcp://127.0.0.1:61618)?initialReconnectDelay=100`. Use this URI inside WSO2 ESB for a highly-available JMS solution.

To create proxy services, sequences, endpoints, message stores, processors etc. in ESB, you can either use the management console or copy the XML configuration to the source view. You can find the source view under menu **Manage > Service Bus > Source View** in the left navigation pane of the WSO2 ESB management console. Alternatively, you can add an XML file to `<ESB_HOME>/repository/deployment/server/synapse-configs/default/proxy-services`.

A sample WSO2 ESB Proxy service for this setup is given below.

```

<proxy xmlns="http://ws.apache.org/ns/synapse" name="FailOverJMS"
transport="http" startOnLoad="true" trace="disable">
 <target>
 <inSequence>
 <log level="full"/>
 <property name="OUT_ONLY" value="true" scope="default"/>
 <clone>
 <target>
 <endpoint>
 <address
uri="jms:/OMS?transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory&java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory&java.naming.provider.url=failover:(tcp://localhost:61616,tcp://localhost:61617)?randomize=false&transport.jms.DestinationType=queue"/>
 </endpoint>
 </target>
 </clone>
 </inSequence>
 </target>
 <publishWSDL key="gov:/services/FileService.wsdl">
 <resource location="Message.xsd" key="gov:/services/Message.xsd"/>
 </publishWSDL>
</proxy>

```

Note `java.naming.provider.url=failover:(tcp://localhost:61616,tcp://localhost:61617)?randomize=false` inside the address endpoint uri attribute. The `randomize=false` parameter makes this setup follow a prioritized fail-over configuration, which means when the first instance fails, it moves to the next. For more information on ActiveMQ fail-over transport and its parameters, refer to ActiveMQ documentation here: <http://activemq.apache.org/failover-transport-reference.html>.

**Integrity**

Integrity is part of message-level security, and can be implemented using a standard like WS-Security. Following sample shows the application of WS-Security for message-level encryption where messages are stored in a message store in WSO2 ESB.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <localEntry key="sec_policy"
 src="file:repository/samples/resources/policy/policy_3.xml"/>

 <in>
 <send>
 <endpoint>
 <address
uri="jms:/StockQuoteJmsProxy2?transport.jms.ConnectionFactoryJNDIName=QueueConnectionF
actory&java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFact
ory&java.naming.provider.url=tcp://localhost:61616">
 <enableSec policy="sec_policy"/>
 <enableAddressing/>
 </address>
 </endpoint>
 </send>
 </in>
 <out>
 <header name="wsse:Security" action="remove"
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-
1.0.xsd"/>
 <send/>
 </out>
 </definitions>

```

## JMS Transactions

In addition to the [transaction mediator](#), WSO2 ESB also supports JMS transactions.

### Note

In WSO2 ESB, JMS transactions only work with either the Callout mediator or the Call mediator in blocking mode.

The [JMS transport](#) shipped with WSO2 ESB supports both local and distributed JMS transactions. You can use local transactions to group messages received in a JMS queue. Local transactions are not supported for messages sent to a JMS queue.

This section describes:

- [JMS Local Transactions](#)
- [JMS Distributed Transactions](#)

### JMS local transactions

A **local transaction** represents a unit of work on a single connection to a data source managed by a resource manager. In JMS, you can use the JMS API to get a transacted session and to call methods for commit or roll back for the relevant transaction objects. This is managed internally by a resource manager. There is no external transaction manager involved in the coordination of such transactions.

Let's explore a sample scenario that demonstrates how to handle a transaction using JMS in a situation where the back-end service is unreachable.

---

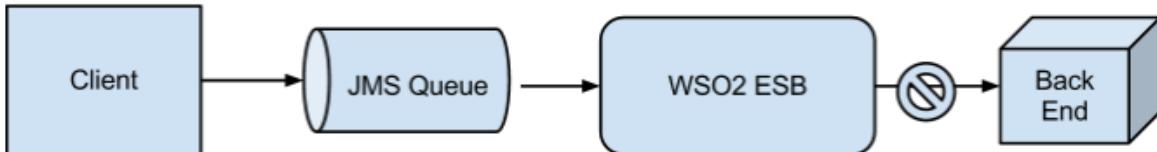
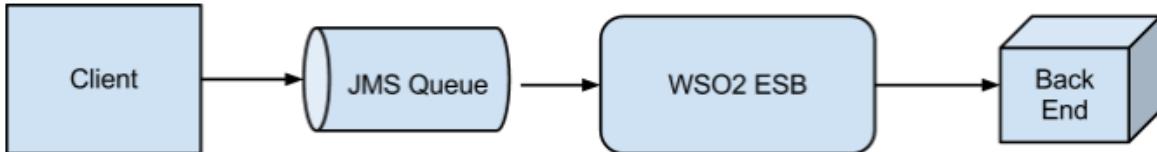
[Sample scenario](#) | [Prerequisites](#) | [Configuring the sample scenario](#) | [Executing the sample scenario](#) | [Testing the](#)

[sample scenario](#) | [Sample Scenario](#) | [Prerequisites](#) | [Configuring the sample scenario](#)

### Sample scenario

A message is read from a JMS queue and is processed by a back-end service. In the successful scenario, the transaction will be committed and the request will be sent to the back end service. In the failure scenario, while executing a sequence, a failure occurs and the ESB receives a fault. This cause the JMS transaction to roll back.

The sample scenario can be depicted as follows:



### Prerequisites

- Windows, Linux or Solaris operating systems with WSO2 ESB installed. For instructions on downloading and installing WSO2 ESB, see [Installation Guide](#).
- WSO2 Message Broker installed. For instructions on downloading and installing WSO2 Message Broker, see [Getting Started with WSO2 Message Broker](#).
- WSO2 ESB's JMS transport configured with WSO2 Message Broker. For instructions on configuring WSO2 ESB's JMS transport with WSO2 Message Broker, see [Configure with WSO2 Message Broker](#).

### Configuring the sample scenario

1. Configure the JMS local transaction by defining the following parameter in the `<ESB HOME>/repository/conf/axis2/axis2.xml` file. By default the session is not transacted. In order to make it transacted, set the parameter to **true**.

```

<parameter name="transport.jms.SessionTransacted">true</parameter>

```

Once done, the JMS listener configuration for WSO2 MB in the `axis2.xml` file should be as follows:

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 >
 <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">TopicConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 <parameter name="transport.jms.SessionTransacted">true</parameter>
 <parameter name="transport.jms.SessionAcknowledgement"
locked="true">CLIENT_ACKNOWLEDGE</parameter>
 </parameter>

 <parameter name="myQueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 >
 <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.SessionTransacted">true</parameter>
 <parameter name="transport.jms.SessionAcknowledgement"
locked="true">CLIENT_ACKNOWLEDGE</parameter>
 </parameter>

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 >
 <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 <parameter name="transport.jms.SessionTransacted">true</parameter>
 <parameter name="transport.jms.SessionAcknowledgement"
locked="true">CLIENT_ACKNOWLEDGE</parameter>
 </parameter>
</transportReceiver>

```

2. Copy and paste the following configuration into the Synapse configuration in <ESB\_HOME>/repository/deployment/server/synapse-configs/<node>/synapse.xml.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="jms" startOnLoad="true">
 <target>
 <inSequence>
 <property name="OUT_ONLY" value="true"/>
 <callout
 serviceURL="http://localhost:9000/services/SimpleStockQuoteService">
 <source type="envelope"/>
 <target key="placeOrder"/>
 </callout>
 <log level="custom">
 <property name="Transaction Action" value="Committed"/>
 </log>
 </inSequence>
 <faultSequence>
 <property name="SET_ROLLBACK_ONLY" value="true" scope="axis2"/>
 <log level="custom">
 <property name="Transaction Action" value="Rolledback"/>
 </log>
 </faultSequence>
 </target>
 <parameter name="transport.jms.ContentType">
 <rules>
 <jmsProperty>contentType</jmsProperty>
 <default>application/xml</default>
 </rules>
 </parameter>
 </proxy>
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default "fault" sequence"/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE"
 expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
 </sequence>
 <sequence name="main">
 <log/>
 <drop/>
 </sequence>
</definitions>

```

According to the above configuration, a message will be read from the JMS queue and will be sent to the SimpleStockQuoteService running on the Axis2 back-end server. If a failure occurs, the transaction will roll back.

In the above configuration, the following property is set to **true** in the fault handler, in order to roll back the transaction when a failure occurs.

```
<property name="SET_ROLLBACK_ONLY" value="true" scope="axis2"/>
```

3. Deploy the back-end service SimpleStockQuoteService . For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

#### 4. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).

You now have a running ESB instance, WSO2 Message Broker instance and a sample back-end service to simulate the sample scenario. Now let's execute the JMS client.

Due to the asynchronous behavior of the [Send Mediator](#), you cannot use it with a http/https endpoint, but you can use it in asynchronous use cases, for example with another JMS as endpoint.

#### ***Executing the sample scenario***

##### **To execute the JMS client**

- Run the following command from the <ESB\_HOME>/samples/axis2Client directory.

```
ant jmsclient -Djms_type=pox -Djms_dest=dynamicQueues/StockQuoteProxy
-Djms_payload=MSFT
```

This will trigger a sample message to the JMS Server.

#### ***Testing the sample scenario***

##### **Successful scenario**

In order to simulate the successful scenario, the message should mediate successfully.

When the message mediates successfully, the output on the Axis2 server start-up console will be as follows:

```
[2013-02-20 11:27:29 IST 2013 samples.services.SimpleStockQuoteService :: Accepted order #1 for : 18926 stocks of MSFT at $ 170.765795426708
```

The ESB debug log will display an INFO message as follows, indicating that the transaction is committed.

```
[2013-02-20 11:34:24,188] INFO - LogMediator Transaction Action = Committed
```

##### **Failure Scenario**

In order to simulate the failure scenario, you need to stop the sample Axis2 Server and [execute the JMS client](#) once again.

In this scenario, the ESB debug log will display an INFO message as follows, indicating that the transaction is rolled back.

```
[2013-02-20 11:54:53,938] INFO - LogMediator Transaction Action = Rollbacked
```

#### **JMS distributed transactions**

WSO2 ESB also supports distributed JMS transactions. You can use the JMS transport with more than one distributed resource, for example, two remote database servers. An external transaction manager coordinates the transaction. Designing and using JMS distributed transactions is more complex than using local JMS transactions.

The transaction manager is the primary component of the distributed transaction infrastructure and distributed JMS transactions are managed by the [XAResource](#) enabled transaction manager in the Java 2 Platform, Enterprise

Edition (J2EE) application server.

You will need to check if your message broker supports [XA transactions](#) prior to implementing distributed JMS transactions.

## XA two-phase commit process

XA is a two-phase commit specification that is used in distributed transaction processing.

» [Click here to expand...](#)

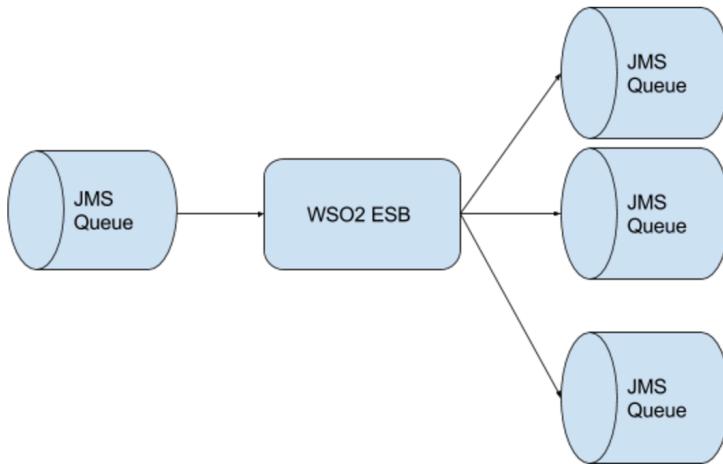
Currently WSO2 Message Broker (WSO2 MB) does not support JMS distributed transactions.

Let's look at a sample scenario for JMS distributed transactions.

[ [JMS local transactions](#) ] [ [Sample scenario](#) ] [ [Prerequisites](#) ] [ [Configuring the sample scenario](#) ] [ [Executing the sample scenario](#) ] [ [Testing the sample scenario](#) ] [ [JMS distributed transactions](#) ] [ [XA two-phase commit process](#) ] [ [Sample Scenario](#) ] [ [Prerequisites](#) ] [ [Configuring the sample scenario](#) ]

### **Sample Scenario**

ESB listens to the message queue and sends that message to multiple queues. If something goes wrong in sending the message to one of those queues, the original message should be rolled back to the listening queue and none of the other queues should receive the message. Thus, the entire transaction should be rolled back.



### **Prerequisites**

- Windows, Linux or Solaris operating systems with WSO2 ESB installed. For instructions on downloading and installing WSO2 ESB, see [Installation Guide](#).
- WSO2 ESB's JMS transport configured with ActiveMQ. For instructions, see [Configure with ActiveMQ](#).

### **Configuring the sample scenario**

1. Create the `JMSListenerProxy` proxy service in ESB with the following configuration:

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="JMSListenerProxy"
 transports="https http jms"

```

```

 startOnLoad="true">
<description/>
<target>
 <inSequence>
 <property name="OUT_ONLY" value="true" />
 <log level="custom">
 <property name="MESSAGE_ID_A"
expression="get-property('MessageID')"/>
 </log>
 <log level="custom">
 <property name="BEFORE" expression="$body" />
 </log>
 <property name="MESSAGE_ID_B"
 expression="get-property('MessageID')"
 scope="operation"
 type="STRING" />
 <property name="failureResultProperty"
 scope="default"
 description="FailureResultProperty">
 <result xmlns="">failure</result>
 </property>
 <enrich>
 <source clone="true" xpath="$ctx:failureResultProperty" />
 <target type="body" />
 </enrich>
 <log level="custom">
 <property name="AFTER" expression="$body" />
 </log>
 <property name="BEFORE1" value="ABCD" scope="axis2" type="STRING" />
 <callout
serviceURL="jms:/ActiveMQPublisher1?transport.jms.ConnectionFactoryJNDIName=XAConnectionFactory&java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory&java.naming.provider.url=tcp://localhost:61616&transport.jms.DestinationType=queue;transport.jms.TransactionCommand=begin">
 <source type="envelope" />
 <target xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
 xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
 xpath="s11:Body/child::*[fn:position()=1] |
s12:Body/child::*[fn:position()=1]" />
 </callout>
 <callout
serviceURL="jms:/ActiveMQPublisher2?transport.jms.ConnectionFactoryJNDIName=XAConnectionFactory&java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory&java.naming.provider.url=tcp://localhost:61616&transport.jms.DestinationType=queue">
 <source type="envelope" />
 <target xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
 xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
 xpath="s11:Body/child::*[fn:position()=1] |
s12:Body/child::*[fn:position()=1]" />
 </callout>
 <callout
serviceURL="jms:/ActiveMQPublisher3?transport.jms.ConnectionFactoryJNDIName=XAConnectionFactory&java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory&java.naming.provider.url=tcp://localhost:61616&transport.jms.DestinationType=queue;transport.jms.TransactionCommand=end">
 <source type="envelope" />
 <target xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
 xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/">

```

```
 xpath="s11:Body/child::*[fn:position()=1] |
s12:Body/child::*[fn:position()=1]"/>
 </callout>
 <drop/>
 </inSequence>
 <faultSequence>
 <log level="custom">
 <property name="Transaction Action" value="Rolledback" />
 </log>
 <callout
serviceURL="jms:/ActiveMQPublisherFault?transport.jms.ConnectionFactoryJNDIName=X
AConnectionFactory& java.naming.factory.initial=org.apache.activemq.jndi.Active
eMQInitialContextFactory& java.naming.provider.url=tcp://localhost:61616& t
ransport.jms.DestinationType=queue;transport.jms.TransactionCommand=rollback">
 <source type="envelope"/>
 <target xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
 xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
 xpath="s11:Body/child::*[fn:position()=1] |
s12:Body/child::*[fn:position()=1]"/>
 </callout>
 </faultSequence>
 </target>
 <parameter name="transport.jms.ContentType">
 <rules>
 <jmsProperty>contentType</jmsProperty>
 <default>application/xml</default>
 </rules>
```

```

</parameter>
<parameter name="transport.jms.Destination">MyJMSQueue</parameter>
</proxy>

```

In the above configuration, ESB listens to a JMS queue named MyJMSQueue and consumes messages as well as sends messages to multiple JMS queues in a transactional manner.

2. To place a message into MyJMSQueue, execute the following command from <ESB\_HOME>/samples/axis2Client directory:

```

ant stockquote -Dmode=placeorder
-Dtrpurl="jms:/MyJMSQueue?transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory&java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory&java.naming.provider.url=tcp://localhost:61616&transport.jms.ContentTypeProperties=Content-Type&transport.jms.DestinationType=queue"

```

You can see how ESB consumes messages from the queue named MyJMSQueue and sends the messages to multiple queues.

To check the rollback functionality provide an unreachable host name to any destination queue and save the configurations. You should be able to observe ESB fault sequence getting invoked and failed message delivered to the destination configured in fault sequence.

## Working with Multiple Types of Brokers

This section explains how you can configure a single ESB instance to make use of multiple message brokers.

1. First, copy all client libraries that come with each JMS broker to <ESB\_HOME>/repository/components/lib directory.

## Client Library Conflicts

When you add client libraries of the JMS brokers to the ESB classpath, you might encounter conflicts with existing libraries from different vendors. You won't be able to integrate the message brokers with the ESB without resolving these conflicts, if any.

2. Configure <ESB\_HOME>/repository/conf/axis2/axis2.xml file for multiple JMS transports. As we are dealing with more than one message broker here, we have to define separate JMS transport listeners for each broker. To do that, edit the axis2.xml file as follows.

In the following code, instead of one jms transport listener, we now have 2 transport listeners as **jms1** and **jms2** configured. In this example, they listen to ActiveMQ and WSO2MB respectively. You can similarly configure for other JMS brokers as well.

```

<!--Uncomment this and configure as appropriate for JMS transport support, after
setting up your JMS environment (e.g. ActiveMQ)-->
<transportReceiver name="jms1" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">TopicConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"

```

```

locked="false">>topic</parameter>
 </parameter>

 <parameter name="myQueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>
 </transportReceiver>

<!--Uncomment this and configure as appropriate for JMS transport support with WSO2 MB
2.x.x -->
 <transportReceiver name="jms2" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">TopicConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 </parameter>

 <parameter name="myQueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>
</pre>

```

```

locked="false">queue</parameter>
</parameter>
</transportReceiver>

```

3. Expose 2 proxy services in ESB, which listen to the JMS transports just configured by adding the following code to synapse.xml. Each of the proxies defined below will listen to different message brokers configured earlier in step 2 above.

```

<proxy name="SimpleStockQuoteService1"
 transports="jms1"
 startOnLoad="true"
 trace="disable">
 ...
</proxy>

<proxy name="SimpleStockQuoteService2"
 transports="jms2"
 startOnLoad="true"
 trace="disable">
 ...
</proxy>

```

## JMS MapMessage Support

As of WSO2 ESB 4.7.0, the JMS transport supports producing and consuming JMS [MapMessage](#) objects, which send a set of name/value pairs.

### Producing a MapMessage

To send a MapMessage from a proxy service or API to a queue, construct an XML payload using the [PayloadFactor](#) mediator (or another method) in the following structure, and send it to a JMS endpoint:

```

<JMSMap xmlns="http://axis.apache.org/axis2/java/transport/jms/map-payload">
 <name1>value1</name1>
 <name2>value2</name2>
 <name3>value3</name3>
</JMSMap>

```

The JMS sender will then produce the equivalent MapMessage object:

```

MapMessage message = session.createMapMessage();
message.setString("name1", "value1");
message.setString("name2", "value2");
message.setString("name3", "value3");

```

### Consuming a MapMessage

When a proxy service receives a JMS MapMessage via a JMS broker, it will convert it to an XML message like this:

```
<JMSMap xmlns="http://axis.apache.org/axis2/java/transports/jms/map-payload">
 <name1>value1</name1>
 <name2>value2</name2>
 <name3>value3</name3>
</JMSMap>
```

## JMS Troubleshooting Guide

### **Abstract**

This troubleshooting guide helps you resolve common problems encountered in JMS integration scenarios with WSO2 ESB.

### **Contents**

- [Handling ClassNotFoundExceptions and NoClassDefFoundExceptions](#)
- [HTTP header conversion](#)
- [JMS property data type mismatch](#)
- [Too-many-threads and out-of-memory issues](#)
- [JMSUtils cannot locate destination](#)

### **Handling ClassNotFoundExceptions and NoClassDefFoundExceptions**

Check if you have deployed all the required client libraries. The missing class should be available in one of the jar files deployed in <ESB HOME>/repository/components/lib directory.

WSO2 ESB comes with geronimo-jms library, which contains the javax.jms packages. Therefore, you do not have to deploy them again.

[Back to Top ^](#)

---

### **HTTP header conversion**

When forwarding HTTP traffic to a JMS queue using WSO2 ESB, you might get an error similar to the one given below.

```
ERROR JMSender Error creating a JMS message from the axis message context
javax.jms.MessageFormatException: MQJMS1058: Invalid message property name:
Content-Type
```

This exception is specific to the JMS broker used, and is thrown by the JMS client libraries used to connect with the JMS broker.

The incoming HTTP message contains a bunch of HTTP headers that have the '-' character. Some noticeable examples are **Content-length** and **Transfer-encoding** headers. When WSO2 ESB forwards a message over JMS, it sets the headers of the incoming message to the outgoing JMS message as JMS properties. But, according to the JMS specification, the '-' character is prohibited in JMS property names. Some JMS brokers like ActiveMQ do not check this specifically, in which case there will not be any issues. But some brokers do and they throw exceptions.

The solution is to simply remove the problematic HTTP headers from the message before delivering it over JMS. You can use the property mediator as follows to achieve this:

```
<property action="remove" name="Content-Length" scope="transport">
<property action="remove" name="Accept-Encoding" scope="transport">
<property action="remove" name="User-Agent" scope="transport">
<property action="remove" name="Content-Type" scope="transport">
```

Alternatively, you can use the `transport.jms.MessagePropertyHyphens` parameter to handle hyphenated properties, instead of handling them as described above. For more information on this parameter, see [the `transport.jms.MessagePropertyHyphens` parameter description and possible values](#).

[Back to Top ^](#)

### JMS property data type mismatch

When the ESB attempts to forward a message over JMS, there are instances that the client libraries throw an exception saying the data type of a particular message property is invalid.

This problem occurs when the developer uses the property mediator to manipulate property values set on the message. Certain implementations of JMS have data type restrictions on properties. But the property mediator always sets property values as strings.

The solution is to revise the mediation sequences and avoid manipulating property values containing non-string values. If you want to set a non-string property value, write a simple custom mediator. Instructions are given in section [Creating Custom Mediators](#). For an example, to set a property named `foo` with integer value `12345`, use the property mediator as follows and set the `type` attribute to `INTEGER`. If the `type` attribute of the property is not specifically set, it will be assigned to `String` by default.

```
<property name="foo" value="12345" type="INTEGER" scope="transport"/>
```

[Back to Top ^](#)

### Too-many-threads and out-of-memory issues

With some JMS brokers, WSO2 ESB tends to spawn new worker threads indefinitely until it runs out of memory and crashes. This problem is caused by a bug in the underlying Axis2 engine. A simple workaround to this problem is to engage the property mediator of the mediation sequence as follows:

```
<property action="remove" name="transportNonBlocking" scope="axis2">
```

This prevents the ESB from creating new worker threads indefinitely. You can use a jconsole like JMX client to monitor the active threads and memory consumption of the ESB.

[Back to Top ^](#)

### JMSUtils cannot locate destination

If your topic or queue name contains the termination characters ":" or "=", JMSUtils will not be able to find the topic/queue and will give you the warning "JMSUtils cannot locate destination". (For more information, see <http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html#load%28java.io.Reader%29>.) For example, if the topic name is `my::topic`, the following configuration will not work, because the topic name will be parsed as `my` instead of `my::topic`:

```
<address uri="jms:/my::topic?transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory&&java.naming.factory.initial=org.wso2.andes.jndi.PropertiesFileInitialContextFactory&&java.naming.provider.url=repository/conf/jndi.properties&&transport.jms.DestinationType=topic"/>
```

To avoid this issue, you can create a key-value pair in the `jndi.properties` file that maps the topic/queue name to a key that either escapes these characters with a backslash (\) or does not contain ":" or "=" . For example:

```
topic.my\::topic = my::topic
```

or

```
topic.myTopic = my::topic
```

You can then use this key in the proxy service as follows:

```
<address uri="jms:/my\::topic?transport.jms.ConnectionFactoryJNDIName=TopicConnectionFactory&&java.naming.factory.initial=org.wso2.andes.jndi.PropertiesFileInitialContextFactory&&java.naming.provider.url=repository/conf/jndi.properties&&transport.jms.DestinationType=topic"/>
```

If you do not want to use the JNDI properties file, you can define the key-value pair right in the proxy configuration:

```
<address uri="jms:/my\::topic?transport.jms.ConnectionFactoryJNDIName=TopicConnectionFactory&&java.naming.factory.initial=org.wso2.andes.jndi.PropertiesFileInitialContextFactory&&topic.my\::topic=my::topic&&java.naming.provider.url=repository/conf/jndi.properties&&transport.jms.DestinationType=topic"/>
```

[Back to Top ^](#)

## JMS FAQ

### Contents

- [What is JMS?](#)
- [What are the main components of a JMS architecture?](#)
- [What is point-to-point message passing?](#)
- [What is publish/subscribe message passing?](#)
- [How to configure WSO2 ESB to behave as a message consumer?](#)
- [How to configure WSO2 ESB to behave as a message producer?](#)
- [How to use content-based routing with JMS transport?](#)
- [How to cache JMS objects with JMS transport?](#)
- [How to configure JMS as a reliable transport with acknowledgments?](#)
- [How to archive clustering with JMS?](#)
- [How to use JMS transactions?](#)
- [How to respond the client after sending a SOAP request to the JMS queue?](#)
- [Does WSO2 ESB implement JMS queues with disk persistence?](#)
- [How does WSO2 ESB manage message recalling when the destination queue cannot be reached?](#)
- [How to prevent message loss due to unavailability of a data source?](#)
- [How can you access and modify JMS headers from within a proxy service?](#)

### What is JMS?

Many modern enterprise applications are built from separate software components, which reside in a distributed or multi-tiered environment on several Java Virtual Machines. These disparate component need to communicate with each other and this is called **messaging**. JMS is one of those messaging APIs.

The JMS API provides messaging that is,

- Loosely coupled - The sender and receiver might have no information each other or the mechanisms used to process messages.
- Asynchronous - Receivers do not request messages. Messages can be delivered as they arrive. The sender does not wait for reply.
- Reliable - A message is sent and received once and only once.

[Back to Top ^](#)

## What are the main components of a JMS architecture?

The following table lists the main components.

JMS Provider	This is the messaging system that implements the JMS API interfaces and provides certain control features.
JMS Clients	System components or programs that send or receive messages.
Messaging Domains	JMS supports both the point-to-point and publish/subscribe message domains.
Messages	These are the objects used to communicate between JMS clients.
Queues	Used to hold messages in the point-to-point domain.
Topics	Used to hold messages in the publish/subscribe domain.
Connection	A client's active connection to the JMS provider
Session	One or more sessions can be created for each connection. Used to create senders and receivers and administer transactions.

[Back to Top ^](#)

## What is point-to-point message passing?

With point-to-point, the sender and receiver agree on a message destination, also known as a queue. The sender leaves the message and the receiver picks it up at any time thereafter. The message remains in the queue until the receiver removes it.

[Back to Top ^](#)

## What is publish/subscribe message passing?

With the publish/subscribe model, the sender, now called a publisher, again sends messages to an agreed destination. The destination is known as a topic by convention, and this time there might be many receivers that are called subscribers. Messages are immediately delivered to all current subscribers, and deleted when all the subscribers have received the message.

[Back to Top ^](#)

## How to configure WSO2 ESB to behave as a message consumer?

Refer to section [ESB as a JMS Consumer](#).

[Back to Top ^](#)

## How to configure WSO2 ESB to behave as a message producer?

Refer to section [ESB as a JMS Producer](#).

[Back to Top ^](#)

## How to use content-based routing with JMS transport?

There are possibly two ways to achieve this.

1. Accept the message from a JMS queue using a JMS proxy, and use the existing mediators to route the message based on the content. For example, [Filter Mediator](#).
2. Use JMS message selector. The selector expression should be provided via the parameter **transport.jms.MessageSelector** in JMS transport.

[Back to Top ^](#)

---

### How to cache JMS objects with JMS transport?

For better performance, it is important to cache JMS objects such as JMS connections, sessions, producers and consumers. JMS transport, when used with WSO2 ESB for example, has the ability to cache the mentioned JMS objects.

The parameter **transport.jms.CacheLevel** is used to configure the cache level required. It can be configured either as a proxy service parameter (when acting as a consumer), or as a parameter in the connection factory definition in <ESB\_HOME>/repository/conf/axis2/axis2.xml file's JMS transport sender element.

The parameter can have the following values.

Value	Description
auto	Uses an appropriate caching level depending on the transaction strategy. auto is the default value.
none	No JMS object will be cached.
connection	MS connections objects will be cached.
session	JMS session and connection objects will be cached.
consumer	JMS connections, sessions and consumers objects will be cached.
producer	JMS connections, sessions and producer objects will be cached.

[Back to Top ^](#)

---

### How to configure JMS as a reliable transport with acknowledgments?

The following values can be used.

Duplicates Allowed	<p>Consumer applications specify this acknowledgement mode using DUPS_OK_ACKNOWLEDGE constant defined in the Session interface. Using this mode, the session acknowledges messages lazily, which provides faster message processing times although some duplicate messages might be delivered multiple times if JMS fails.</p> <p>Recommended only for applications that are tolerant to message duplicates.</p>
--------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Auto acknowledge	<p>This is the default acknowledgement mode, which is specified using the AUTO_ACKNOWLEDGE constant defined in the Session interface. For each message, the session automatically acknowledges that a client has received a message in either of the following instances:</p> <ul style="list-style-type: none"> <li>• Right before the call to a message consumer's receive.</li> <li>• When <b>receiveNoWait</b> returns a message.</li> <li>• Right after the <b>onMessage</b> method returns successfully after invoking the consumer's MessageListener.</li> </ul> <p>If a JMS provider or the message consumer crashes while it is processing a message, the message will either be re-delivered or lost when using the automatic acknowledgement mode.</p>
Client acknowledge	<p>Consumer applications specify this acknowledgement mode using the CLIENT_ACKNOWLEDGE constant defined in the Session interface. It gives consumer more control over when messages are acknowledged. A consumer can group a number of messages, and then invoke the acknowledge method of the message to instruct the JMS provider that the message and all other messages received until this point have been consumed.</p> <p>When a consumer uses client acknowledge, it can use the recover method of the session to revert back to its last checkpoint. This causes the session to re-deliver all messages that have not yet been acknowledged by the consumer.</p> <p>If a client crashes and later re-connects to its queue or topic, the session will effectively be recovered and the consumer will receive all unacknowledged messages.</p>
Transactional Acknowledge	<p>A transacted session is a related group of consumed and produced messages that are treated as a single work unit. A transaction can be either committed or rolled back.</p> <p>When the session's <b>commit</b> method is called, the consumed messages are acknowledged, and the associated produced messages are sent. When a session's <b>rollback</b> method is called, the produced messages are destroyed, and the consumed messages are recovered. As soon as either the commit or rollback method is called, the current transaction ends and a new transaction is immediately started.</p> <p>For example, if an application moves messages from one destination to another, it should use a transacted session to ensure that the message is either successfully transferred, or not transferred at all. In this case, the commit method should be called after each message.</p>

[Back to Top ^](#)

---

### How to archive clustering with JMS?

Refer to section [Clustering with JMS](#).

[Back to Top ^](#)

---

### How to use JMS transactions?

Refer to section [JMS Transactions](#).

[Back to Top ^](#)

---

### How to respond the client after sending a SOAP request to the JMS queue?

This can be achieved with the following configuration. Replace the following line according to your end point URI: <address uri="<http://localhost:9000/services/SimpleStockQuoteService>" />.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <registry provider="org.wso2.carbon.mediation.registry.WSO2Registry">
```

```

<parameter name="cachableDuration">15000</parameter>
</registry>
<proxy name="StockQuoteProxy"
 transports="http"
 startOnLoad="true"
 trace="disable">
<description/>
<target>
 <inSequence>
 <property name="OUT_ONLY" value="true"/>
 <clone>
 <target>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </target>
 <target sequence="test"/>
 </clone>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
</proxy>
<sequence name="test">
 <payloadFactory>
 <format>
 <ns:a xmlns:ns="http://services.samples">
 <ns:b>Accepted</ns:b>
 </ns:a>
 </format>
 </payloadFactory>
 <property name="HTTP_SC" value="202" scope="axis2"/>
 <header name="To" action="remove"/>
 <property name="RESPONSE" value="true"/>
 <send/>
</sequence>
<sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default 'fault' sequence"/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
</sequence>
<sequence name="main">
 <in>
 <log level="full"/>
 <filter source="get-property('To')" regex="http://localhost:9000.*">
 <send/>
 </filter>
 </in>
 <out>
 <send/>
 </out>
</sequence>

```

```

<description>The main sequence for the message mediation</description>
</sequence>
</definitions>

```

[Back to Top ^](#)

### Does WSO2 ESB implement JMS queues with disk persistence?

If the broker supports JMS message persistence, the ESB can read/write from a persistence message queue.

[Back to Top ^](#)

### How does WSO2 ESB manage message recalling when the destination queue cannot be reached?

You can define message stores to handle failed messages.

[Back to Top ^](#)

### How to prevent message loss due to unavailability of a data source?

If you have sent messages through JMS endpoint to an unavailable data services server, messages will get queued at the JMS broker. When you start the data services server soon after, the JMS queue will be consumed by the server. But, there can be a message loss due to the unavailability of the data source if it is configured as a Carbon Data Source.

You can avoid this message loss by setting the following parameters in the data service configuration.

1. Click on the data service in the service list, and navigate to service description page.
2. Add the following parameters to the data service. If the ESB's management console UI is used to add them, you can add the name first and then enter the values after saving.

```

<parameter name="transport.jms.SessionAcknowledgement"
locked="false">CLIENT_ACKNOWLEDGE</parameter>

<parameter name="transport.jms.CacheLevel" locked="false">none</parameter>

```

This will ensure that no messages will be lost even if the data source is not available at the initialization. You might notice exceptions in the log but the inserts will be done to the database accordingly.

[Back to Top ^](#)

### How can you access and modify JMS headers from within a proxy service?

You can set custom values for JMS headers from within a proxy service using the [Property mediator](#) as follows:

```

<property name="JMS_PRIORITY" value="2" scope="transport" />
<property name="JMS_COORELATION_ID" value="1234567" scope="transport" />
<property name="JMS_MESSAGE_ID" value="CustomMessageID" scope="transport" />

```

You can then retrieve these values using [get-property](#) function (e.g., `get-property('transport', 'JMS_PRIORITY')`).

Ability to modify JMS headers depends on the JMS provider, where there may be limitations in modifying

some headers when using certain providers. In such cases, even if the header values are set within the proxy service, they would not get modified as expected.

[Back to Top ^](#)

## JSON Support

WSO2 ESB provides support for [JavaScript Object Notation \(JSON\)](#) payloads in messages. The following sections describe how to work with JSON in the ESB:

- [JSON message builders and formatters](#)
- [XML representation of JSON payloads](#)
- [Converting a payload between XML and JSON](#)
- [Accessing content from JSON payloads](#)
- [Logging JSON payloads](#)
- [Constructing and transforming JSON payloads](#)
- [XML to JSON transformation parameters](#)
- [Validating JSON messages](#)
- [Troubleshooting, debugging, and logging](#)

### JSON message builders and formatters

The ESB provides two types of message builders and formatters for JSON. The default builder and formatter keep the JSON representation intact without converting it to XML. You can access the payload content using JSON Path or XPath and convert the payload to XML at any point in the mediation flow.

- `org.apache.synapse.commons.json.JsonStreamBuilder`
- `org.apache.synapse.commons.json.JsonStreamFormatter`

If you want to convert the JSON representation to XML before the mediation flow begins, use the following builder and formatter instead. Note that some data loss can occur during the JSON to XML to JSON conversion process.

- `org.apache.synapse.commons.json.JsonBuilder`
- `org.apache.synapse.commons.json.JsonFormatter`

The builders and formatters are configured in the `messageBuilders` and `messageFormatters` sections, respectively, of the Axis2 configuration files located in the `<ESB_HOME>/repository/conf/axis2` directory. Both types of JSON builders use [StAXON](#) as the underlying JSON processor.

The following builders and formatters are also included for compatibility with previous versions of the ESB:

- `org.apache.axis2.json.JSONBuilder/JSONMessageFormatter`
- `org.apache.axis2.json.JSONStreamBuilder/JSONStreamFormatter`
- `org.apache.axis2.json.JSONBadgerfishOMBuilder/JSONBadgerfishMessageFormatter`

Always use the same type of builder and formatter combination. Mixing different builders and formatters will cause errors at runtime.

If you want the ESB to handle JSON payloads that are sent using a media type other than `application/json`, you must register the JSON builder and formatter for that media type in the following two files at minimum (for best results, register them in all Axis2 configuration files found in the `<ESB_HOME>/repository/conf/axis2` directory):

- `<ESB_HOME>/repository/conf/axis2/axis2.xml`
- `<ESB_HOME>/repository/conf/axis2/axis2_blocking_client.xml`

For example, if the media type is `text/javascript`, register the message builder and formatter as follows:

```
<messageBuilder contentType="text/javascript"
 class="org.apache.synapse.commons.json.JsonStreamBuilder"/>

<messageFormatter contentType="text/javascript"
 class="org.apache.synapse.commons.json.JsonStreamFormatter"/>
```

When you modify the builders/formatters in Axis2 configuration, make sure that you have enabled only one correct message builder/formatter pair for a given media type.

### XML representation of JSON payloads

When building the XML tree, JSON builders attach the converted XML infoset to a special XML element that acts as the root element of the final XML tree. If the original JSON payload is of type `object`, the special element is `<json Object/>`. If it is an `array`, the special element is `<jsonArray/>`. Following are examples of JSON and XML representations of various objects and arrays.

#### Null objects

JSON:

```
{"object":null}
```

XML:

```
<j(jsonObject>
 <object></object>
</j(jsonObject>
```

#### Empty objects

JSON:

```
{"object":{}}
```

XML:

```
<j(jsonObject>
 <object></object>
</j(jsonObject>
```

#### Empty strings

JSON:

```
{ "object": "" }
```

**XML:**

```
<j(jsonObject>
 <object></object>
</j(jsonObject>
```

### Empty array

**JSON:**

```
[]
```

**XML (JsonStreamBuilder):**

```
<j(jsonArray></j(jsonArray>
```

**XML (JsonBuilder):**

```
<j(jsonArray>
 <?xml-multiple jsonElement?>
</j(jsonArray>
```

### Named arrays

**JSON:**

```
{ "array": [1, 2] }
```

**XML (JsonStreamBuilder):**

```
<j(jsonObject>
 <array>1</array>
 <array>2</array>
</j(jsonObject>
```

**XML (JsonBuilder):**

```
<j(jsonObject>
 <?xml-multiple array?>
 <array>1</array>
 <array>2</array>
</j(jsonObject>
```

JSON:

```
{ "array" : [] }
```

XML (JsonStreamBuilder):

```
<jsonObject></jsonObject>
```

XML (JsonBuilder):

```
<jsonObject>
<?xml-multiple array?>
</jsonObject>
```

## Anonymous arrays

JSON:

```
[1,2]
```

XML (JsonStreamBuilder):

```
<jsonArray>
<jsonElement>1</jsonElement>
<jsonElement>2</jsonElement>
</jsonArray>
```

XML (JsonBuilder):

```
<jsonArray>
<?xml-multiple jsonElement?>
<jsonElement>1</jsonElement>
<jsonElement>2</jsonElement>
</jsonArray>
```

JSON:

```
[1, []]
```

XML (JsonStreamBuilder):

```
<jsonArray>
<jsonElement>1</jsonElement>
<jsonElement>
 <jsonArray></jsonArray>
</jsonElement>
</jsonArray>
```

**XML (JsonBuilder):**

```
<jsonArray>
<?xml-multiple jsonElement?>
<jsonElement>1</jsonElement>
<jsonElement>
 <jsonArray>
 <?xml-multiple jsonElement?>
 </jsonArray>
</jsonElement>
</jsonArray>
```

## XML processing instructions (PIs)

Note the addition of `xml-multiple` processing instructions to the XML payloads whose JSON representations contain arrays. `JsonBuilder` (via StAXON) adds these instructions to the XML payload that it builds during the JSON to XML conversion so that during the XML to JSON conversion, `JsonFormatter` can reconstruct the arrays that are present in the original JSON payload. `JsonFormatter` interprets the elements immediately following a processing instruction to construct an array.

## Special characters

When building XML elements, the ESB handles the '\$' character and digits in a special manner when they appear as the first character of a JSON key. Following are examples of two such occurrences. Note the addition of the `_JsonReader_PS_` and `_JsonReader_PD_` prefixes in place of the '\$' and digit characters, respectively.

**JSON:**

```
{ "$key":1234 }
```

**XML:**

```
<jsonObject>
<_JsonReader_PS_key>1234</_JsonReader_PS_key>
</jsonObject>
```

**JSON:**

```
{ "32X32": "image_32x32.png" }
```

**XML:**

```
<jsonObject>
<_JsonReader_PD_32X32>image_32x32.png</_JsonReader_PD_32X32>
</jsonObject>
```

## Converting spaces

Although you can have spaces in JSON elements, [you cannot have them when converted to XML](#). Therefore, you can handle spaces when converting JSON message payloads to XML, by adding the following property to the <ESB\_HOME>/repository/conf/synapse.properties file: synapse.commons.json.buildValidNCNames

For example, consider the following JSON message:

```
{
 "abc def" : "this is a sample value"
}
```

The output converted to XML is as follows:

```
<abc_jsonreader_32_def>this is a sample value</abc_jsonreader_32_def>
```

The value 32 represents the standard char value of the space.

This works other way around as well. When you need to convert XML to JSON, with a JSON element that needs to have a space within it. Then, you use "<\_JsonReader\_32\_" in the XML element, to get the space in the JSON output. For example, if you consider the following XML payload;

```
<abc_jsonreader_32_def>this is a sample value</abc_jsonreader_32_def>
```

The JSON output will be as follows:

```
{
 "abc def" : "this is a sample value"
}
```

## Converting a payload between XML and JSON

To convert an XML payload to JSON, set the `messageType` property to `application/json` in the `axis2` scope before sending message to an endpoint. Similarly, to convert a JSON payload to XML, set the `messageType` property to `application/xml` or `text/xml`. For example:

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="tojson"
 transports="https,http"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
<target>
 <inSequence>
 <property name="messageType" value="application/json" scope="axis2"/>
 <respond/>
 </inSequence>
</target>
<description/>
</proxy>

```

Use the following command to invoke this proxy service:

```

curl -v -X POST -H "Content-Type:application/xml" -d@request1.xml
"http://localhost:8280/services/tojson"

```

If the request payload is as follows:

```

<coordinates>
 <location>
 <name>Bermuda Triangle</name>
 <n>25.0000</n>
 <w>71.0000</w>
 </location>
 <location>
 <name>Eiffel Tower</name>
 <n>48.8582</n>
 <e>2.2945</e>
 </location>
</coordinates>

```

The response payload will look like this:

```
{
 "coordinates": {
 "location": [
 {
 "name": "Bermuda Triangle",
 "n": 25.0000,
 "w": 71.0000
 },
 {
 "name": "Eiffel Tower",
 "n": 48.8582,
 "e": 2.2945
 }
]
 }
}
```

Note that we have used the [Property mediator](#) to mark the outgoing payload to be formatted as JSON:

```
<property name="messageType" value="application/json" scope="axis2" />
```

JSON requests cannot be converted to XML if it contains invalid XML characters.

If you need to convert complex XML responses (e.g., XML with with `xsi:type` values), you will need to set the message type using the [Property mediator](#) as follows:

```
<property name="messageType" value="application/json/badgerfish" scope="axis2"
type="STRING" />
```

You will also need to ensure you register the following message builder and formatter as specified in [Message Builders and Formatters](#).

```
<messageBuilder contentType="text/javascript"
 class="org.apache.axis2.json.JSONBadgerfishOMBuilder"/>

<messageFormatter contentType="text/javascript"
 class="org.apache.axis2.json.JSONBadgerfishMessageFormatter"/>
```

## Accessing content from JSON payloads

There are two ways to access the content of a JSON payload within the ESB.

- [JSONPath expressions](#) (with `json-eval()` method)
- [XPath expressions](#)

JSONPath allows you to access fields of JSON payloads with faster results and less processing overhead. Although it is possible to evaluate XPath expressions on JSON payloads by assuming the XML representation of the JSON payload, we recommend that you use JSONPath to query JSON payloads. It is also possible to evaluate both JSONPath and XPath expressions on a payload (XML/JSON) at the same time.

You can use JSON path expressions with following mediators:

Mediator	Usage
Log	<p>As a log property:</p> <pre>&lt;log&gt;     &lt;property name="location"               expression="json-eval(\$.coordinates.location[0].name)"/&gt; &lt;/log&gt;</pre>
Property	<p>As a standalone property:</p> <pre>&lt;property name="location"           expression="json-eval(\$.coordinates.location[0].name)"/&gt;</pre>
PayloadFactory	<p>As the payload arguments:</p> <pre>&lt;payloadFactory media-type="json"&gt;     &lt;format&gt;{ "RESPONSE": "\$1" }&lt;/format&gt;     &lt;args&gt;         &lt;arg evaluator="json"             expression=".coordinates.location[0].name" /&gt;     &lt;/args&gt; &lt;/payloadFactory&gt;</pre>
<p><b>IMPORTANT:</b> You MUST omit the <code>json-eval()</code> method within the payload arguments to evaluate JSON paths within the PayloadFactory mediator. Instead, you MUST select the correct expression evaluator (<code>xml</code> or <code>json</code>) for a given argument.</p>	
Switch	<p>As the switch source:</p> <pre>&lt;switch source="json-eval(\$.coordinates.location[0].name)"&gt;</pre>
Filter	<p>As the filter source:</p> <pre>&lt;filter source="json-eval(\$.coordinates.location[0].name)"         regex="Eiffel.*"&gt;</pre>

### JSON path syntax

Suppose we have the following payload:

```
{
 "id": 12345,
 "id_str": "12345",
 "array": [1, 2, [[], [{"inner_id": 6789}]]],
 "name": null,
 "object": {},
 "$schema_location": "unknown",
 "12X12": "image12x12.png"
}
```

The following table summarizes sample JSONPath expressions and their outputs:

Expression	Result
\$. .	{ "id":12345, "id_str":"12345", "array": [1, 2, [ [], [ {"inner_id":6789} ] ] ], "name":null, "object":{}, "\$schema_location":"unknown", "12X12":"image12x12.png" }
\$.id	12345
\$.name	null
\$.object	{ }
\$.['\$schema_location']	unknown
\$.12X12	image12x12.png
\$.array	[1, 2, [ [], [ {"inner_id":6789} ] ]]
\$.array[2][1][0].inner_id	6789

You can learn more about JSONPath syntax [here](#).

### Logging JSON payloads

To log JSON payloads as JSON, use the [Log mediator](#) as shown below. The `json-eval()` method returns the `java.lang.String` representation of the existing JSON payload.

```
<log>
 <property name="JSON-Payload" expression="json-eval($.)" />
</log>
```

To log JSON payloads as XML, use the Log mediator as shown below:

```
<log level="full"/>
```

For more information on logging, see [Troubleshooting, debugging, and logging](#) below.

## Constructing and transforming JSON payloads

To construct and transform JSON payloads, you can use the PayloadFactory mediator or Script mediator as described in the rest of this section.

### **PayloadFactory mediator**

The [PayloadFactory mediator](#) provides the simplest way to work with JSON payloads. Suppose we have a service that returns the following response for a search query:

```
{
 "geometry": {
 "location": {
 "lat": -33.867260,
 "lng": 151.1958130
 }
 },
 "icon": "bar-71.png",
 "id": "7eaf7",
 "name": "Biaggio Cafe",
 "opening_hours": {
 "open_now": true
 },
 "photos": [
 {
 "height": 600,
 "html_attributions": [
],
 "photo_reference": "CoQBegAAAI",
 "width": 900
 }
],
 "price_level": 1,
 "reference": "CnRqAAAAtz",
 "types": [
 "bar",
 "restaurant",
 "food",
 "establishment"
],
 "vicinity": "48 Pirrama Road, Pyrmont"
}
```

We can create a proxy service that consumes the above response and creates a new response containing the location name and tags associated with the location based on several fields from the above response.

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="singleresponse"
 transports="https,http"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
 <target>
 <outSequence>
 <payloadFactory media-type="json">
 <format>{
 "location_response" : {
 "name" : "$1",
 "tags" : $2
 }
 </format>
 <args>
 <arg evaluator="json" expression=".name"/>
 <arg evaluator="json" expression=".types"/>
 </args>
 </payloadFactory>
 <send/>
 </outSequence>
 <endpoint>
 <address uri="http://localhost:8280/location"/>
 </endpoint>
 </target>
 <description/>
</proxy>

```

Use the following command to invoke this service:

```
curl -v -X GET "http://localhost:8280/services/singleresponse"
```

The response payload would look like this:

```
{
 "location_response": {
 "name": "Biaggio Cafe",
 "tags": ["bar", "restaurant", "food", "establishment"]
 }
}
```

Note the following aspects of the proxy service configuration:

- We use the `payloadFactory` mediator to construct the new JSON payload.
- The `media-type` attribute is set to `json`.
- Because JSONPath expressions are used in arguments, the `json` evaluators are specified.

### Configuring the payload format

The `<format>` section of the proxy service configuration defines the format of the response. Notice that in the example above, the name and tags field values are enclosed by double quotes ("), which creates a string value in the final response. If you do not use quotes, the value that gets assigned uses the real type evaluated by the

expression (boolean, number, object, array, or null).

It is also possible to instruct the PayloadFactory mediator to load a payload format definition from the registry. This approach is particularly useful when using large/complex payload formats in the definitions. To load a format from the registry, click **Pick From Registry** instead of **Define inline** when defining the PayloadFactory mediator.

For example, suppose we have saved the following text content in the registry under the location `conf:/repository/esb/transform`. (The resource name is “transform”.)

```
{
 "location_response" : {
 "name" : "$1",
 "tags" : $2
 }
}
```

We can now modify the definition of the PayloadFactory mediator to use this format text saved as a registry resource as the payload format. The new configuration would look as follows (note that the `<format>` element now uses the key attribute to point to the registry resource key):

```
<payloadFactory media-type="json">
 <format key="conf:/repository/esb/transform"/>
 ...
</payloadFactory>
```

When saving format text for the PayloadFactory mediator as a registry resource, be sure to save it as text content with the “text/plain” media type.

### **Script mediator**

The [Script mediator](#) in JavaScript is useful when you need to create payloads that have recurring structures such as arrays of objects. The Script mediator defines the following important methods that can be used to manipulate payloads in many different ways:

- `getPayloadJSON`
- `setPayloadJSON`
- `getPayloadXML`
- `setPayloadXML`

By combining any of the setters with a getter, we can handle almost any type of content transformation within the ESB. For example, by combining `getPayloadXML` and `setPayloadJSON`, we can easily implement an XML to JSON transformation scenario. In addition, we can perform various operations (such as deleting individual keys, modifying selected values, and inserting new objects) on JSON payloads to transform from one JSON format to another JSON format by using the `getPayloadJSON` and `setPayloadJSON` methods. Following is an example of a JSON to JSON transformation performed by the Script mediator.

Suppose a second service returns the following response:

```
{
 "results" : [
 {
 "geometry" : {
 "location" : {
 "lat" : -33.867260,
 "lng" : 151.1958130
 }
 },
 "icon" : "bar-71.png",
 "id" : "7eaf7",
 "name" : "Biaggio Cafe",
 "opening_hours" : {
 "open_now" : true
 },
 "photos" : [
 {
 "height" : 600,
 "html_attributions" : [],
 "photo_reference" : "CoQBegAAAI",
 "width" : 900
 }
],
 "price_level" : 1,
 "reference" : "CnRqAAAAAtz",
 "types" : ["bar", "restaurant", "food", "establishment"],
 "vicinity" : "48 Pirrama Road, Pyrmont"
 },
 {
 "geometry" : {
 "location" : {
 "lat" : -33.8668040,
 "lng" : 151.1955790
 }
 },
 "icon" : "generic_business-71.png",
 "id" : "3ef98",
 "name" : "Doltone House",
 "photos" : [
 {
 "height" : 600,
 "html_attributions" : [],
 "photo_reference" : "CqQBmgAAAL",
 "width" : 900
 }
],
 "reference" : "CnRrAAAAAV",
 "types" : ["food", "establishment"],
 "vicinity" : "48 Pirrama Road, Pyrmont"
 }
],
 "status" : "OK"
}
```

The following proxy service shows how we can transform the above response using JavaScript with the Script mediator.

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="locations"
 transports="https,http"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
 <target>
 <outSequence>
 <script language="js"
 key="conf:/repository/esb/transform.js"
 function="transform"/>
 <send/>
 </outSequence>
 <endpoint>
 <address uri="http://localhost:8280/locations"/>
 </endpoint>
 </target>
 <description/>
 </proxy>

```

The registry resource `transform.js` contains the JavaScript function that performs the transformation:

```

function transform(mc) {
 payload = mc.getPayloadJSON();
 results = payload.results;
 var response = new Array();
 for (i = 0; i < results.length; ++i) {
 location_object = results[i];
 l = new Object();
 l.name = location_object.name;
 l.tags = location_object.types;
 l.id = "ID:" + (location_object.id);
 response[i] = l;
 }
 mc.setPayloadJSON(response);
}

```

`mc.getPayloadJSON()` returns the current JSON payload as a JavaScript object. This object can be manipulated as a normal JavaScript variable within a script as shown in the above JavaScript code. The `mc.setPayloadJSON()` method can be used to replace the existing payload with a new payload. In the above script, we build a new array object by using the fields of the incoming JSON payload and set that array object as the new payload (see the response payload returned by the final proxy service below.)

Use the following command to invoke the proxy service:

```
curl -v -X GET "http://ggrky:8280/services/locations"
```

The response payload would look like this:

```
[
 {
 "id": "ID:7eaf7",
 "tags": ["bar", "restaurant", "food", "establishment"],
 "name": "Biaggio Cafe"
 },
 {
 "id": "ID:3ef98",
 "tags": ["food", "establishment"],
 "name": "Doltone House"
 }
]
```

If you want to get the response in XML instead of JSON, you would modify the out sequence by adding the Property mediator as follows:

```
<outSequence>
 <script language="js"
 key="/repository/esb/transform.js"
 function="transform"/>
 <property name="messageType" value="application/xml" scope="axis2"/>
 <send/>
</outSequence>
```

The response will then look like this:

```
<jsonArray>
 <jsonElement>
 <id>ID:7eaf7</id>
 <tags>bar</tags>
 <tags>restaurant</tags>
 <tags>food</tags>
 <tags>establishment</tags>
 <name>Biaggio Cafe</name>
 </jsonElement>
 <jsonElement>
 <id>ID:3ef98</id>
 <tags>food</tags>
 <tags>establishment</tags>
 <name>Doltone House</name>
 </jsonElement>
</jsonArray>
```

If you are not getting the results you want when the Script mediator converts the JSON payload directly into XML, you can build the XML payload iteratively with the Script mediator as shown in the following script.

```

function transformXML(mc) {
 payload = mc.getPayloadJSON();
 results = payload.results;
 var response = <locations/>;
 for (i = 0; i < results.length; ++i) {
 var elem = results[i];
 response.locations += <location>
 <id>{elem.id}</id>
 <name>{elem.name}</name>
 <tags>{elem.types}</tags>
 </location>
 }
 mc.setPayloadXML(response);
}

```

The response would now look like this:

```

<locations>
 <location>
 <id>7eaf7</id>
 <name>Biaggio Cafe</name>
 <tags>bar,restaurant,food,establishment</tags>
 </location>
 <location>
 <id>3ef98</id>
 <name>Doltone House</name>
 <tags>food,establishment</tags>
 </location>
</locations>

```

Finally, let's look at how you can perform delete, modify, and add field operations on JSON payloads with the Script mediator in JavaScript. Let's send the JSON message returned by the `locations` proxy service as the request for the following proxy service, `transformjson`:

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="transformjson"
 transports="https,http"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
 <target>
 <inSequence>
 <script language="js"><![CDATA[
 payload = mc.getPayloadJSON();
 for (i = 0; i < payload.length; ++i) {
 payload[i].id_str = payload[i].id;
 delete payload[i].id;
 payload[i].tags[payload[i].tags.length] = "pub";
 }
 mc.setPayloadJSON(payload);
]]></script>
 <log>
 <property name="JSON-Payload" expression="json-eval($.)"/>
 </log>
 <respond/>
 </inSequence>
 </target>
 <description/>
 </proxy>

```

The proxy service will convert the request into the following format:

```
[
 {
 "name": "Biaggio Cafe",
 "tags": ["bar", "restaurant", "food", "establishment", "pub"],
 "id_str": "ID:7eaf7"
 },
 {
 "name": "Doltone House",
 "tags": ["food", "establishment", "pub"],
 "id_str": "ID:3ef98"
 }
]
```

Note that the transformation (line 9 through 17) has added a new field `id_str` and removed the old field `id` from the request, and it has added a new tag `pub` to the existing tags list of the payload.

For additional examples that demonstrate different ways to manipulate JSON payloads within the ESB mediation flow, see the following samples:

- [Sample 440: Converting JSON to XML Using XSLT](#)
- [Sample 441: Converting JSON to XML Using JavaScript](#)

#### XML to JSON transformation parameters

You can use XML to JSON transformation parameters when you need to transform XML formatted data into the JSON format.

Following are the XML to JSON transformation parameters and their descriptions:

Parameter	Description	Default Value
synapse.commons.json.preserve.namespace	Preserves the namespace declarations in the JSON output in XML to JSON transformations.	false
synapse.commons.json.buildValidNCNames	Builds valid XML NCNames when building XML element names in XML to JSON transformations.	false
synapse.commons.json.output.autoPrimitive	Allows primitive types in the JSON output in XML to JSON transformations.	true
synapse.commons.json.output.namespaceSepChar	The namespace prefix separation character for the JSON output in XML to JSON transformations.	The default separation character is -
synapse.commons.json.output.enableNSDeclarations	Adds XML namespace declarations in the JSON output in XML to JSON transformations.	false
synapse.commons.json.output.disableAutoPrimitive.regex	Disables auto primitive conversion in XML to JSON transformations.	null
synapse.commons.json.output.jsonoutAutoArray	Sets the JSON output to an array element in XML to JSON transformations.	true
synapse.commons.json.output.jsonoutMultiplePI	Sets the JSON output to an xml multiple processing instruction in XML to JSON transformations.	true
synapse.commons.json.output.xmloutAutoArray	Sets the XML output to an array element in XML to JSON transformations.	true
synapse.commons.json.output.xmloutMultiplePI	Sets the XML output to an xml multiple processing instruction in XML to JSON transformations.	false

<code>synapse.commons.json.output.emptyXmlElemToEmptyStr</code>	Sets an empty element to an empty JSON string in XML to JSON transformations.	true
-----------------------------------------------------------------	-------------------------------------------------------------------------------	------

### Validating JSON messages

You can use the [Validate mediator](#) to validate JSON messages against a specified JSON schema as described in the rest of this section.

#### **Validate mediator**

The parameters available in this section are as follows.

Parameter Name	Description
<b>Schema keys defined for Validate Mediator</b>	This section is used to specify the key to access the main schema based on which validation is carried out, as well as to specify the JSON, which needs to be validated.
<b>Source</b>	The JSONPath expression to extract the JSON that needs to be validated. E.g: <code>json-e val(\$.msg)</code>

Following example use the below sample schema `StockQuoteSchema.json` file. Add this sample schema file (i.e. `StockQuoteSchema.json`) to the following Registry path: `conf:/schema/StockQuoteSchema.json`. For instructions on adding the schema file to the Registry path, see [Adding a Resource](#).

When adding this sample schema file to the Registry, specify the **Media Type** as application/json.

```
{
 "$schema": "http://json-schema.org/draft-04/schema#",
 "type": "object",
 "properties": {
 "getQuote": {
 "type": "object",
 "properties": {
 "request": {
 "type": "object",
 "properties": {
 "symbol": {
 "type": "string"
 }
 },
 "required": [
 "symbol"
]
 }
 },
 "required": [
 "request"
]
 }
 },
 "required": [
 "getQuote"
]
}
```

In this example, the required schema for validating messages going through the Validate mediator is given as a registry key (i.e. schema\StockQuoteSchema.json). You do not have any source attributes specified. Therefore, the schema will be used to validate the complete JSON body. The mediation logic to follow if the validation fails is defined within the on-fail element. In this example, the [PayloadFactory mediator](#) creates a fault to be sent back to the party, which sends the message.

```
<validate>
 <schema key="conf:/schema/StockQuoteSchema.json"/>
 <on-fail>
 <payloadFactory media-type="json">
 <format>{ "Error":$1}</format>
 <args>
 <arg evaluator="xml" expression="$ctx:ERROR_MESSAGE"/>
 </args>
 </payloadFactory>
 <property name="HTTP_SC" value="500" scope="axis2"/>
 <respond/>
 </on-fail>
</validate>
```

An example for a valid JSON payload request is given below.

```
{
 "getQuote": {
 "request": {
 "symbol": "WSO2"
 }
 }
}
```

## Troubleshooting, debugging, and logging

To assist with troubleshooting, you can enable debug logging at several stages of the mediation of a JSON payload within the ESB by adding one or more of the following loggers to the <ESB\_HOME>/repository/conf/log4j.properties file and restarting the ESB.

Be sure to turn off these loggers when running the ESB in a production environment, as logging every message will significantly reduce performance.

Following are the available loggers:

Message builders and formatters

- log4j.logger.org.apache.synapse.commons.json.JsonStreamBuilder=DEBUG
- log4j.logger.org.apache.synapse.commons.json.JsonStreamFormatter=DEBUG
- log4j.logger.org.apache.synapse.commons.json.JsonBuilder=DEBUG
- log4j.logger.org.apache.synapse.commons.json.JsonFormatter=DEBUG

JSON utility class

log4j.logger.org.apache.synapse.commons.json.JsonUtil=DEBUG

PayloadFactory mediator

log4j.logger.org.apache.synapse.mediators.transform.PayloadFactoryMediator=DEBUG

JSONPath evaluator

log4j.logger.org.apache.synapse.util.xpath.SynapseJsonPath=DEBUG

## REST Support

You can send and receive RESTful messages through the ESB using a proxy service or API. For complete information, see [Using REST](#).

## WebSocket Support

WebSocket is a protocol that provides full-duplex communication channels over a single TCP connection, and can be used by any client or server application.

WSO2 ESB provides WebSocket support via [WSO2 ESB WebSocket Transport](#), [WSO2 ESB WebSocket Inbound Protocol](#) and [WSO2 ESB Secure WebSocket Inbound Protocol](#).

This section describes the following common use cases that can be implemented with WSO2 ESB's WebSocket support:

- Sending a Message from a WebSocket Client to a WebSocket Endpoint
- Sending a Message from a WebSocket Client to an HTTP Endpoint
- Sending a Message from a HTTP Client to a WebSocket Endpoint

### Sending a Message from a WebSocket Client to a WebSocket Endpoint

The following sections walk you through a sample scenario that demonstrates how to send a message from a WebSocket client to a WebSocket endpoint via WSO2 ESB:

- [Introduction](#)
- [Prerequisites](#)
- [Configuring the sample scenario](#)
- [Executing the sample scenario](#)
- [Analyzing the output](#)

### ***Introduction***

If you need to send a message from a WebSocket client to a WebSocket endpoint via WSO2 ESB, you need to establish a persistent Websocket connection from the WebSocket client to WSO2 ESB as well as from WSO2 ESB to the WebSocket back-end.

To demonstrate this scenario, you need to create two dispatching sequences. One for the client to back-end mediation, and another for the back-end to client mediation. Finally you need to configure WSO2 ESB's WebSocket inbound endpoint to use the created sequences and listen on port 9091.

### ***Prerequisites***

- Start the WSO2 ESB server. For information on how to start the ESB server. see [Running the Product](#).
- Download the [netty-example-4.0.30.Final.jar](#) file.

### ***Configuring the sample scenario***

- Create the sequence for client to back-end mediation as follows:

```
<sequence name="dispatchSeq">
 <property name="OUT_ONLY" value="true"/>
 <send>
 <endpoint>
 <address uri="ws://localhost:8082/websocket"/>
 </endpoint>
 </send>
</sequence>
```

- Create the sequence for back-end to client mediation as follows:

```
<sequence name="outDispatchSeq" xmlns="http://ws.apache.org/ns/synapse">
 <log level="full"/>
 <respond/>
</sequence>
```

- Configure WSO2 ESB's WebSocket inbound endpoint as follows to use the created sequences and listen on port 9091:

```

<inboundEndpoint name="test" onError="fault" protocol="ws"
 sequence="dispatchSeq" suspend="false">
 <parameters>
 <parameter name="inbound.ws.port">9091</parameter>
 <parameter name="ws.outflow.dispatch.sequence">outDispatchSeq</parameter>
 <parameter name="ws.client.side.broadcast.level">0</parameter>
 <parameter name="ws.outflow.dispatch.fault.sequence">fault</parameter>
 </parameters>
</inboundEndpoint>

```

### **Executing the sample scenario**

- Execute the following command to start the WebSocket client:

```

java -DclientPort=9091 -cp netty-example-4.0.30.Final.jar:lib/*:.
io.netty.example.http.websocketx.client.WebSocketClient

```

### **Analyzing the output**

If you analyze the log, you will see that a connection from the WebSocket client to WSO2 ESB is established, and the sequences are executed by the WebSocket inbound endpoint.

You will also see that the message sent to the WebSocket server is not transformed, and that the response injected to the out sequence is also not transformed.

### **Sending a Message from a WebSocket Client to an HTTP Endpoint**

The following sections walk you through a sample scenario that demonstrates how to send a message from a WebSocket client to an HTTP endpoint via WSO2 ESB:

- [Introduction](#)
- [Prerequisites](#)
- [Configuring the sample scenario](#)
- [Executing the sample scenario](#)
- [Analyzing the output](#)

#### **Introduction**

If you need to send a message from a WebSocket client to an HTTP endpoint via WSO2 ESB, you need to establish a persistent Websocket connection from the WebSocket client to WSO2 ESB.

To demonstrate this scenario, you need to create two dispatching sequences. One for the client to back-end mediation, and another for the back-end to client mediation. Finally you need to configure WSO2 ESB's WebSocket inbound endpoint to use the created sequences and listen on port 9091.

#### **Prerequisites**

- Start the WSO2 ESB server. For information on how to start the ESB server. see [Running the Product](#).
- Download the [netty-example-4.0.30.Final.jar](#) file.

#### **Configuring the sample scenario**

- Create the sequence for client to back-end mediation as follows:

```

<sequence name="dispatchSeq">
 <switch source="get-property('websocket.source.handshake.present')">
 <case regex="true">
 <drop/>
 </case>
 <default>
 <call>
 <endpoint>
 <address uri="http://www.mocky.io/v2/56f84ee5240000d1127866c8"/>
 </endpoint>
 </call>
 <respond/>
 </default>
 </switch>
</sequence>

```

This sequence calls an HTTP endpoint.

- Create the sequence for back-end to client mediation as follows:

```

<sequence name="outDispatchSeq" xmlns="http://ws.apache.org/ns/synapse">
 <log/>
 <respond/>
</sequence>

```

- Configure WSO2 ESB's WebSocket inbound endpoint as follows to use the created sequences and listen on port 9091:

```

<inboundEndpoint name="test" onError="falut" protocol="ws"
 sequence="dispatchSeq" suspend="false">
 <parameters>
 <parameter name="inbound.ws.port">9091</parameter>
 <parameter name="ws.outflow.dispatch.sequence">outDispatchSeq</parameter>
 <parameter name="ws.client.side.broadcast.level">0</parameter>
 <parameter name="ws.outflow.dispatch.fault.sequence">fault</parameter>
 </parameters>
</inboundEndpoint>

```

### ***Executing the sample scenario***

- Execute the following command to start the WebSocket client:

```

java -DclientPort=9091 -cp netty-example-4.0.30.Final.jar:lib/*:.
io.netty.example.http.websocketx.client.WebSocketClient

```

### ***Analyzing the output***

If you analyze the log, you will see that a connection from the WebSocket client to WSO2 ESB is established. You will also see that the sequences are executed by the WebSocket inbound endpoint.

## Sending a Message from a HTTP Client to a WebSocket Endpoint

The following sections walk you through a sample scenario that demonstrates how to send a message from a HTTP client to a WebSocket endpoint via WSO2 ESB:

- [Introduction](#)
- [Prerequisites](#)
- [Configuring the sample scenario](#)
- [Executing the sample scenario](#)
- [Analyzing the output](#)

### ***Introduction***

If you need to send a message from a HTTP client to a WebSocket endpoint via WSO2 ESB, you need to establish a persistent Websocket connection from WSO2 ESB to the WebSocket back-end.

To demonstrate this scenario, you need to create two dispatching sequences. One for the client to back-end mediation, and another for the back-end to client mediation. Then you need to create a proxy service to call the created sequences.

### ***Prerequisites***

- Start the WSO2 ESB server. For information on how to start the ESB server. see [Running the Product](#).
- Download the [netty-example-4.0.30.Final.jar](#) file.
- Execute the following command to start the WebSocket server on port 8082:

```
java -cp netty-example-4.0.30.Final.jar:lib/*:.
io.netty.example.http.websocketx.server.WebSocketServer
```

### ***Configuring the sample scenario***

- Create the sequence for client to back-end mediation as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<sequence name="dispatchSeq" xmlns="http://ws.apache.org/ns/synapse">
 <in>
 <property name="OUT_ONLY" value="true" />
 <property name="FORCE_SC_ACCEPTED" scope="axis2" type="STRING"
value="true" />
 <property name="websocket.accept.contentType" scope="axis2"
value="application/json" />
 <send>
 <endpoint>
 <address uri="ws://localhost:8082/websocket" />
 </endpoint>
 </send>
 </in>
</sequence>
```

- Create the sequence for the back-end to client mediation as follows:

```
<sequence name="outDispatchSeq" xmlns="http://ws.apache.org/ns/synapse">
 <log level="full" />
</sequence>
```

- Create a proxy service as follows to call the above sequences:

```
<?xml version="1.0" encoding="UTF-8"?>
<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="websocketProxy1"
 transports="http,https"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
 <target inSequence="dispatchSeq" faultSequence="outDispatchSeq" />
 <description/>
</proxy>
```

### **Executing the sample scenario**

Execute the following command to invoke the proxy service:

```
curl -v --request POST -d "<?xml version=\"1.0\" encoding=\"UTF-8\"?><soapenv:Envelope
xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope/\"><soapenv:Body><test>Value</test></soapenv:Body></soapenv:Envelope>" -H Content-Type:"text/xml"
http://localhost:8280/services/websocketProxy1
```

### **Analyzing the output**

If you analyze the log, you will see that a HTTP request is sent to the WebSocket server, and that the WebSocket server injects the response to the out sequence.

## **Integrating with WSO2 BAM, WSO2 DAS and WSO2 CEP**

WSO2 ESB includes a data agent that can capture data events and send them to WSO2 Business Activity Monitor (BAM), WSO2 Data Analytics Server (WSO2 DAS) or WSO2 Complex Event Processor (CEP) using their Thrift API. To enable this integration, you take the following steps:

1. Install and configure WSO2 BAM, WSO2 DAS or WSO2 CEP as described in the [BAM documentation](#), [DAS documentation](#) or [CEP documentation](#). If you are installing BAM, DAS or CEP on the same server as the ESB, be sure to [configure the ports](#) as described below.
2. [Configure a server profile](#) and one or more streams in the ESB Management Console.
3. Add the [BAM mediator](#) to your sequence and connect it to the server profile you created step 2.

When using the data publisher API to publish data in a periodic manner to WSO2 BAM/DAS/CEP, the eviction time and eviction idle time for the connections should be higher than the periodic interval. This is required to re-use the created socket connections from the pool, avoiding closure of it and creation of new connections. The default eviction period is 5.5 seconds (5500 milliseconds). If you are publishing events in a periodic interval as more than 5.5s, you need to tune the `<secureEvictionTimePeriod>` parameter accordingly, in the `<PRODUCT_HOME>/repository/conf/data-bridge/thrift-agent-config.xml` file of the agent in the client side, by increasing this default value.

### **Configuring the BAM/DAS/CEP ports**

If you are running BAM, DAS or CEP on the same server as the ESB, be sure to set the port offset in `<PRODUCT_HOME>/repository/conf/carbon.xml` so there won't be a conflict with the ESB's HTTPS port. For example,

if you set the port offset of BAM to 1, its HTTPS port will be 9444 instead of 9443. You must also increment the Cassandra URL port by the same amount in <BAM/DAS/CEP\_HOME>/repository/conf/datasources/master-datasources.xml. For example, if your port offset is 1, change <url>jdbc:cassandra://localhost:**9160**/EVENT\_KS</url> to <url>jdbc:cassandra://localhost:**9161**/EVENT\_KS</url> in both the WSO2BAM\_CASSANDRA\_DATASOURCE and WSO2BAM\_UTIL\_DATASOURCE data sources. Additionally, when you set up the BAM server profile in the ESB Management Console, increment the authentication port by the same amount as the offset (such as 7712 instead of the default 7711 if the port offset is 1).

## Configuring a Server Profile

Before adding the **BAM mediator** to a sequence, you must configure a server profile that contains the transport and credential data required to connect to the BAM/DAS/CEP thrift server. In each server profile, you also need to configure one or more event streams to identify the data to be extracted from the configuration context of the mediation sequence.

The following section walks you through the steps to configure a BAM server profile.

You can follow the same steps to configure a server profile for any server with a listening thrift port, such as WSO2 Data Analytics Server (WSO2 DAS) or WSO2 Complex Event Processor (WSO2 CEP). You can then use the BAM mediator to connect to that server through the server profile that you configure.

### **To configure a BAM server profile:**

1. On the Configure tab in the ESB Management Console, click **BAM Server Profile**.
2. To edit an existing BAM server profile, click **Edit Profile** for that profile. To add a new profile, click **Add profile**.
3. Specify the following:
  - **Profile Name:** Enter a unique value for this profile.
  - **Server Credential:** Enter the user name and password to log in to the BAM server.
  - **Protocol:** Leave Thrift selected.
  - **Enable Security:** If you require message confidentiality between the ESB server and BAM server, click **Enable Security**.
  - **IP Address:** Enter the host IP address of the BAM server. By default, this is the IP address of the localhost. You can verify the host IP address on the home page of the BAM Management Console.
  - **Receiver Port:** If security was not enabled, specify the Thrift server port. The default is 7611, which you should increment by the same value as the port offset if you configured a port offset for BAM (such as 7612 if the BAM port offset is 1). For more information, see [Integrating with WSO2 BAM, WSO2 DAS and WSO2 CEP](#).
  - **Authentication Port:** Specify a value 100 higher than the receiver port. The default is 7711, which you should increment by the same value as the port offset. For more information, see [Integrating with WSO2 BAM, WSO2 DAS and WSO2 CEP](#).
4. To configure an event stream, specify the following, and then click **Add Stream**:
  - **Name:** Any string with alpha-numeric characters.
  - **Version:** Distinguishes different streams with the same stream name. Default version should be **1.0.0**.
  - **Nickname:** A meaningful nickname that will help you identify the stream.
  - **Description:** A meaningful description of this particular stream defined by the stream name/version pair.
5. Repeat step 4 to add additional streams as needed.
6. Click **Save**.

You can now select this server profile and one of its streams when adding a **BAM Mediator**.

## Working with Event Sinks

Event sinks contain information about transport endpoints provided by other systems. Events can be published to these endpoints. Endpoint transport and authentication URLs and credentials are captured in event sinks.

### **Adding an event sink**

Follow the steps below to add an event sink.

1. Open the ESB Management Console. On the **Configure** tab, click **Event Sinks** to open the **Event Sinks Configuration** page.
2. Click **Add Event Sink** to open the **Add Event Sink** page.

The screenshot shows the 'Add Event Sink' configuration page. At the top, there's a breadcrumb navigation: Home > Configure > Event Sinks > Event Sink Editor. On the right, there's a 'Help' button with a question mark icon. The main title is 'Add Event Sink'. Below it, there are five input fields with labels and values:

- Name\*: TestEvent
- Username\*: Admin
- Password\*: (redacted)
- Receiver URL\*: tcp://10.200.3.218:7611
- Authenticator URL: ssl://10.200.3.218:7711

At the bottom left are two buttons: a blue 'Save' button with a disk icon and a red 'Cancel' button with a cross icon.

Enter values for the following parameters.

Parameter Name	Description
<b>Name</b>	The name of the event sink.
<b>Username</b>	The user name of the data-bridge endpoint.
<b>Password</b>	The password of the data-bridge endpoint.
<b>Receiver URL</b>	The URL of the data-bridge endpoint. For example, <code>tcp://10.200.3.218:7611</code> . Multiple endpoints can be specified for load balancing and multiple receiver modes, by using the correct URL format.
<b>Authenticator URL</b>	The secure data-bridge endpoint URL that should be used for authentication. For example, <code>ssl://10.200.3.218:7711</code> . The URL specified here should match the specified <b>Receiver URL</b> parameter.

3. Click **Save**.

#### **Sample**

Let's look at a sample where WSO2 ESB uses event sinks to publish mediation data to WSO2 Data Analytics Server (WSO2 DAS).

#### **Setting up WSO2 Data Analytics Server**

1. **Download and install WSO2 DAS. For instructions on how to download and install WSO2 DAS, see [Getting Started with WSO2 DAS](#).**

The unzipped WSO2 DAS distribution folder will be referred to as <DAS\_HOME> throughout the documentation.

It is not possible to start multiple WSO2 products with their default configurations simultaneously in the same environment. Since all WSO2 products use the same port in their default configuration, there will be port conflicts. Therefore, to avoid port conflicts, apply a port offset in the `<DAS_HOME>/repository/conf/carbon.xml` file by changing the offset value to 1 as follows:

```
<Ports>
 <!-- Ports offset. This entry will set the value of the ports
defined below to
the define value + Offset.
e.g. Offset=2 and HTTPS port=9443 will set the effective HTTPS
port to 9445
-->
<Offset>1</Offset>
```

2. Open a command prompt and go to the `<DAS_HOME>/bin` directory.
3. Start WSO2 DAS by executing `sh wso2server.sh` (on Linux/OS X) or `wso2server.bat` (on Windows).
4. Log in to the management console of WSO2 DAS server at <https://localhost:9444/carbon/> using admin for the username and password.
5. Create an event stream in DAS by navigating to **Main->Streams** in the management console. Click on **Add Event Stream** and fill in the details as in the following table:

Field	Value
Event Stream Name	stockquote_stream
Event Stream Version	1.0.0

6. Add the Stream Attributes as in the following table:

<b>Meta Data Attributes</b>	
Attribute Name	http_method
Attribute Type	string
Attribute Name	destination
Attribute Type	string
<b>Correlation Attributes</b>	
Attribute Name	date
Attribute Type	string
<b>Payload Data Attributes</b>	
Attribute Name	symbol
Attribute Type	string
Attribute Name	request
Attribute Type	string

## Define New Event Stream

Enter Event Stream Details [switch to source view](#)

Event Stream Name*	stockquote_stream <small>Name of the Event Stream</small>
Event Stream Version*	1.0.0 <small>Version of the event stream (Eg : 1.0.0)</small>
Event Stream Description	<small>Description of the event stream</small>
Event Stream Nick-Name	<small>Nick name of the event stream</small>

**Stream Attributes**

Meta Data Attributes

Attribute Name	Attribute Type	Actions
http_method	string	Delete
destination	string	Delete

Attribute Name :  Attribute Type :

Correlation Data Attributes

Attribute Name	Attribute Type	Actions
date	string	Delete

Attribute Name :  Attribute Type :

Payload Data Attributes

Attribute Name	Attribute Type	Actions
symbol	string	Delete
request	string	Delete

Attribute Name :  Attribute Type :

[Add Event Stream](#) [Next \[Persist Event\]](#)

Click **Next[Persist Event]**.

- On the next page, select the check box **Persist Event Stream** and then select the attributes that need to be persisted by selecting the relevant check boxes. Select all attributes by selecting the check boxes **Persist Attribute** under Meta Data Attributes, Correlation Data Attributes and Payload Data Attributes. See [WSO2 DAS documentation](#) for details on persisting data.

[Home](#)

[Help](#)

## Edit Event Stream

Enter Event Stream Details

Persist Event Stream

Record Store **EVENT\_STORE**

Meta Data Attributes

<input checked="" type="checkbox"/> Persist Attribute	Attribute Name	Attribute Type	Primary Key	Index Column	Score Param	Is a Facet
<input checked="" type="checkbox"/>	http_method	STRING	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	destination	STRING	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Correlation Data Attributes

<input checked="" type="checkbox"/> Persist Attribute	Attribute Name	Attribute Type	Primary Key	Index Column	Score Param	Is a Facet
<input checked="" type="checkbox"/>	date	STRING	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Payload Data Attributes

<input checked="" type="checkbox"/> Persist Attribute	Attribute Name	Attribute Type	Primary Key	Index Column	Score Param	Is a Facet
<input checked="" type="checkbox"/>	symbol	STRING	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	request	STRING	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Arbitrary Data Attributes

No arbitrary data attributes are defined

Attribute Name :  Attribute Type :  Primary Key :  Index Column :  Score Param :  Is a Facet :

[Advanced](#)

[Back](#) [Save Event Stream](#)

**Click Save Event Stream.**

- You need to add an **Event Receiver** in WSO2 DAS to receive events. In WSO2 DAS, navigate to **Main->Receivers** and click on **Add Event Receiver** and fill in the following information:

Field	Value
Event Receiver Name	ESB_RECEIVER
Input Event Adapter Type	Select <b>wso2event</b>

**Click Add Event Receiver.**

#### Create a New Event Receiver

Enter Event Receiver Details

Event Receiver Name\*  ⓘ Enter a unique name to identify Event Receiver

From

Input Event Adapter Type\*  ⓘ Select the type of Adapter to receive events

Following url formats are used to receive events  
For load-balancing: use "," to separate values for multiple endpoints  
Eg: {tcp://<hostname>:<port>},{tcp://<hostname>:<port>},...}

For failover: use "!" to separate multiple endpoints  
Eg: {tcp://<hostname>:<port>}!{tcp://<hostname>:<port>} ...}

For more than one cluster: use "{}" to separate multiple clusters  
Eg: {tcp://<hostname>:<port>}/{tcp://<hostname>:<port>} ...},{tcp://<hostname>:<port>}

Ports available for Thrift protocol - TCP port:7612 or SSL port:7712  
Ports available for Binary protocol - TCP port:9612 or SSL port:9712

Adapter Properties

Is events duplicated in cluster  ⓘ This depends on how events are published to the server, 'true' only if multiple receiver URLs are defined in different receiver groups ({}).

To

Event Stream\*  ⓘ The event stream that will be generated by the received events

Mapping Configuration

Message Format\*  ⓘ Select the input message format

[Advanced](#)

**Add Event Receiver**

#### Setting up WSO2 ESB

- If you have not already done so, see [Getting Started with WSO2 ESB](#) for details on installing and running WSO2 ESB.
- Log in to the management console at <https://localhost:9443/carbon/> using admin for the username and password.
- Create an event sink in WSO2 ESB by navigating to **Configure ->Event Sinks** in the management console. Click on **Add Event Sink**.

The screenshot shows the WSO2 ESB Management Console interface. The left sidebar has tabs for Main, Monitor, Configure, and Tools, with 'Configure' selected. Under 'Configure', there are several sections: Users and Roles, User Stores, Keystores, Cloud Gateway Agent, Features, Logging, Datasources, BAM Server Profile, Server Roles, Event Sinks (which is highlighted with a red box), Multitenancy, Add New Tenant, and View Tenants. The main content area is titled 'Event Sinks Configurations' and contains a table with three columns: Name, Username, and Receiver URL. A red box highlights the 'Add Event Sink' button at the bottom of the table.

Fill in the details as in the following table:

Field	Value
Name	das_event_sink
Username	admin
Password	admin
Receiver URL	tcp://localhost:7612
Authenticator URL	ssl://localhost:7712

In WSO2 DAS the default thrift port and the authentication port are 7611 and 7711 respectively. Due to the port offset you configured the new ports are 7612 and 7712 as defined above.

4. Create a REST API by navigating to **Main -> APIs** in the management console and click on **Add API**. Fill in the details as in the table below:

Field	Value
API Name	StockQuoteAPI
context	/stockquote

Click on **Switch to source view**.

**Add API**

API [Switch to source view](#)

API Name *	StockQuoteAPI
Context *	/stockquote
Hostname	<input type="text"/>
Port	<input type="text"/>

- Root  Add Resource

Save
Cancel

In source view, add the following API configuration where the publish mediator uses the created event sink das\_event\_sink to publish to the event stream named stockquote\_stream that we created in WSO2 DAS:

```

<api xmlns="http://ws.apache.org/ns/synapse" name="StockQuoteAPI"
context="/stockquote">
 <resource methods="GET" uri-template="/view/{symbol}">
 <inSequence>
 <payloadFactory media-type="xml">
 <format>
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>$1</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </format>
 <args>
 <arg evaluator="xml"
expression="get-property('uri.var.symbol')"/>
 </args>
 </payloadFactory>
 <header name="Action" value="urn:getQuote" />
 <publishEvent>
 <eventSink>das_event_sink</eventSink>
 <streamName>stockquote_stream</streamName>
 <streamVersion>1.0.0</streamVersion>
 <attributes>
 <meta>
 <attribute xmlns:ns="http://org.apache.synapse/xsd"
name="http_method" type="STRING" defaultValue="" expression="get-property('axis2', 'HTTP_METHOD')"/>
 <attribute xmlns:ns="http://org.apache.synapse/xsd"

```

```
name="destination" type="STRING" defaultValue="" expression="get-property('To')"
/>
 </meta>
 <correlation>
 <attribute xmlns:ns="http://org.apache.synapse/xsd"
name="date" type="STRING" defaultValue=""
expression="get-property('SYSTEM_DATE')" />
 </correlation>
 <payload>
 <attribute xmlns:m0="http://services.samples"
name="symbol" type="STRING" defaultValue=""
expression="$body/m0:getQuote/m0:request/m0:symbol" />
 <attribute name="request" type="STRING" defaultValue=""
value="getSimpleQuote" />
 </payload>
 <arbitrary/>
 </attributes>
 </publishEvent>
 <property name="messageType" value="text/xml" scope="axis2"/>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" format="soap12"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
```

```

 </outSequence>
 </resource>
</api>

```

5. Deploy the back-end service `SimpleStockQuoteService`. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).
6. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
7. Open a command line terminal and enter the following request to invoke the API you defined above:

```
curl -v http://localhost:8280/stockquote/view/WSO2
```

You will see the response from the `SimpleStockQuoteService`.

## Exploring published data in WSO2 DAS

In WSO2 DAS navigate to **Main->Data Explorer** and select the table `STOCKQUOTE_STREAM` from the drop down list and click on **Search**. You can then view the data published from ESB in the table as follows:

The screenshot shows the WSO2 Data Explorer interface. At the top, there is a search bar with fields for 'Table Name\*' (set to 'STOCKQUOTE\_STREAM'), 'Maximum Result Count' (set to '1000'), and search options ('By Date Range' or 'By Query'). Below the search bar are two buttons: 'Search' (highlighted with a blue border) and 'Reset'. The main area is titled 'Results' and contains a note: 'Note: Total record count for the table is not available.' Below this is a table titled 'STOCKQUOTE\_STREAM' with the following data:

	meta_http_method	meta_destination	correlation_date	symbol	request	_timestamp
	GET	/stockquote/view/WSO2	10/24/16 5:26 PM	WSO2	getSimpleQuote	2016-10-24 17:26:09 GST

At the bottom of the table, there are navigation links: '<< < 1 2 ... 99 100 > >>' and 'Go to page: 1 Row count: 10'. To the right, it says 'Showing 1-10 of 1000'.

## Integrating with Other Technologies

WSO2 ESB can be integrated with many different products and technologies. For more information, read the following sections:

- [AMQP Support](#)
- [ebMS Integration](#)
- [PayPal WS API Integration](#)
- [SAP Integration](#)

### AMQP Support

AMQP is a wire-level messaging protocol that describes the format of the data that is sent across the network. If a system or application can read and write AMQP, it can exchange messages with any other system or application that understands AMQP, regardless of the implementation language.

WSO2 ESB provides the [RabbitMQ AMQP transport](#) as an inbuilt feature, which allows you to send or receive AMQP messages by calling an AMQP broker (RabbitMQ) directly without the need to use different transport mechanisms, such as JMS.

For information on other ways to use AMQP with WSO2 ESB, see this [article](#).

### ebMS Integration

The enterprise business XML Message Services (ebMS) standard provides a uniform way for businesses and governments to exchange messages. It is based on SOAP with Attachments and supports guaranteed delivery

(messages are not lost if a failure occurs) and idempotent delivery (each message is delivered only once).

The Digalink adapter from Yenlo enables ebMS messaging through WSO2 ESB. For more information, see <http://www.yenlo.nl/en/digalink/>.

## PayPal WS API Integration

To learn how to integrate ESB with PayPal WS API, see the following article.

### SAP Integration

**Systems, Applications, and Products (SAP)** for data processing is an industry leading enterprise software solution that is widely used among product and process oriented enterprises for finance, operations, HR and many other aspects of a business. SAP ERP solutions provide reliable and efficient platforms to build and integrate enterprise or business-wide data and information systems with ease.

WSO2 ESB leverages the best of both worlds by providing the integration layer so that an existing SAP R/3 based solutions of an enterprise can be integrated with other data/business oriented systems so that you can mix-and-match requirements with minimal effort. As a result, enterprises can keep parts of their systems independent of SAP and extensible for many other systems, solutions and middleware.

The WSO2 SAP adapter is shipped with WSO2 ESB and is implemented as a transport for WSO2 ESB. This is provided in the <ESB\_HOME>/repository/components/plugins directory as `org.wso2.carbon.transports.sap-VERSION.jar` (e.g. `org.wso2.carbon.transports.sap_1.0.0.jar`).

The WSO2 SAP adapter has full IDoc and experimental BAPI support. It uses the SAP JCO library as the underlying framework to communicate with SAP. This section describes how to set up WSO2 ESB in a SAP environment, how to install the SAP JCo middleware library, SAP Intermediate Document (IDoc) and Business Application Programming Interface (BAPI) adapters.

---

[Installing WSO2 SAP Adapter](#) | [Setting up the Client Configuration File](#) | | [Setting up the Server Configuration File](#) | [Configuring WSO2 SAP Adapter](#) | [Additional Configuration Parameter](#) | [Troubleshooting](#)

---

### Installing WSO2 SAP Adapter

Follow the instructions below to install and set up the ESB SAP adapter.

1. Download and install WSO2 ESB by following the instruction in [Installation Guide](#).
2. Download the `sapidoc3.jar` and `sapjco3.jar` middleware libraries from the SAP support portal and copy those libraries to the <ESB\_HOME>/repository/components/lib directory.

### Note

You need to have SAP login credentials to access the SAP support portal.

3. Download the native SAP JCo library and copy it to the system path. You need to select the system path applicable to your operating system as described below.

Linux 32-bit	Copy the Linux native SAP jcolibrary <code>libsapjco3.so</code> to <JDK_HOME>/jre/lib/i386/server.
Linux 64-bit	Copy the Linux native SAP jcolibrary <code>libsapjco3.so</code> to <JDK_HOME>/jre/lib/amd64.
Windows	Copy the Windows native SAP jcolibrary <code>sapjco3.dll</code> to <WINDOWS_HOME>/system32.

4. Copy the following SAP endpoint properties files to the <ESB\_HOME>/repository/conf/sap directory. You need to have two properties files, one at the server-end and the other at the client-end to communicate

with an external SAP endpoint using IDoc or BAPI.

This directory does not exist by default. A SAP system administrator has to create the directory and provide access rights so that you can read the properties files saved in the directory.

- **\*.dest** : This is where we store SAP endpoint parameters when WSO2 ESB is configured as a client to an external SAP endpoint.
- **\*.server** : This is where we store SAP endpoint parameters when WSO2 ESB is configured as a server to an external SAP endpoint.

For details on creating the properties files and defining the relevant properties, see [Setting up the Client Configuration File](#) and [Setting up the Server Configuration File](#).

5. Start the ESB using the `-Djava.library.path` switch to specify the location of your SAP jco library. For example `./wso2server.sh`

`-Djava.library.path=/usr/lib/jvm/jre1.7.0/lib/i386/server/`

#### ***Setting up the Client Configuration File***

To setup WSO2 ESB as a client to a SAP system you need to create the `*.dest` properties file and define the relevant properties. The following table lists the properties and the description of each property that should be specified in the `*.dest` properties file.

Property	Description
<code>jco.client.client</code>	Client logon
<code>jco.client.user</code>	User logon
<code>jco.client.alias_user</code>	Alias user name
<code>jco.client.passwd</code>	Logon password
<code>jco.client.lang</code>	Logon language
<code>jco.client.sysnr</code>	R/3 system number
<code>jco.client.ashost</code>	R/3 application server
<code>jco.client.mshost</code>	R/3 message server
<code>jco.client.gwhost</code>	Gateway host
<code>jco.client.gwserv</code>	Gateway service
<code>jco.client.r3name</code>	R/3 name
<code>jco.client.group</code>	Group of application servers
<code>jco.client.tpname</code>	Program ID of external server program
<code>jco.client.tphost</code>	Host of external server program
<code>jco.client.type</code>	Type of remote host (3=R/3, E=External)
<code>jco.client.codepage</code>	Initial code page for logon

<code>jco.client.use_sapgui</code>	Use remote SAP graphical user interface
<code>jco.client.mysapss02</code>	Use the specified SAP cookie version 2 as the logon ticket
<code>jco.client.grt_data</code>	Additional data for GUI
<code>jco.client.use_guihost</code>	Host to which the remote GUI is redirected
<code>jco.client.use_guiserv</code>	Service to which the remote GUI is redirected
<code>jco.client.use_guiprogid</code>	Progid of the server that starts the remote GUI
<code>jco.client.snc_partnername</code>	SNC partner name (for example, CN=B20, O=SAP-AG, C=DE) snc_mode
<code>jco.client.snc_mode</code>	SNC mode (0 or 1)
<code>jco.client.snc_qop</code>	SNC level of security (1-9)
<code>jco.client.snc_myname</code>	SNC name; overrides default SNC partner
<code>jco.client.snc_lib</code>	Path to the library
<code>jco.client.Dest</code>	R/2 destination
<code>jco.client.saplogon_id</code>	SAPLOGON string on 32-bit Windows
<code>jco.client.extiddata</code>	Data for external application (PAS)
<code>jco.client.extidtype</code>	Type of external authentication (PAS)
<code>jco.client.x509cert</code>	Use the specified X509-certificate as the logon ticket
<code>jco.client.msserv</code>	R/3 port number of message server
<code>jco.client.profile_name</code>	Profile name used for shared memory communication
<code>jco.client.idle_timeout</code>	Idle timeout for the connection
<code>jco.client.ice Ignore</code>	RFC library character conversion errors (1 or 0)
<code>jco.client.logon</code>	Enable or disable logon check at open time (1 or 0)
<code>jco.client.trace</code>	Enable or disable RFC trace (1 or 0)
<code>jco.client.abap_debug</code>	Enable ABAP debugging (1 or 0)
<code>jco.client.getss02</code>	Get or do not get a SSO ticket after logon (1 or 0)
<code>jco.client.toupper</code>	Enable or disable uppercase character conversions for logon

## Note

You can obtain the values for these properties from your SAP system administrator.

The \*.dest properties file should be named <SAP-GHOST>.dest. For example, if the name of your SAP

gateway is SAPSYS, the name of the file should be SAPSYS.dest .

Following is a sample configuration for the \*.dest properties file:

```
jco.client.client=800
jco.client.user=wso2_user
jco.client.passwd=wso2pass14
jco.client.lang=en
jco.client.ashost=/H/217.116.29.154/S/3299/H/10.100.5.120/S/3200
jco.client.gwserv=3300
jco.client.sysnr=00
jco.client.idle_timeout=300
jco.client.logon=0
jco.client.msserv=3600
jco.client.trace=0
jco.client.getsso2=0
jco.client.r3name=CPT
```

### **Setting up the Server Configuration File**

To setup WSO2 ESB as an IDoc server you need to create the \*.server properties file and define the relevant properties. The following table lists the properties and the description of each property that should be specified in the \*.server properties file.

Property	Description
<b>jco.server.gwhost</b>	Gateway host
<b>jco.server.gwserv</b>	Gateway service
<b>jco.server.progid</b>	Program ID of the server
<b>jco.server.trace</b>	You can enable or disable the RFC trace
<b>jco.server.repository_destination</b>	Name of the .dest file. For example, if the .dest file is SAPSYS01.dest , set this to SAPSYS01 .
<b>jco.server.params</b>	Arbitrary parameters for RFC library
<b>jco.server.snc_myname</b>	SNC name
<b>jco.server.snc_qop</b>	SNC level of security (1-9)
<b>jco.server.snc_lib</b>	Path to the SNC library
<b>jco.server.profile_name</b>	Name of the profile file used during start-up
<b>jco.server.unicode</b>	Determines whether or not you connect in unicodemode (1=true, 0=false)
<b>jco.server.max_startup_delay</b>	Maximum server start-up delay time in seconds
<b>jco.server.connection_count</b>	Number of SAP to ESB connections

<b>jco.server.name</b>	Name of the server configuration. This needs to be the same name provided in the SAP configuration.
------------------------	-----------------------------------------------------------------------------------------------------

**Note**

You can obtain the values for these properties from your SAP system administrator.

This file should be named <SAP-GHOST>.server. For example, if the name of your SAP gateway is SAPSYS, the name of the file should be SAPSYS.server.

Following is a sample configuration for the \*.server properties file:

```
jco.server.gwhost=/H/217.116.29.154/S/3299/H/10.100.5.120/S/3200
jco.server.gwserv=3300
jco.server.progid=IGS.CPT
jco.server.repository_destination=IGS.CPT
jco.server.name=IGS.CPT
jco.server.unicode=1
```

**Configuring WSO2 SAP Adapter**

Go to the required tab for detailed steps based on how you need to configure WSO2 SAP Adapter.

[Configure with IDoc adapter](#)[Configure with BAPI adapter](#)

WSO2 SAP adapter can be used with IDoc, which is a synchronous interface used when exchanging data with the SAP system. WSO2 ESB can be configured for [Sending IDocs](#) or [Receiving IDocs](#) when using the SAP adapter.

**Sending IDocs**

Follow the instructions below to configure WSO2 ESB as an IDoc client using the SAP adapter.

1. Uncomment the following line in <ESB\_HOME>/repository/conf/axis2/axis2.xml file to enable the IDoc transport sender on axis2 core.

```
<transportSender name="idoc"
class="org.wso2.carbon.transports.sap.SAPTransportSender" />
```

2. Create IDocSender proxy service with the following configuration:

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="IDocSender"
 transports="http"
 startOnLoad="true"
 trace="enable"
 statistics="enable">
 <target>
 <inSequence>
 <log level="full"/>
 <send>
 <endpoint name="sapidocendpoint">
 <address uri="idoc:/SAPSYS"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence/>
 </target>
 <parameter name="serviceType">proxy</parameter>
 <description/>
 </proxy>

```

- The SAP endpoint client properties file `SAPSYS.dest` should be in `<ESB_HOME>/repository/conf/sap` folder.
- Additional axis2 level sender parameters that can be defined in the axis2 core are listed in [SA P Transport Sender Parameters](#).

You can now send IDocs using the configured WSO2 SAP adapter.

### **Receiving IDocs**

Follow the instructions below to configure WSO2 ESB as an IDoc server using the SAP adapter.

1. Uncomment the following line in `<ESB_HOME>/repository/conf/axis2/axis2.xml` file to enable IDoc transport receiver in axis2 core.

```

<transportReceiver name="idoc"
 class="org.wso2.carbon.transports.sap.SAPTransportListener"/>

```

2. Ensure the server configuration file `SAPSYS.server` is available in `<ESB_HOME>/repository/conf/sap` folder.
3. Start the ESB using the `-Djava.library.path` switch to specify the location of your SAP jco library.  
For example `sh wso2server.sh -Djava.library.path=/usr/lib/jvm/jre1.7.0/lib/i386/server/`
4. Create the IDocReceiver proxy service with the following configuration:

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="IDocReceiver"
 transports="idoc"
 statistics="enable"
 trace="enable"
 startOnLoad="true">
 <target>
 <inSequence>
 <log level="full"/>
 <drop/>
 </inSequence>
 <outSequence>
 <log level="full"/>
 <send/>
 </outSequence>
 </target>
 <parameter name="transport.sap.enableTIDHandler">enabled</parameter>
 <parameter name="transport.sap.serverName">SAPSYS</parameter>
 <description/>
 </proxy>

```

- The SAP endpoint server properties file `SAPSYS.server` should be in `<ESB_HOME>/repository/conf/sapfolder`.
- Additional proxy level listener parameters that can be defined in the proxy configuration are listed in [Proxy Service Listener Parameters](#).

Once the proxy service configuration is saved, WSO2 SAP adapter is now ready to receive IDoc messages. WSO2 SAP adapter so that it can be used with BAPI, which is a synchronous interface used when exchanging data with the SAP system. WSO2 ESB can be configured for [Sending BAPIs](#) or [Receiving BAPIs](#) when using the SAP adapter.

### ***Sending BAPIs***

Follow the instructions below to configure WSO2 ESB as a BAPI client using the SAP adapter.

1. Uncomment the following line in `<ESB_HOME>/repository/conf/axis2/axis2.xml` file to enable the BAPI transport sender in axis2 core.

```

<transportSender name="bapi"
 class="org.wso2.carbon.transports.sap.SAPTransportSender" />

```

2. Start the ESB using the `-Djava.library.path` switch to specify the location of your SAP jco library. For example `sh wso2server.sh -Djava.library.path=/usr/lib/jvm/jre1.7.0/lib/i386/server`
3. Create the BAPISender proxy service with the following configuration:

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="BAPISender"
 transports="bapi"
 startOnLoad="true"
 trace="disable">
 <target>
 <inSequence>
 <send>
 <endpoint name="sap_bapi_endpoint">
 <address uri="bapi:/SAPSYS" />
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <log level="full" />
 <send/>
 </outSequence>
 </target>
 </proxy>

```

- The SAP endpoint client properties file `SAPSYS.dest` should be in `<ESB_HOME>/repository/conf/sap` folder.
- Additional axis2 level sender parameters that can be defined in the axis2 core are listed in [SA P Transport Sender Parameters](#).

## Receiving BAPIs

Follow the instructions below to configure WSO2 ESB as a BAPI server using the SAP adapter.

1. Uncomment the following line in `<ESB_HOME>/repository/conf/axis2/axis2.xml` file to enable the BAPI transport listener in axis2 core.

```

<transportReceiver name="bapi"
 class="org.wso2.carbon.transports.sap.SAPTransportListener" />

```

2. Start the ESB using the `-Djava.library.path` switch to specify the location of your SAP jco library.

For example `sh wso2server.sh`

`-Djava.library.path=/usr/lib/jvm/jre1.7.0/lib/i386/server`

3. Create the `BAPIServer` proxy service with the following configuration:

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="BAPIReceiver"
 transports="bapi"
 statistics="enable"
 trace="enable"
 startOnLoad="true">
 <target>
 <inSequence>
 <log level="full"/>
 <drop/>
 </inSequence>
 <outSequence>
 <log level="full"/>
 <send/>
 </outSequence>
 </target>
 <parameter name="transport.sap.enableTIDHandler">enabled</parameter>
 <parameter name="transport.sap.serverName">SAPSYS</parameter>
 <description/>
 </proxy>

```

- The SAP endpoint server properties file `SAPSYS.server` should be in `<ESB_HOME>/repository/conf/sap` folder.
- Additional proxy level listener parameters that can be defined in the proxy configuration are listed in [Proxy Service Listener Parameters](#).

#### ***Additional Configuration Parameter***

This section describes additional parameters that can be used when configuring WSO2 SAP adapter.

#### ***SAP Transport Sender Parameters***

Following are descriptions of the SAP client properties that can be defined in the message context with axis2-client scope when using WSO2 ESB as a SAP client to send messages. These properties can be added in `<ESB_HOME>/repository/conf/axis2/axis-client.xml` file:

Property	Description
<code>transport.sap.xmlMapper</code>	The key of custom IDOC XML mapper to use. This key should be defined in the <code>transport.sap.customXMLMappers</code> parameter. If no key is specified the default IDoc XML mapper will be used.

<b>transport.sap.xmlParserOptions</b>	The options for the default IDoc XML parser to be used in the default IDoc XML mapper. Multiple options can be combined using the bitwise OR " " operator. The possible parser options are as follows:  PARSE_ACCEPT_ONLY_XMLVERSION_10 3328 PARSE_ACCEPT_ONLY_XMLVERSION_11 2816 PARSE_ACCEPT_ONLY_XMLVERSIONS_10_TO_11 2304 PARSE_IGNORE_INVALID_CHAR_ERRORS 4 PARSE_IGNORE_UNKNOWN_FIELDS 2 PARSE_REFUSE_UNKNOWN_XMLVERSION 256 PARSE_REFUSE_XMLVERSION_10 512 PARSE_REFUSE_XMLVERSION_11 1024 PARSE_WITH_FIELD_VALUE_CHECKING 1 PARSE_WITH_IGNORE_UNKNOWN_FIELDS 2 PARSE_WITHOUT_FIELD_DATATYPE_CHECKING 8
---------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Following is an Axis2 transport sender property that can be defined in <ESB\_HOME>/repository/conf/axis2.xml file:

**transport.sap.customXMLMappers** : The key/value list of custom mappers, where the values are fully qualified class names for custom mappers implementing org.wso2.carbon.transports.sap.IDocXMLMapper.

### Proxy Service Listener Parameters

Following are descriptions of the proxy level listener parameters that can be defined in a proxy configuration when using WSO2 ESB as a SAP server:

Parameter	Description
<b>transport.sap.serverName</b>	The name of the server containing the JCO server configuration.
<b>transport.sap.enableErrorListener</b>	Set this to enable the default error listener. If this is used together with the <code>transport.sap.customErrorListener</code> parameter, the custom error listener will be used.
<b>transport.sap.enableTIDHandler</b>	Set this to enable the transaction handler to handle transactions that are received from a SAP system. Transactional applications must provide a custom implementation using the <code>transport.sap.customTIDHandler</code> parameter.
<b>transport.sap.customTIDHandler</b>	The fully qualified class name for the custom TID handler implementing <code>JCoServerTIDHandler</code> .
<b>transport.sap.connections</b>	The number of registered connections managed by the server instance. The default value is 1 and the maximum value is 100.
<b>transport.sap.customErrorListener</b>	The fully qualified class name for the custom error listener implementing <code>JCoServerErrorListener</code> .
<b>transport.sap.customExceptionListener</b>	The fully qualified class name for the custom exception listener implementing <code>JCoServerExceptionListener</code> .

## Troubleshooting

Given below are general troubleshooting guidelines.

- **How to handle the Server unknown error**

An example of this error message is as follows:

```
[2010-10-25 19:53:00,405] ERROR - DefaultErrorListener Exception occurred on : JCOSEVER01 and connection : null
com.sap.conn.jco.JCoException: (129) JCO_ERROR_SERVER_STARTUP: Server startup failed at Mon Oct 25 19:53:00 IST 2010.
This is caused by either a) erroneous server settings, b) the backend system has been shutdown,
c) network problems. Will try next startup in 1 seconds.
Could not start server: Connect to SAP gateway failed
Connect parameters: TPNAME=JCOSEVER01 GWHOST=cynthia GWSERV=sapgw00
ERROR service 'sapgw00' unknown
TIME Mon Oct 25 19:53:00 2010
RELEASE 720
COMPONENT NI (network interface)
VERSION 40
RC -3
MODULE nixxsl.cpp
LINE 184
DETAIL NiSrvLGetServNo: service name cached as unknown
COUNTER 2
at com.sap.conn.jco.rt.DefaultServer.openConnection(DefaultServer.java:1168)
at com.sap.conn.jco.rt.DefaultServer.openConnection(DefaultServer.java:1057)
at
com.sap.conn.jco.rt.DefaultServer.adjustConnectionCount(DefaultServer.java:1004)
at
com.sap.conn.jco.rt.DefaultServerManager$DispatcherWorker.run(DefaultServerManager.java:
299)
at java.lang.Thread.run(Thread.java:619)
Caused by: com.sap.conn.jco.JCoException: (129) JCO_ERROR_SERVER_STARTUP: Could not start server: Connect to SAP gateway failed
Connect parameters: TPNAME=JCOSEVER01 GWHOST=cynthia GWSERV=sapgw00
ERROR service 'sapgw00' unknown
TIME Mon Oct 25 19:53:00 2010
RELEASE 720
COMPONENT NI (network interface)
VERSION 40
RC -3
MODULE nixxsl.cpp
LINE 184
DETAIL NiSrvLGetServNo: service name cached as unknown
COUNTER 2
at
com.sap.conn.jco.rt.MiddlewareJavaRfc$JavaRfcServer.accept(MiddlewareJavaRfc.java:2135)
at com.sap.conn.jco.rt.ServerConnection.accept(ServerConnection.java:380)
at com.sap.conn.jco.rt.DefaultServer.openConnection(DefaultServer.java:1149)
© 2012 WSO2
```

```
.. 4 more
Caused by: RfcException: [null]
message: Connect to SAP gateway failed
Connect parameters: TPNAME=JCOSERVER01 GWHOST=cynthia GWSERV=sapgw00
ERROR service 'sapgw00' unknown
TIME Mon Oct 25 19:53:00 2010
RELEASE 720
COMPONENT NI (network interface)
VERSION 40
RC -3
MODULE nixxsl.cpp
LINE 184
DETAIL NiSrvLGetServNo: service name cached as unknown
COUNTER 2
Return code: RFC_FAILURE(1)
error group: 102
key: RFC_ERROR_COMMUNICATION
at com.sap.conn.rfc.engine.RfcIoControl.error_end(RfcIoControl.java:255)
at com.sap.conn.rfc.engine.RfcIoControl.ab_rfcallaccept(RfcIoControl.java:43)
at com.sap.conn.rfc.api.RfcApi.RfcAccept(RfcApi.java:41)
at
```

```
com.sap.conn.jco.rt.MiddlewareJavaRfc$JavaRfcServer.accept(MiddlewareJavaRfc.java
:2121)
... 6 more
```

The solution to overcome this is to add your SAP server names to the `/etc/services` file with the relevant ports. For example, the following lines can be added if we consider the example error given above.

```
sapgw00 3300/tcp
sapgw01 3301/tcp
```

- **How to handle connection to message server host failed error**

An example of this error message is as follows:

```
Connection parameters: TYPE=B DEST=SAPSYS01 MSHOST=SAPSYS01 MSSERV=3600
R3NAME=ERD GROUP=PUBLIC PCS=1

ERROR Group PUBLIC not found

TIME Fri Jan 24 15:48:53 2014
```

This indicates that the username (i.e. wso2-user) used in the above configuration is not assigned to the 'public' user group.

The solution to overcome this is to add wso2-user to the user group named public in your SAP system. If such a group does not exist, a user group named 'public' needs to be created and the above user needs to be added to that group.

## Extending the ESB

This section explains how to write custom extensions (artifacts) and custom mediators and mediator implementations. It contains the following topics:

- [Working with Connectors](#)
- [Writing Tasks](#)
- [Writing a Synapse Handler](#)
- [Writing a Custom Message Builder and Formatter](#)
- [Writing a WSO2 ESB Mediator](#)
- [Places for Putting Custom Mediators](#)
- [Writing Custom Mediator Implementations](#)
- [Uploading Artifacts](#)

### Working with Connectors

A connector is a collection of [templates](#) that define operations users can call from their ESB configurations to easily access specific logic for processing messages. Typically, connectors are used to wrap the API of an external service such as Twitter or Google Spreadsheet. Each connector provides operations that perform different actions in that service. For example, the Twitter connector has operations for creating a tweet, getting a user's followers, and more.

WSO2 ESB provides a variety of connectors that allow your message flows to connect to and interact with services such as Twitter and Salesforce. To download ESB connectors, go to the [WSO2 Connector Store](#). You can also [write your own connector](#) to provide access to other services. Before you use the ESB connectors or a custom connector, you need to add and enable them in your ESB instance.

The following topics provide more information on working with connectors:

- [Working with Connectors via WSO2 ESB Tooling](#)
- [Working with Connectors via the Management Console](#)
- [Using a Connector](#)

To browse through the complete list of WSO2 ESB connectors and for information on configuring operations for each connector, see [WSO2 ESB Connectors Documentation](#)

## Working with Connectors via WSO2 ESB Tooling

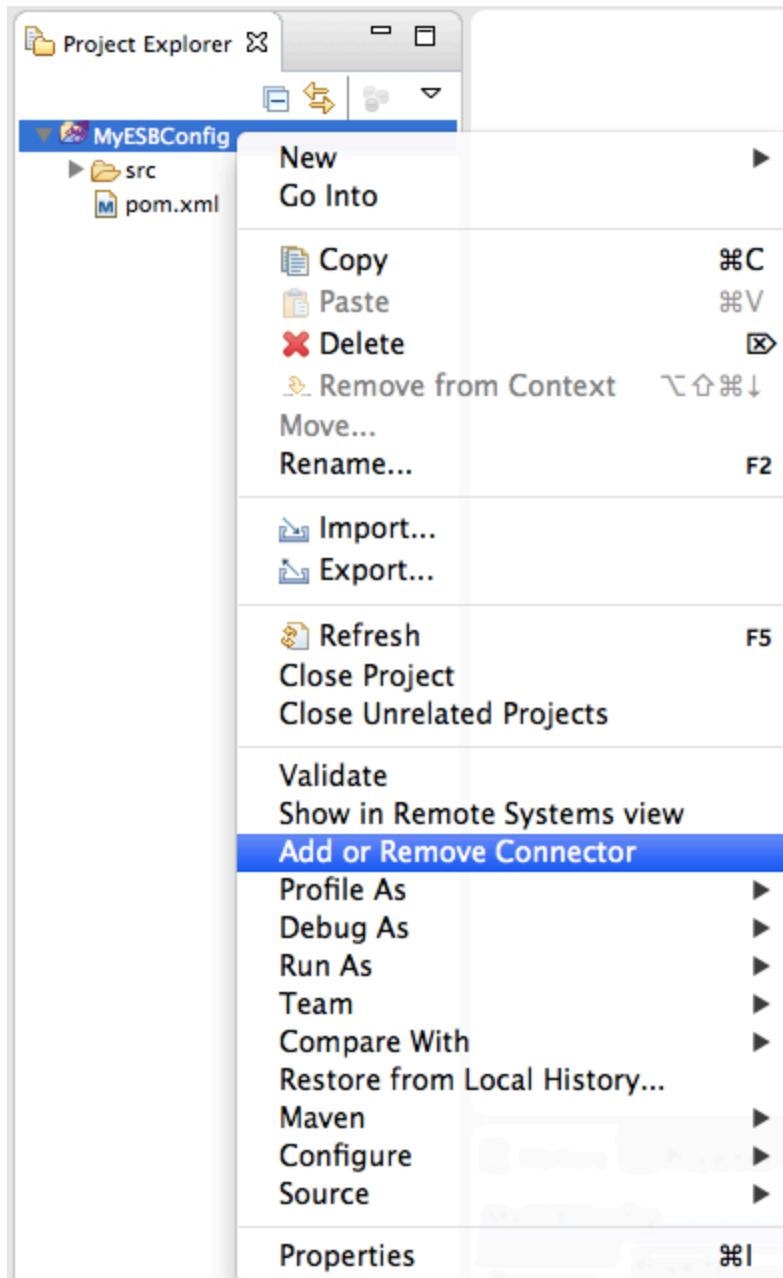
You can develop configurations with connectors, and deploy the configurations and connectors as composite application archive (CAR) files into WSO2 Enterprise Service Bus using WSO2 ESB tooling.

You need to have WSO2 ESB tooling installed to import a connector and create configurations using the connector operations via ESB tooling. For instructions on installing WSO2 ESB tooling, see [Installing WSO2 ESB Tooling](#).

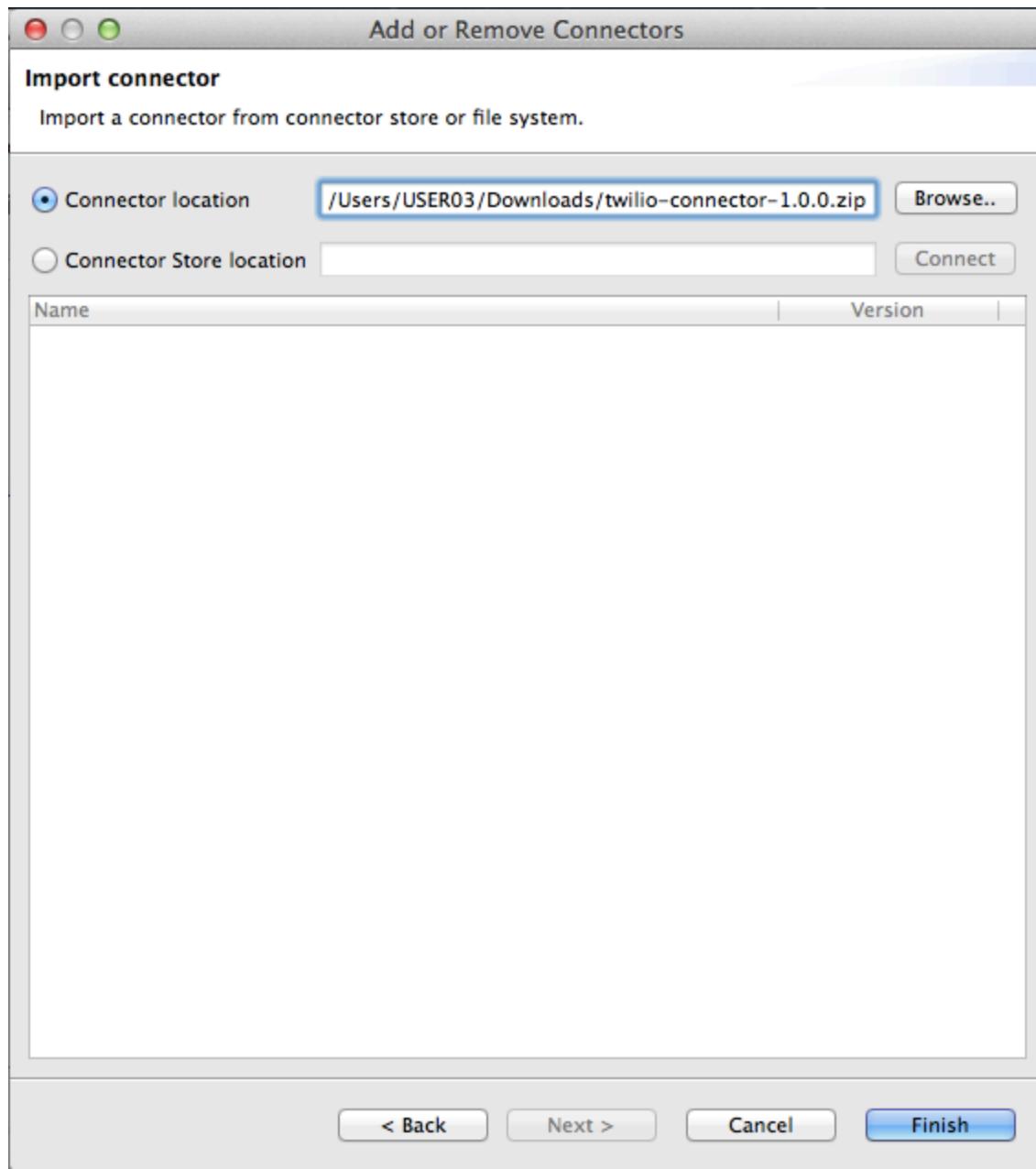
### ***Importing connectors***

Follow the steps below to import connectors into WSO2 ESB tooling:

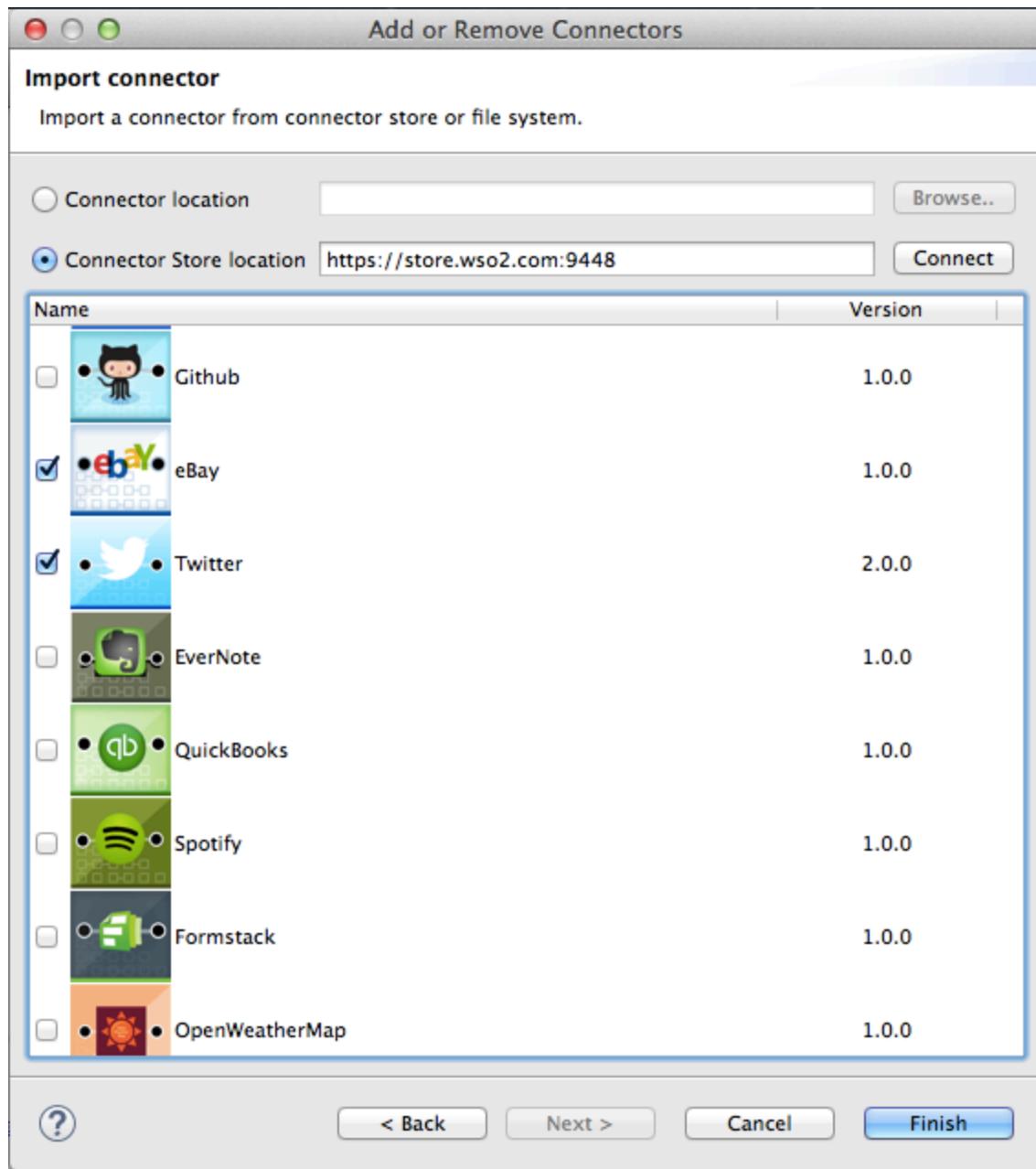
1. Right-click on the ESB Config project where you want to use the connector and click **Add or Remove Connector**.



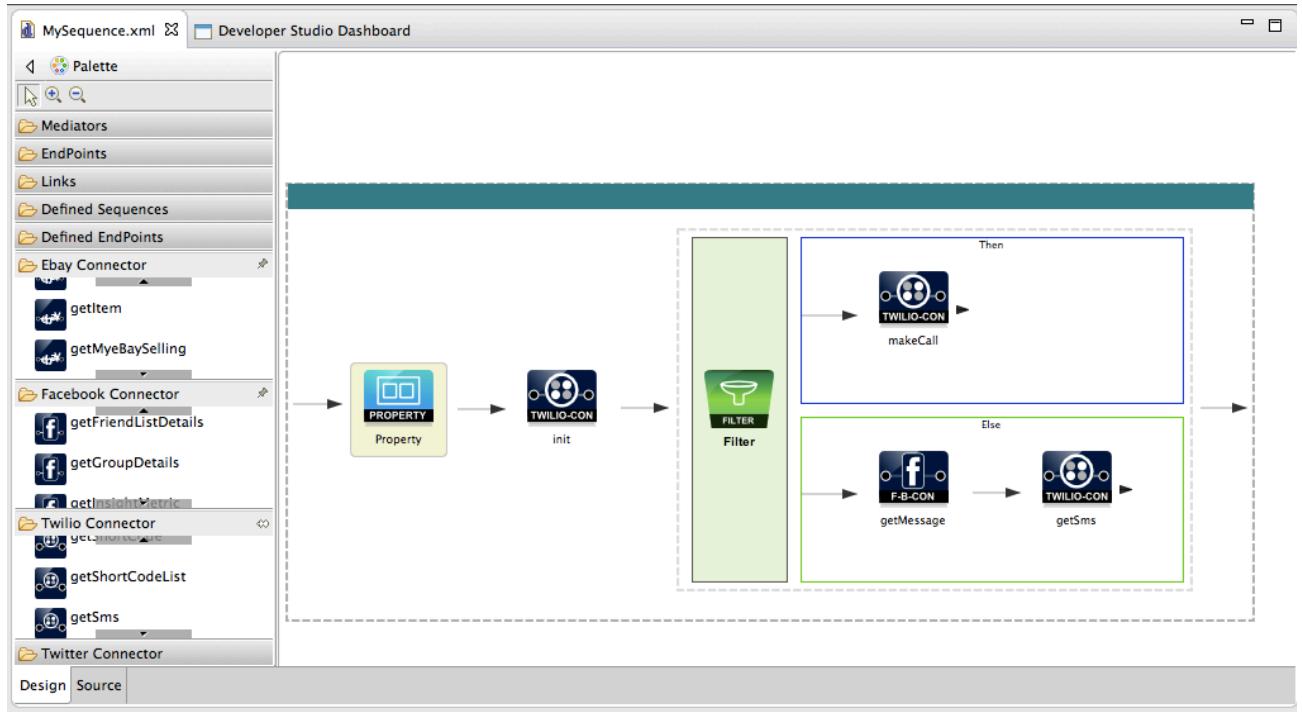
2. On the wizard that appears, select **Add Connector** and click **Next**.
  - a. If you have already downloaded the connectors, select the **Connector location** option and browse to the connector file from the file system. Click **Finish**. The connector is imported into the workspace and available for use with all the ESB projects in the workspace.



- b. If you have not downloaded any connectors, select the **Connector Store location** option to connect to the [connector store](https://store.wso2.com:9448) from Developer Studio and import the required connectors into the workspace. Set <https://store.wso2.com:9448> as the connector store location and click **Connect**. Select the required connectors and click **Finish**.



3. After importing the connectors into Developer Studio, the connector operations are available in the tool palette. You can drag and drop connector operations into your sequences and proxy services.

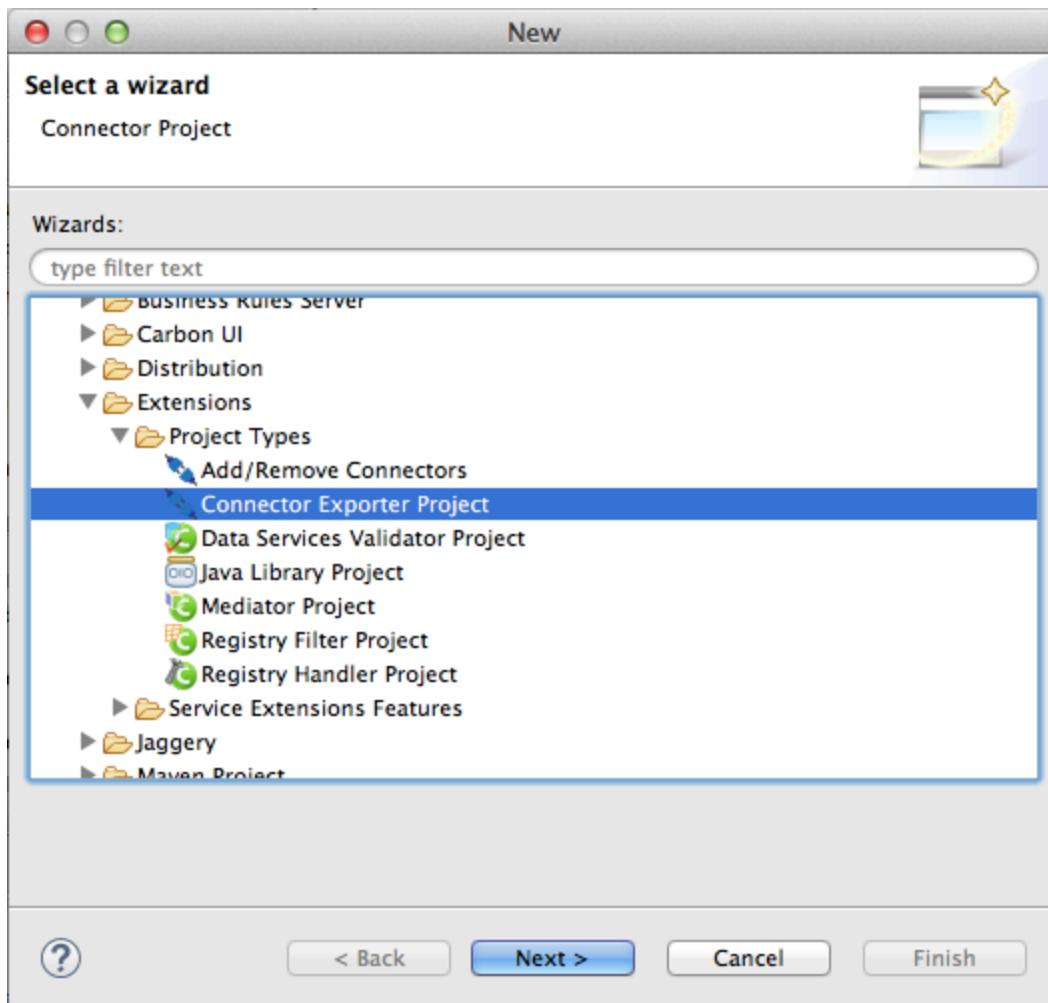


For complete information on each of the predefined connectors, see [ESB connectors](#).

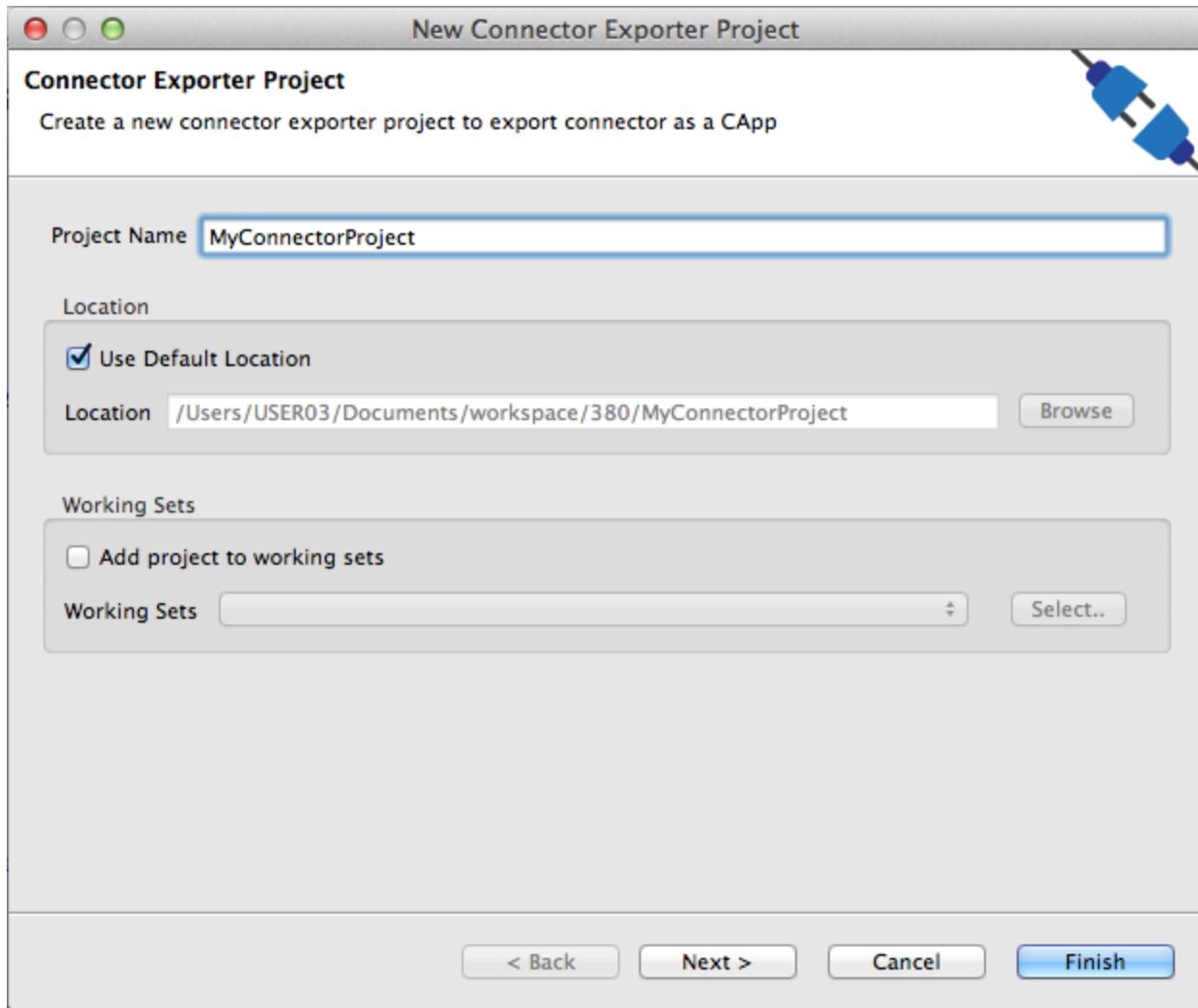
#### ***Creating a CAR file including connectors***

Follow the steps below to create a composite application archive (CAR) file containing the connectors:

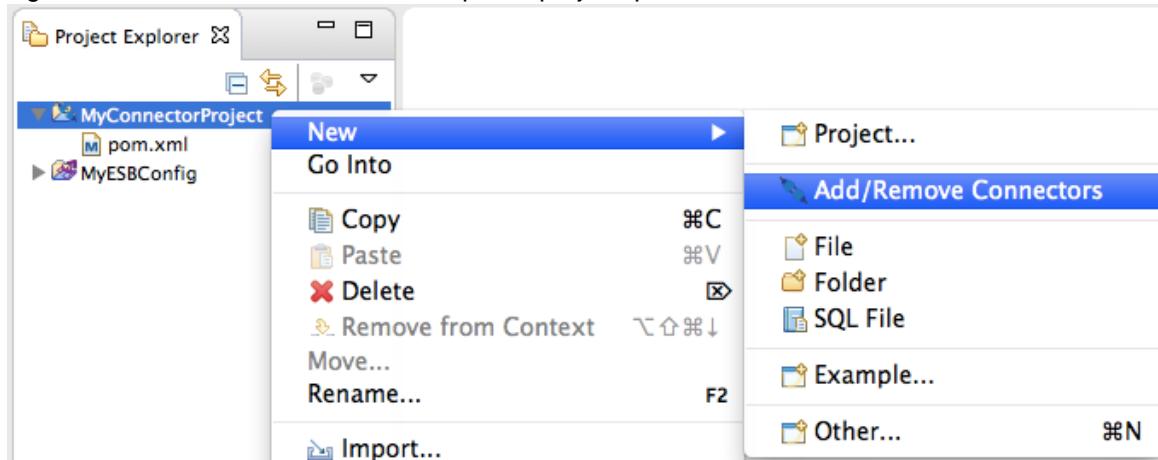
1. Click **File > New > Other** and select **Connector Exporter Project** under **WSO2 > Extensions > Project Types** and click **Next**.



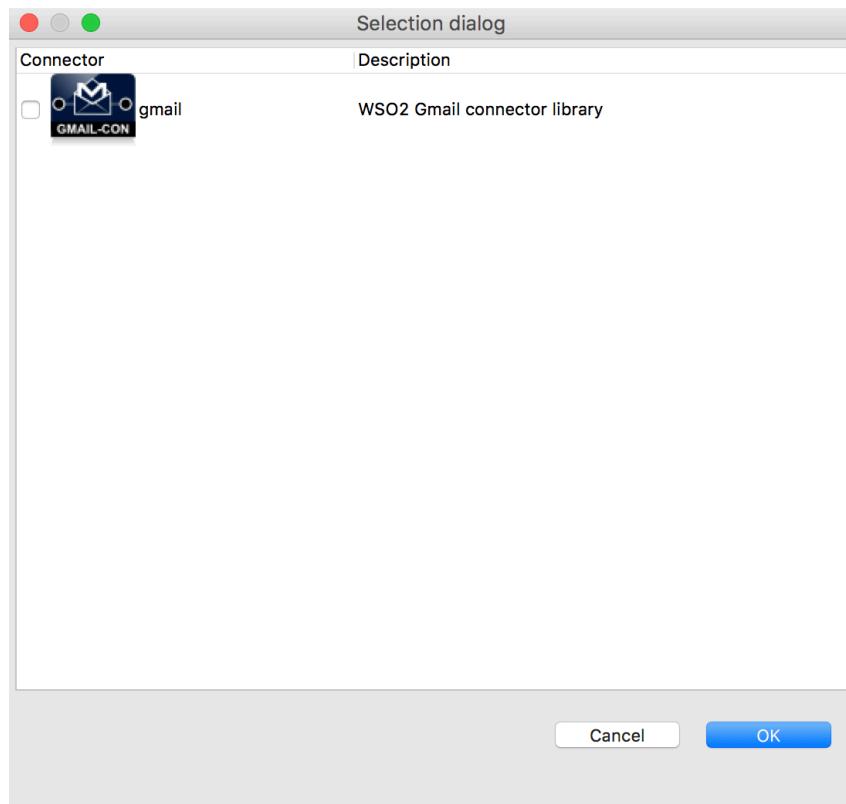
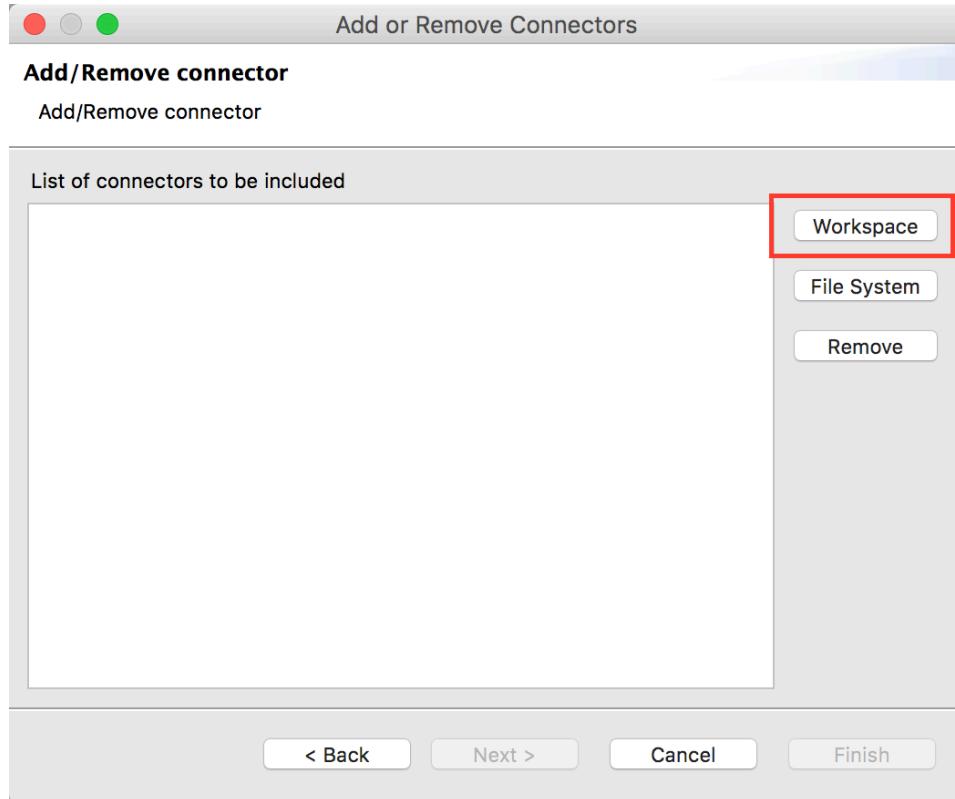
2. Alternatively, open the Developer Studio Dashboard (click **Developer Studio > Open Dashboard**) and click **Connector Exporter Project** in the Enterprise Service Bus area.
3. Enter a project name and click **Finish**.



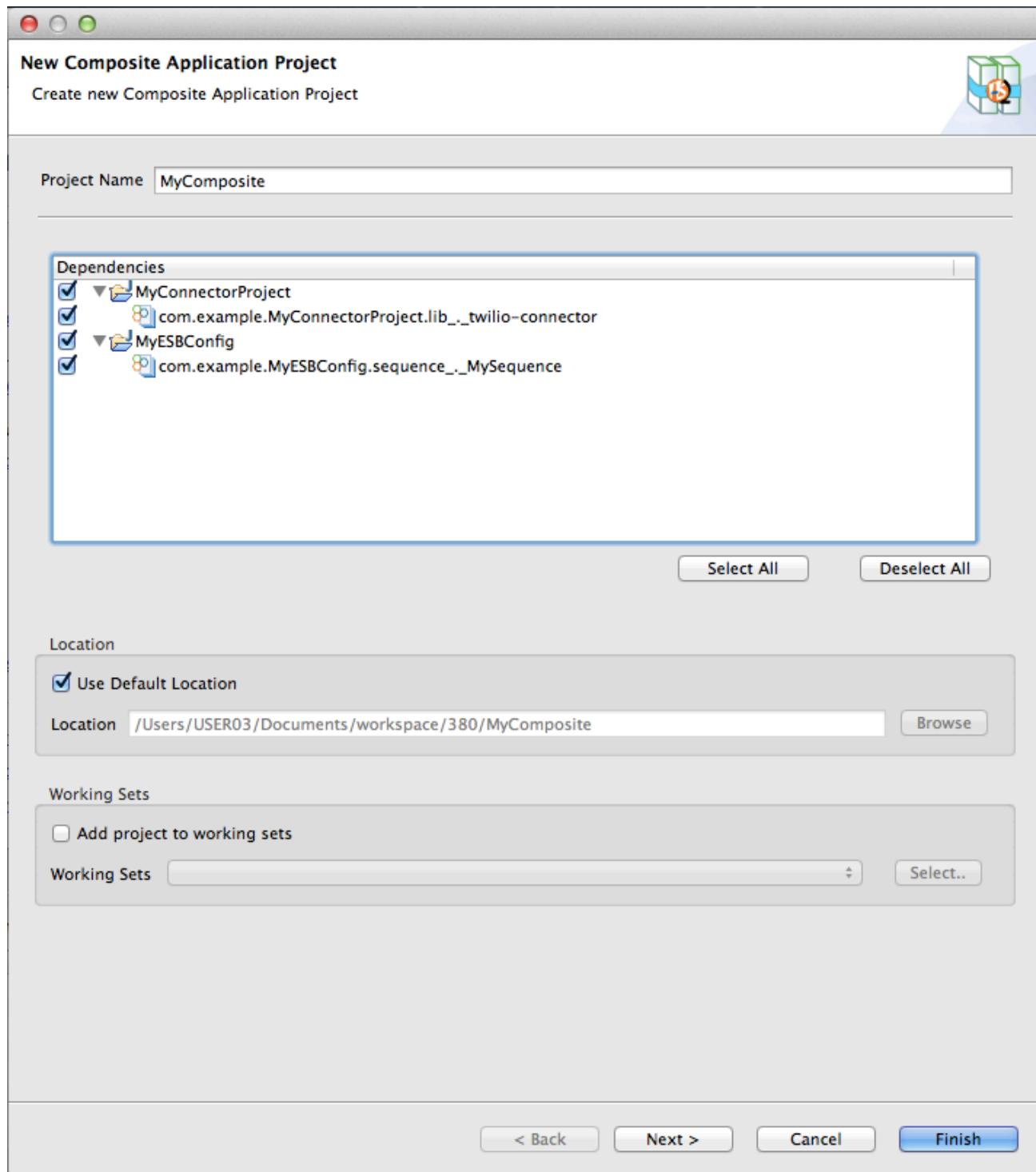
- Right-click on the created connector exporter project, point to **New** and then click **Add/Remove Connectors**.



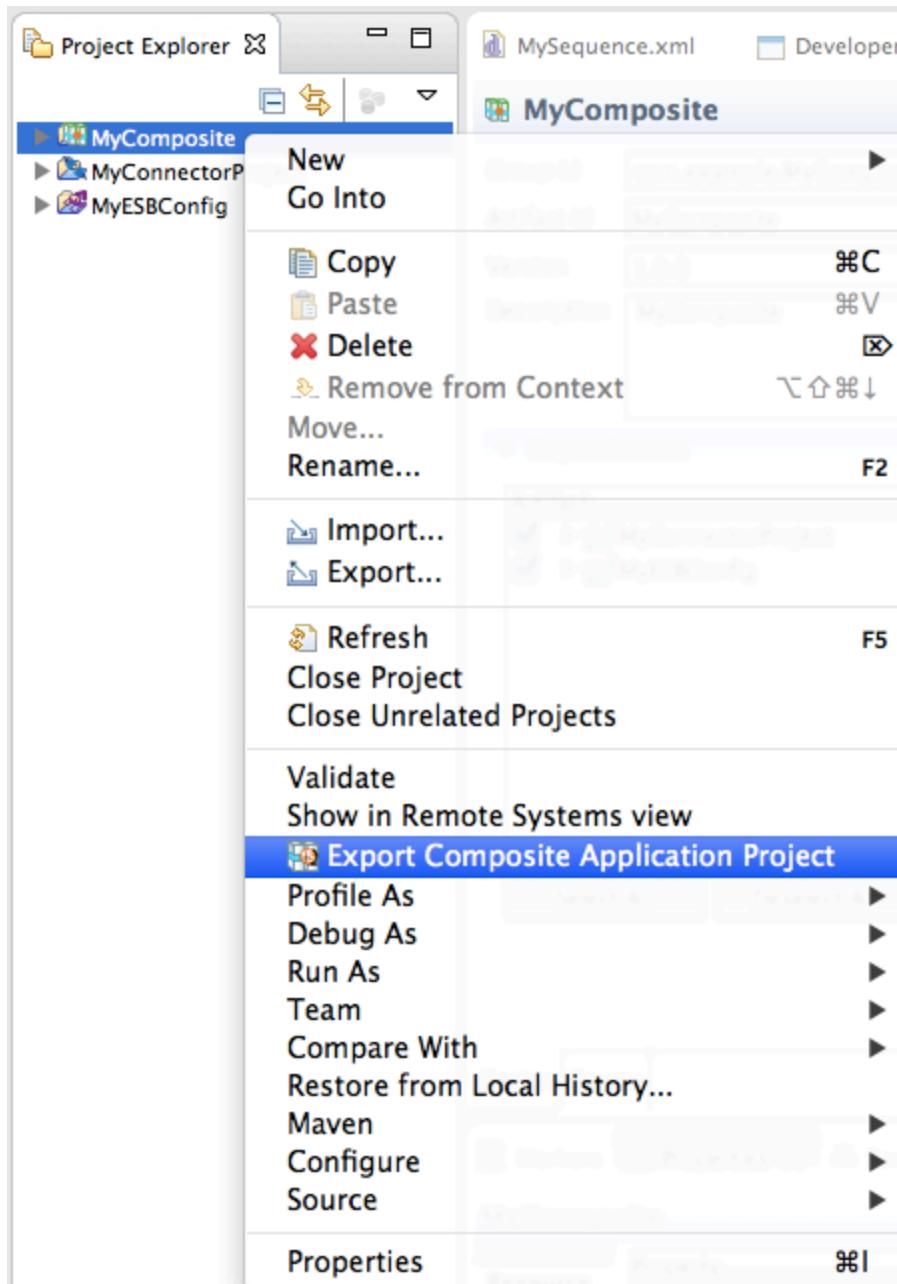
- Click **Add Connector** and then select **Workspace**. This will list down the connectors that have been imported into the ESB Tooling environment workspace.



6. Select the connector and click **OK**.
7. Create a Composite Application (C-App) project including the required artifacts.



8. Right-click on the C-App project and click **Export Composite Application Project** to create a CAR file out of that project.

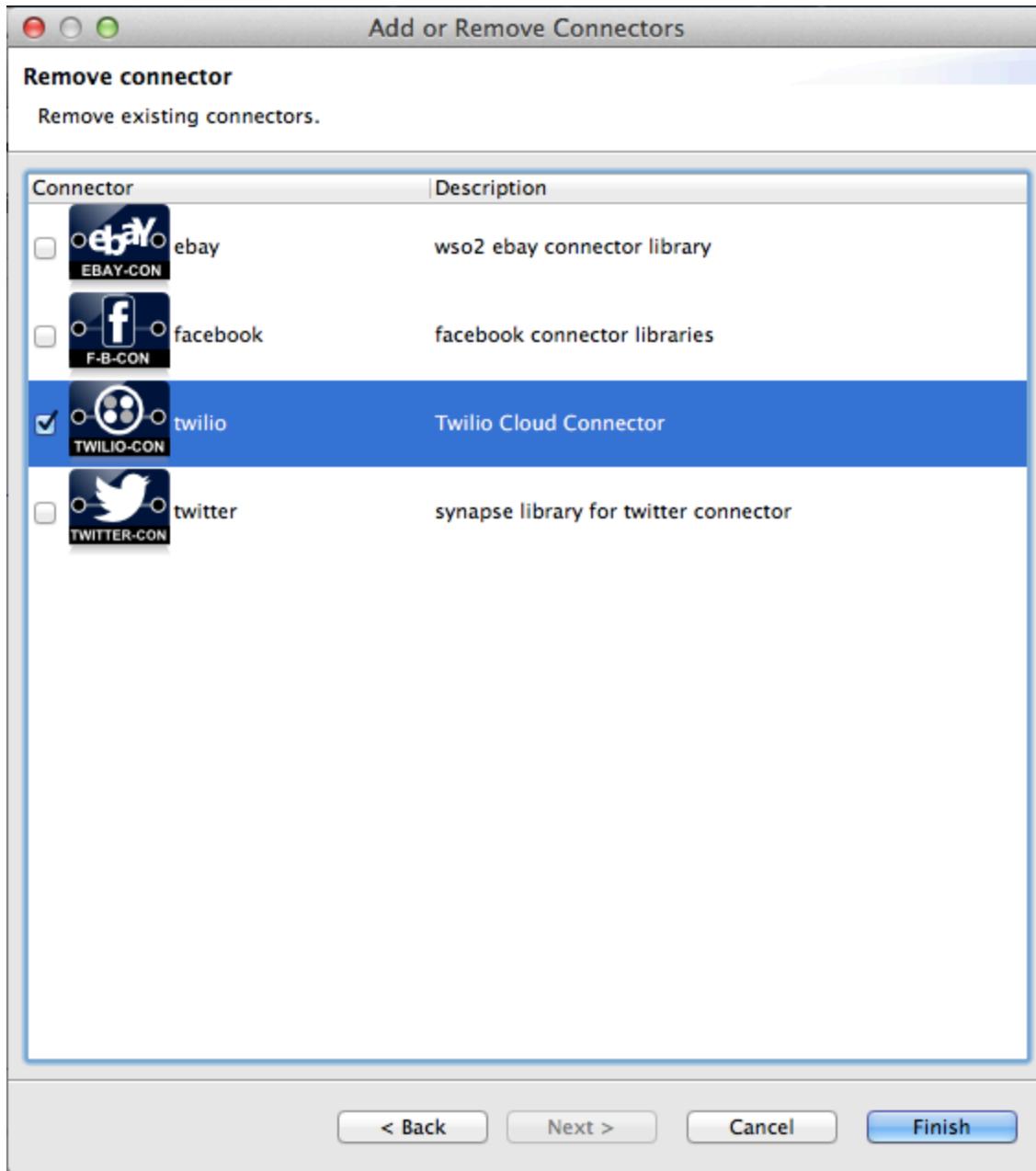


For more information, see [Packaging Artifacts into Composite Applications](#).

### **Removing connectors**

Follow the steps below to remove connectors from WSO2 ESB tooling:

1. Right-click on the relevant ESB Config project and click **Add or Remove Connector**.
2. On the wizard that appears, select **Remove Connector** and click **Next**.
3. Select the connectors you want to remove and click **Finish**.



## Working with Connectors via the Management Console

There are over a hundred [WSO2 ESB connectors](#) that you can download from the [WSO2 Connector Store](#). You can also [write your own connectors](#) and make them available to other users as ZIP files.

Before you can use a connector, you must add it to your ESB instance and then enable it. See the following topics for information on how to manage connectors via the Management Console:

- [Adding a connector](#)
- [Enabling/disabling a connector](#)

### Adding a connector

Follow the steps given below to add a connector.

1. On the **Main** tab in the ESB Management Console, under **Connectors** click **Add**.
2. Click **Browse**, specify the ZIP file, and click **Open**.
3. Click **Upload**.

The connector will now appear in the Connectors list and is ready to be enabled in your ESB instance.

### Enabling/disabling a connector

Follow the steps given below to enable a connector.

1. On the **Main** tab in the ESB Management Console, under **Connectors** click **List** to view the uploaded connectors.
2. Click **Enable** next to a connector you want to enable, and then confirm that you want to change its status. Repeat this step for each connector you want to enable.

Follow the steps given below to disable a connector.

- If you want a connector to be temporarily unavailable in your ESB instance, click **Disable**.
- If you want to remove the connector permanently from your instance, click **Delete**.

### Using a Connector

When a connector is [enabled in your ESB instance](#), you can access its functionality by adding a reference to it from your ESB configuration (such as in a sequence) and then calling its operations. To see the operations available in a connector, display your list of connectors (on the Main tab of the ESB Management Console, under Connectors click **List**), and then click the connector name.

For example, if the JIRA connector is enabled in your ESB instance and you want to use it in a sequence, you would add the following entry to your sequence configuration to connect to JIRA:

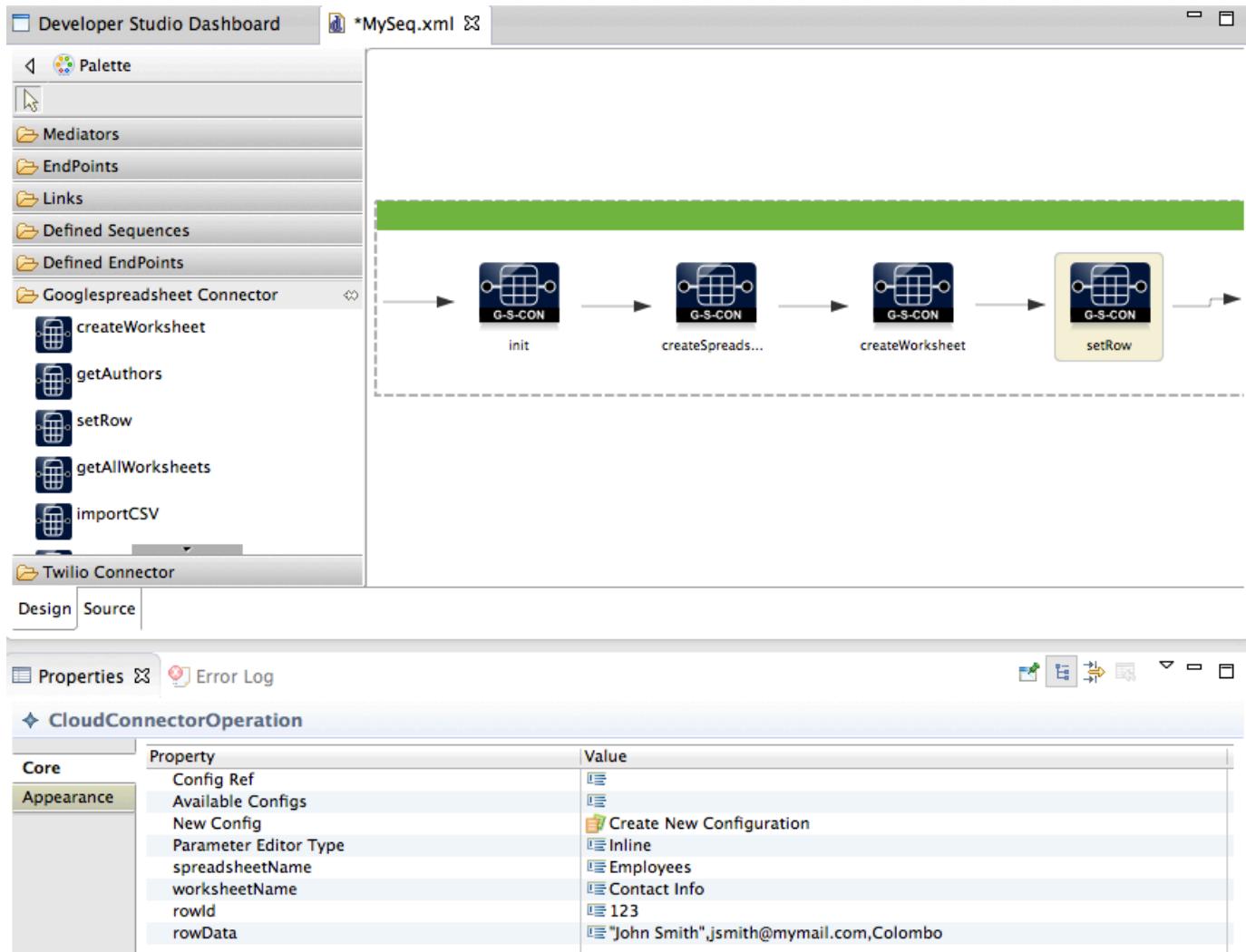
```
<jira.init>
 <username>myname@mycompany.com</username>
 <password>{wso2:vault-lookup('xxxx.email.login')}</password>
 <uri>https://xxx:xx/xx</uri>
</jira.init>
```

Then, if you want to get an issue, you can use the JIRA connector's `getIssue` operation and pass in the issue ID as follows:

```
<jira.getIssue [configKey="Ref Local Entry"]>
 <issueIdOrKey>ESBJAVA-2095</issueIdOrKey>
</jira.getIssue>
```

### Using connectors in Developer Studio

If you are using [Developer Studio](#), you can import a connector into your ESB config project by right-clicking the project, choosing **Import Connector**, and then navigating to the connector's ZIP file (for a list of predefined ESB connectors you can download, see [ESB Connectors](#)). The operations in the connector are then displayed in the graphical ESB editor, allowing you to drag them to your proxy service, sequence, or other configuration.

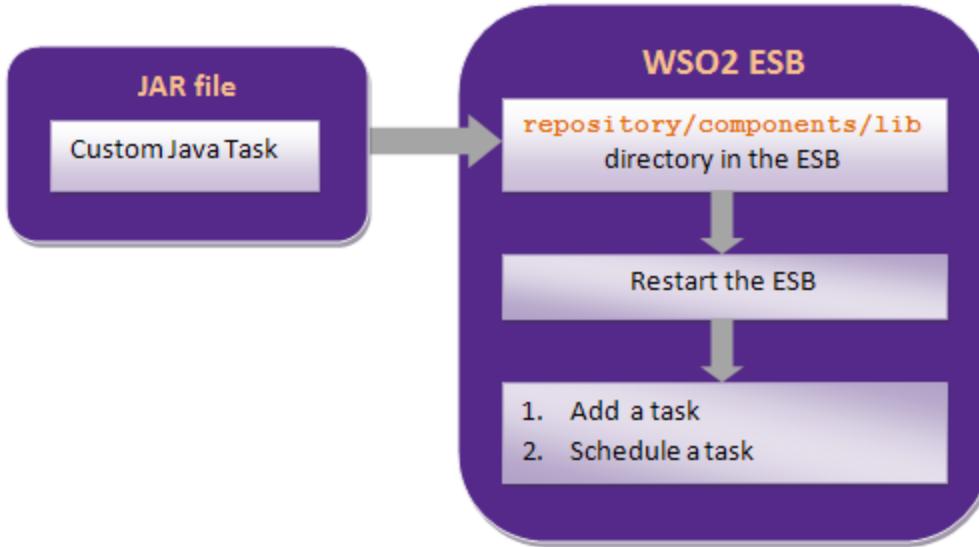


Notice that the first operation in the workflow is `init`, which initializes and establishes the connection to the service (in this example, it contains the authentication information for connecting to Google Spreadsheets). This connection is used by all subsequent operations in the flow unless you specify a different `init` configuration for a specific operation. To specify an `init` that has been saved as a local entry, enter its name in the `Config Ref` property. To create a new `init` for this operation, click the `New Config` property row, click the browse button that appears, and then enter the initialization parameters.

For more information on using connectors in Developer Studio, see [Working with Connectors](#) in the Developer Studio documentation.

## Writing Tasks

The following diagram illustrates the process of creating and scheduling a custom task implementation.



The main steps while writing a task are as follows:

1. Write the Task class
2. Customize the task
3. Compile and bundle the task
4. Add the task to the WSO2 ESB class path
5. Configure and schedule the task in ESB Console

#### **Step 1. Write the Task Class**

The custom Task class should implement `org.apache.synapse.startup.Task`. Each task should therefore implement the `Task` interface. This interface has a single `execute()` method. This method contains the code that is to be run at the specified intervals.

```

package org.apache.synapse.task;

/** Represents an executable Task*/

public interface Task {

 /** Execute method will be invoked by the QuartzJob.
 */
 public void execute();
}

```

The `execute()` method contains following actions:

1. Check whether the file exists at the desired location.
2. If it does, then read the file line by line composing place order messages for each line in the text file.
3. Individual messages are then injected to the synapse environment with the given `To endpoint` reference.
4. Set each message as `OUT_ONLY` since it is not expected any response for messages.

In addition to the `execute()` method, it is also possible to make the class implement a `JavaBean` interface. The WSO2 ESB console can then be used to configure the properties of this `JavaBean`.

See [Writing Tasks Sample](#) to learn more information on how to write tasks in Java. You can also use JavaScript, Ruby, Groovy or other Apache BSF scripting languages for this purpose.

## Step 2. Customize the Task

It is possible to pass values to a task at run time using property elements. When creating a `Task` object, WSO2 ESB will initialize the properties with the given values in the configuration file. For those properties given as XML elements, properties need to be defined within the `Task` class using the following format:

```
public void setMessage(_property_ elem) {
 message = elem;
}
```

It can be initialized with an XML element as follows:

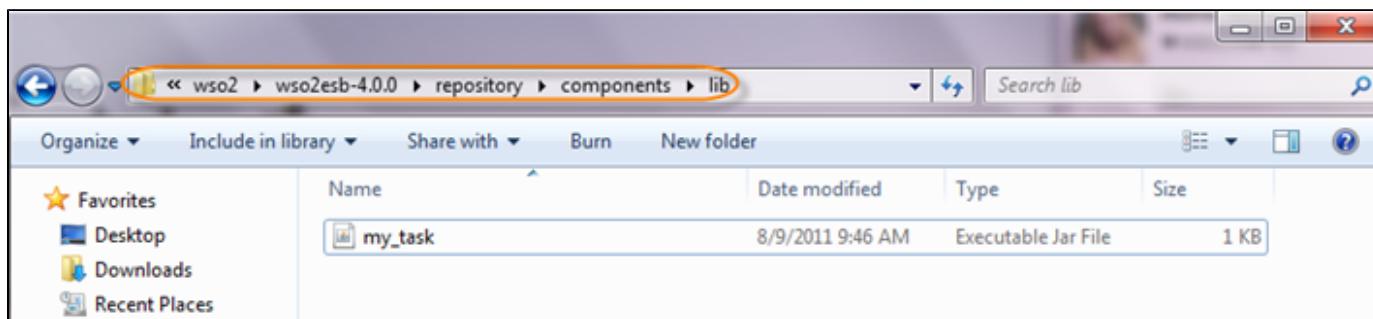
```
<property name="message">
<m0:getQuote xmlns:m0="http://services.samples/xsd">
<m0:request>
<m0:symbol>IBM</m0:symbol>
</m0:request>
</m0:getQuote>
</property>
```

## Step 3. Compile and bundle the task

Assemble the compiled class `Task` as a JAR file.

## Step 4. Add to the WSO2 ESB Class Path

After compiling and bundling the `Task` class, you need to add it to the WSO2 ESB class path. Place the JAR file to the `repository/components/lib` directory of the ESB.



The `Task` class will be available for use from the next time you start WSO2 ESB.

### Notice

It is required to restart the ESB for the JAR containing the task implementation to be picked up by the server runtime. An OSGi bundle containing the task implementation will be created automatically and it will be deployed in the server.

## Step 5. Configure and Schedule in ESB Console

See [Adding and Scheduling Tasks](#).

For more information see <http://wso2.org/library/2900>.

## Writing Tasks Sample

This page describes how to write a custom task. To better illustrate the topic, we provide instructions based on the example.

The created **task** will read a text file at a specified location and places orders for stocks that are given in the text file is created. The text file has entries such as this one:

```
IBM,100,120.50
```

Each line in the text file contains details for a stock order:

- symbol
- quantity
- price

The "PlaceStockOrderTask" will read the text file, a line at a time, and create orders using the given values to be sent to a sample Axis2 server. The text file will then be tagged as processed to include a system time stamp. The task will be scheduled to run every 15 seconds.

The main steps while writing a task are as follows:

1. Write the Task class
2. Customize the task
3. Compile and bundle the task
4. Add the task to the WSO2 ESB class path
5. Configure and schedule the task in ESB Console

### Step 1. Write the Task Class

The PlaceStockOrderTask class implements `org.apache.synapse.startup.Task`. Each task should therefore implement the `Task` interface. This interface has a single `execute()` method. This method contains the code that is to be run at the specified intervals.

The `execute()` method contains the following actions:

1. Check whether the file exists at the desired location.
2. If it does, then read the file line by line composing place order messages for each line in the text file.
3. Individual messages are then injected to the synapse environment with the given `To endpoint` reference.
4. Set each message as `OUT_ONLY` since it is not expected any response for messages.

In addition to the `execute()` method, it is also possible to make the class implement a JavaBean interface. The WSO2 ESB console can then be used to configure the properties of this JavaBean.

The complete code listing of the Task class is provided below:

```
public class PlaceStockOrderTask implements Task, ManagedLifecycle {
 private Log log = LogFactory.getLog(PlaceStockOrderTask.class);
 private String to;
 private String stockFile;
 private SynapseEnvironment synapseEnvironment;

 public void execute() {
 log.debug("PlaceStockOrderTask begin");

 if (synapseEnvironment == null) {
 log.error("Synapse Environment not set");
 }
 }
}
```

```

return; }

if (to == null) {
log.error("to not set");
return; }

File existFile = new File(stockFile);

if(!existFile.exists()) {
log.debug("waiting for stock file");
return; }

try {

// file format IBM,100,120.50

BufferedReader reader = new BufferedReader(new FileReader(stockFile));
String line = null;

while((line = reader.readLine()) != null){
line = line.trim();

if(line == "") {
continue;
}

String[] split = line.split(",");
String symbol = split[0];
String quantity = split[1];
String price = split[2];
MessageContext mc = synapseEnvironment.createMessageContext();
mc.setTo(new EndpointReference(to));
mc.setSoapAction("urn:placeOrder");
mc.setProperty("OUT_ONLY", "true");
OMElement placeOrderRequest = createPlaceOrderRequest(symbol, quantity, price);
PayloadHelper.setXMLPayload(mc, placeOrderRequest);
synapseEnvironment.injectMessage(mc);
log.info("placed order symbol:" + symbol + " quantity:" + quantity + " price:" +
price);
}

reader.close();
}

catch (IOException e) {
throw new SynapseException("error reading file", e);
}

File renamefile = new File(stockFile);
renamefile.renameTo(new File(stockFile + "." + System.currentTimeMillis()));
log.debug("PlaceStockOrderTask end"); }

public static OMElement createPlaceOrderRequest(String symbol, String qty, String
purchPrice) {
OMFactory factory = OMAbstractFactory.getOMFactory();
OMNamespace ns = factory.createOMNamespace("http://services.samples/xsd", "m0");
OMElement placeOrder= factory.createOMElement("placeOrder", ns);
OMElement order = factory.createOMElement("order", ns);
OMElement price = factory.createOMElement("price", ns);
}

```

```
OMElement quantity = factory.createOMEElement("quantity", ns);
OMElement symb = factory.createOMEElement("symbol", ns);
price.setText(purchPrice);
quantity.setText(qty);
symb.setText(symbol);
order.addChild(price);
order.addChild(quantity);
order.addChild(symb);
placeOrder.addChild(order);
return placeOrder;
}

public void destroy() {
}

public void init(SynapseEnvironment synapseEnvironment) {
this.synapseEnvironment = synapseEnvironment;
}

public SynapseEnvironment getSynapseEnvironment() {
return synapseEnvironment;
}

public void setSynapseEnvironment(SynapseEnvironment synapseEnvironment) {
this.synapseEnvironment = synapseEnvironment;
}

public String getTo() {
return to;
}

public void setTo(String to) {
this.to = to;
}

public String getStockFile() {
return stockFile;
}

public void setStockFile(String stockFile) {
```

```

this.stockFile = stockFile;
}
}

```

When creating the customized schedule tasks, if the injecting sequence of the message flow contains Publish Event mediators, set the following property in the Synapse message context:

```
mc.setProperty("CURRENT_TASK_EXECUTING_TENANT_IDENTIFIER", PrivilegedCarbonContext.getThreadLocalCarbonContext().getTenantId());
```

Also, add the following dependency to the POM file of the custom task project: WSO2 Carbon - Utilities bundle (symbolic name: org.wso2.carbon.utils)

This is a bean implementing two properties - To and StockFile. These are used to configure the task. Implementing ManagedLifecycle for Initialization and Cleanup

Since a task implements ManagedLifecycle interface, ESB will call the `init()` method at the initialization of a Task object and `destroy()` method when a Task object is destroyed:

```

public interface ManagedLifecycle {
public void init(SynapseEnvironment se);
public void destroy();
}

```

In PlaceStockOrderTask it is stored the Synapse environment object reference in an instance variable for later use with the `init()` method. The `SynapseEnvironment` is needed to inject messages into the ESB.

## Step 2. Customize the Task

It is possible to pass values to a task at run time using property elements. In this example, the location of the stock order file and its address was given using two properties within the Task object.

The properties can be:

- **String type**
- **OMEElement type**

### Tip

For OMEElement type, it is possible to pass XML elements as values in the configuration file.

When creating a Task object, WSO2 ESB will initialize the properties with the given values in the configuration file.

For example, the following properties in the Task class

```

public String getStockFile() {
return stockFile;
}
public void setStockFile(String stockFile) {
this.stockFile = stockFile;
}

```

are initialized with the given values within the property element of the task in the configuration.

```
<syn:property xmlns="http://ws.apache.org/ns/synapse"
name="stockFile" value="/home/upul/test/stock.txt"/>
```

For those properties given as XML elements, properties need to be defined within the Task class using the following format:

```
public void setMessage(OMELEMENT elem) {
 message = elem;}
```

It can be initialized with an XML element as follows:

(OMELEMENT comes from [Apache AXIOM](#) which is used by WSO2 ESB. AXIOM is an object model similar to DOM. To learn more about AXIOM, see the tutorial in the [AXIOM user guide](#).)

```
<property name="message">
<m0:getQuote xmlns:m0="http://services.samples/xsd">
<m0:request>
<m0:symbol>IBM</m0:symbol>
</m0:request>
</m0:getQuote>
</property>
```

### Step 3. Compile and Bundle the Task

Assemble the compiled class Task as a JAR file.

### Step 4. Add to the WSO2 ESB Class Path

After compiling and bundling the Task class, you need to add it to the WSO2 ESB class path. Place the JAR file to the `repository/components/lib` directory of the ESB.

The Task class will be available for use from the next time you start WSO2 ESB.

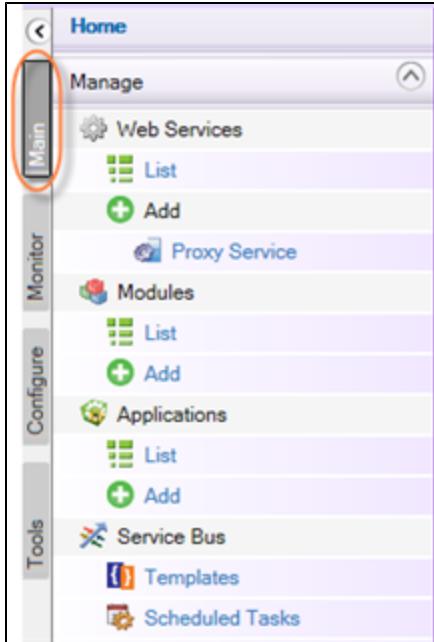
#### Notice

It is required to restart the ESB for the JAR containing the task implementation to be picked up by the server runtime. An OSGi bundle containing the task implementation will be created automatically and it will be deployed in the server.

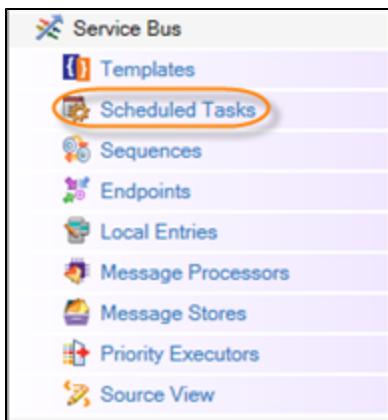
### Step 5. Configure and Schedule in ESB Console

After adding the Task class to class path and restarting the WSO2 ESB, you can add configuration settings using the ESB console. Follow the steps given below (for the detailed information on how to schedule tasks see [Adding and Scheduling Tasks](#)):

1. Log in to the ESB console.
2. Click the "Main" button to access the "Manage" menu.



3. Click "Scheduled Tasks" in the "Manage" group.



4. Click "Add Task."

The screenshot shows the "Scheduled Tasks" page. The title is "Scheduled Tasks". Below it is a section titled "Available Defined Scheduled Tasks". A table lists one task: "Task1" in the "Task Name" column and "Edit" and "Delete" buttons in the "Action" column. At the bottom of the table is a green "Add Task" button, which is highlighted with an orange circle.

Task Name	Action
Task1	Edit  Delete

5. The "New Scheduled Task" page will open. Enter "placeOrder" as "Task Name."

**New Scheduled Task**

<b>New Scheduled Task</b>	
Task Name*	<input type="text" value="placeOrder"/> <span style="border: 2px solid orange; border-radius: 50%; padding: 2px;">placeOrder</span>
Task Group*	<input type="text" value="synapse.simple.quartz"/>
Task Implementation*	<input type="text" value="org.apache.synapse.startup.tasks.MessageTask"/> <span style="border: 1px solid #ccc; padding: 2px;">Load Task Properties</span>
<b>Trigger Information of the Task</b>	
Trigger Type	<input checked="" type="radio"/> Simple <input type="radio"/> Cron
Count	<input type="text"/>
Interval (in seconds)*	<input type="text"/>
<b>Miscellaneous Information</b>	
Pinned Servers	<input type="text"/>
(separated by comma or space)	
<span style="border: 1px solid #ccc; padding: 2px 10px;">Schedule</span> <span style="border: 1px solid #ccc; padding: 2px 10px;">Cancel</span>	

6. Enter "org.wso2.esb.tutorial.tasks.PlaceStockOrderTask" as "Task Implementation."

Task Implementation*	<input type="text" value="org.wso2.esb.tutorial.tasks.PlaceStockOrderTask"/> <span style="border: 2px solid orange; border-radius: 50%; padding: 2px;">org.wso2.esb.tutorial.tasks.PlaceStockOrderTask</span> <span style="border: 1px solid #ccc; padding: 2px;">Load Task Properties</span>
----------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

7. Click "Load Task Properties."

<span style="border: 2px solid orange; border-radius: 50%; padding: 2px;">Load Task Properties</span>
-------------------------------------------------------------------------------------------------------

8. ESB console will load the custom Task class and show you the properties it finds.

9. Enter <http://localhost:9000/soap/SimpleStockQuoteService> for "To" property. Keep property type as "Literal."

10. Copy the stockfile.txt to a new directory. The stockfile.txt will contain the following entries:

```
IBM,100,120.50
MSFT,200,70.25
SUN,400,60.758.
```

Enter stock.txt file location for the stockFile property. Keep property type as "Literal."

11. Select the "Trigger Type" as "Simple."

<b>Trigger Information of the Task</b>	
Trigger Type	<input checked="" type="radio"/> Simple <input type="radio"/> Cron

12. Enter 15000 to "Interval."

Interval (in seconds)*	<input type="text" value="15000"/> <span style="border: 2px solid orange; border-radius: 50%; padding: 2px;">15000</span>
------------------------	---------------------------------------------------------------------------------------------------------------------------

13. Click "Schedule."

## New Scheduled Task

**New Scheduled Task**

Task Name*	placeOrder		
Task Group*	synapse.simple.quartz		
Task Implementation*	org.wso2.esb.tutorial.tasks.PlaceStockOrder <a href="#">Load Class</a>		
Property Name	Property Type	Property Value	Action
To	Literal	http://localhost:9000/soap/SimpleStockQuote	Delete
StockFile	Literal	/home/upul/test/stock	Delete
SynapceEnvironment	Literal		Delete

**Trigger Information of the Task**

Trigger Type	<input checked="" type="radio"/> Simple <input type="radio"/> Cron
Count	
Interval (in seconds)*	1500

**Miscellaneous Information**

Pinned Servers	
(separated by comma or space)	

[Schedule](#) [Cancel](#)

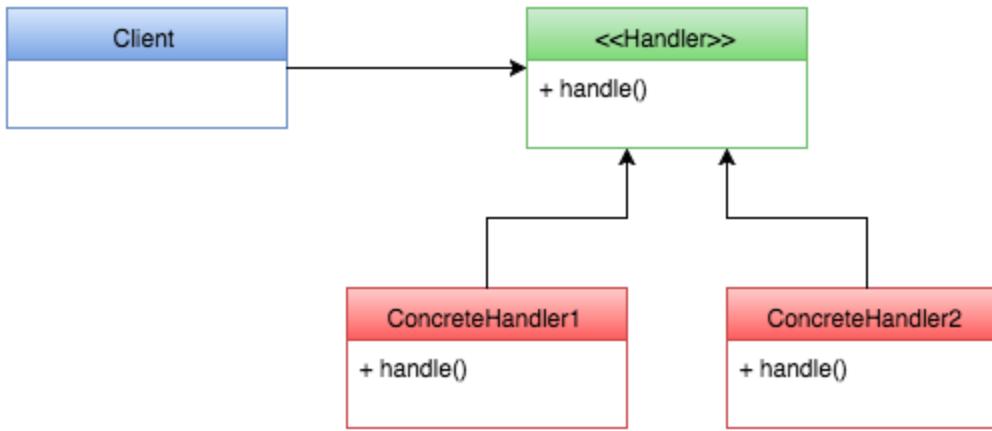
## Writing a Synapse Handler

This section gives an introduction to what a handler is and describes how you can write a synapse handler by walking you through a basic example.

- [Introduction to handlers](#)
- [Writing a concrete Synapse handler](#)
- [Deploying the Synapse handler](#)
- [Engaging the Synapse handler](#)

### ***Introduction to handlers***

Handlers can be used to process requests in a scenario where you have multiple requests, and each request needs be processed in a specific manner. A Handler defines the interface that is required to handle the request and concreteHandlers are to handle requests in a specific manner based on what needs to be done with regard to each type of request. The diagram below illustrates this.



Synapse handler is the interface used to register server response callbacks. Synapse handler provides the abstract handler implementation that executes the request in flow, request out flow, response in flow and response out flow.

The diagram below is an illustration of how the specified flows execute in the abstract handler implementation.



- **Request in flow**

```
public boolean handleRequestInFlow(MessageContext synCtx);
```

This executes when the request reaches the synapse engine.

- **Request out flow**

```
public boolean handleRequestOutFlow(MessageContext synCtx);
```

This executes when the request goes out of the synapse engine.

- **Response in flow**

```
public boolean handleResponseInFlow(MessageContext synCtx);
```

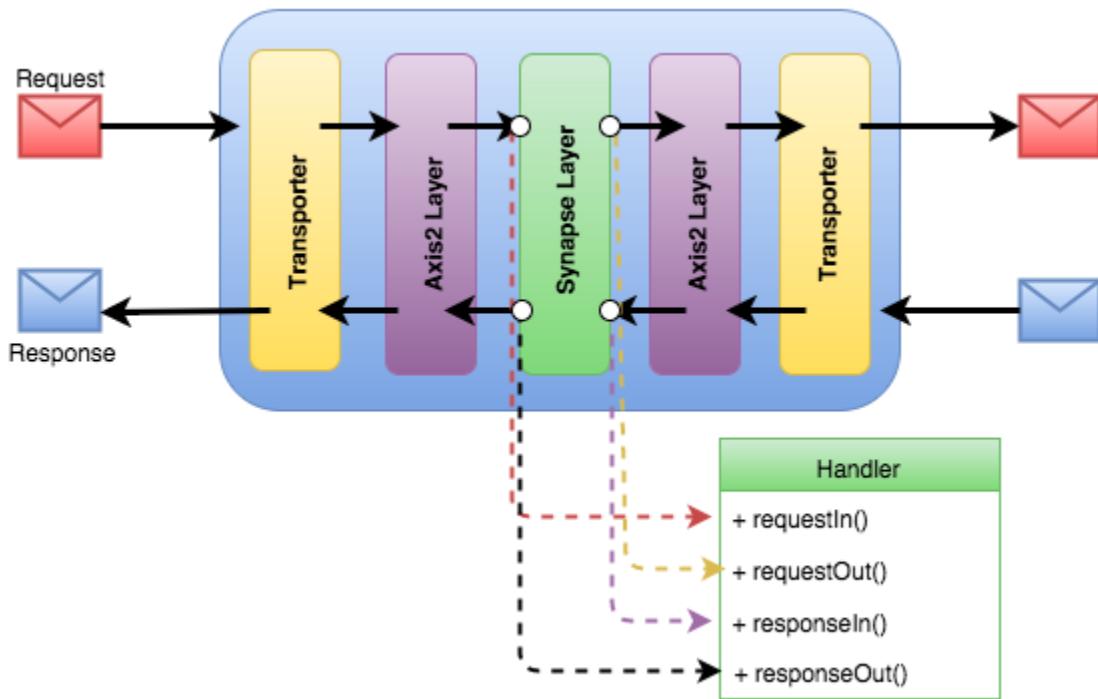
This executes when the response reaches the synapse engine.

- **Response out flow**

```
public boolean handleResponseOutFlow(MessageContext synCtx);
```

This executes when the response goes out of the synapse engine.

The diagram below illustrates the basic component structure of the ESB and how the flows mentioned above execute in the request path and response path.



Now that you understand what a handler is, let's see how you can write a concrete Synapse handler.

#### ***Writing a concrete Synapse handler***

The easiest way to write a concrete Synapse handler is to extend the `org.apache.synapse.AbstractSynapseHandler` class. You can also write a concrete Synapse handler by implementing `org.apache.synapse.SynapseHandler`, which is the `SynapseHandler` interface.

Following is an example Synapse handler implementation that extends the `org.apache.synapse.AbstractSynapseHandler` class:

```

public class TestHandler extends AbstractSynapseHandler {

 private static final Log log = LogFactory.getLog(TestHandler.class);

 @Override
 public boolean handleRequestInFlow(MessageContext synCtx) {
 log.info("Request In Flow");
 return true;
 }

 @Override
 public boolean handleRequestOutFlow(MessageContext synCtx) {
 log.info("Request Out Flow");
 return true;
 }

 @Override
 public boolean handleResponseInFlow(MessageContext synCtx) {
 log.info("Response In Flow");
 return true;
 }

 @Override
 public boolean handleResponseOutFlow(MessageContext synCtx) {
 log.info("Response Out Flow");
 return true;
 }
}

```

### ***Deploying the Synapse handler***

When you write your own Synapse handler you can deploy it either as an OSGi bundle or [JAR file](#) to the ESB.

### ***Engaging the Synapse handler***

To engage the deployed Synapse handler, you need to add the following configuration to the <ESB\_HOME>/repository/conf/synapse-handlers.xml file.

```

<handlers>
 <handler name="TestHandler" class="package.TestHandler" />
</handlers>

```

## **Writing a Custom Message Builder and Formatter**

In addition to using the default message builders and formatters in WSO2 ESB, you can create your own custom message builders and formatters.

- Custom message builder
- Custom message formatter

## Custom message builder

Let's look at how to create a custom message builder using a sample scenario where you need to Base64 encode an XML entry field. In this sample, you retrieve the text content from the payload and then Base64 encode the text. This is then converted to SOAP, and the content is then processed in the ESB mediation flow.

1. You will first need to write a class implementing the `org.apache.axis2.builder.Builder` interface in the Axis2 Kernel module and then override the `processDocument` method. Within the `processDocument` method, you can define your specific logic to process the payload content as required and then convert it to SOAP format.

```
package org.test.builder;

import org.apache.axiom.om.OMAbstractFactory;
import org.apache.axiom.om.OMELEMENT;
import org.apache.axiom.om.impl.OMNodeEx;
import org.apache.axiom.om.impl.builder.StAXBuilder;
import org.apache.axiom.om.impl.builder.StAXOMBuilder;
import org.apache.axiom.om.util.StAXParserConfiguration;
import org.apache.axiom.om.util.StAXUtils;
import org.apache.axiom.soap.SOAPBody;
import org.apache.axiom.soap.SOAPEnvelope;
import org.apache.axiom.soap.SOAPFactory;
import org.apache.axis2.AxisFault;
import org.apache.axis2.Constants;
import org.apache.axis2.builder.Builder;
import org.apache.axis2.context.MessageContext;
import org.apache.commons.codec.binary.Base64;

import javax.xml.stream.XMLStreamException;
import java.io.IOException;
import java.io.InputStream;
import java.io.PushbackInputStream;

public class CustomBuilderForTextXml implements Builder{
 public OMELEMENT processDocument(InputStream inputStream, String s,
MessageContext messageContext) throws AxisFault {
 SOAPFactory soapFactory = OMAbstractFactory.getSOAP11Factory();
 SOAPEnvelope soapEnvelope = soapFactory.getDefaultEnvelope();

 PushbackInputStream pushbackInputStream = new
PushbackInputStream(inputStream);

 try {
 int byteVal = pushbackInputStream.read();
 if (byteVal != -1) {
 pushbackInputStream.unread(byteVal);

 javax.xml.stream.XMLStreamReader xmlReader =
StAXUtils.createXMLStreamReader(StAXParserConfiguration.SOAP,
 pushbackInputStream, (String)
messageContext.getProperty(Constants.Configuration.CHARACTER_SET_ENCODING));

 StAXBuilder builder = new StAXOMBuilder(xmlReader);
 OMNodeEx documentElement = (OMNodeEx)
builder.getDocumentElement();
 documentElement.setParent(null);
 String elementVal = ((OMELEMENT) documentElement).getText();
 }
 } catch (XMLStreamException e) {
 throw new AxisFault("Error reading XML stream");
 } catch (IOException e) {
 throw new AxisFault("Error reading input stream");
 }
 }
}
```

```
byte[] bytesEncoded =
Base64.encodeBase64(elementVal.getBytes());
((OMElement) documentElement).setText(new String(bytesEncoded));
SOAPBody body = soapEnvelope.getBody();
body.addChild(documentElement);
}
} catch (IOException e) {
e.printStackTrace();
} catch (XMLStreamException e) {
e.printStackTrace();
}
}
```

```

 return soapEnvelope;
 }
}

```

2. Create a JAR file of this class and add it into the classpath of the Axis2 installation, i.e., the <ESB\_HOME>/repository/components/lib folder.
3. To enable your custom message builder for content type text/xml, add the following line in the Message Builders section in the <ESB\_HOME/repository/conf/axis2/axis2.xml> file:

```

<messageBuilder contentType="text/xml"
class="org.apache.axis2.transport.http.CustomBuilderForTextXml"/>

```

## Custom message formatter

Similarly, you can write your own message formatter to manipulate the outgoing payload from the ESB.

When creating a custom message formatter, you will need to create a class implementing the `org.apache.axis2.transport.MessageFormatter` interface and then override the `writeTo` method. You can implement your logic within the `writeTo` method.

Add the following line in the Message Formatters section in the <ESB\_HOME/repository/conf/axis2/axis2.xml> file:

```

<messageFormatter contentType= "text/xml"
class="org.apache.axis2.transport.http.SOAPMessageFormatter" />

```

The class name used in the above line should be the name used for the class when writing the formatter.

## Writing a WSO2 ESB Mediator

Mediators provide an easy way of extending the ESB functionalities.

There are two ways of writing an ESB mediator:

- **Using the Class Mediator** - This does not allow mediator specific XML configurations. See [Writing Custom Mediator Implementations](#) for more information.
- **Writing the mediator with factory and serialize methods** - This allows mediator to have its own XML configuration. See [Writing Custom Configuration Implementations for Mediators](#) for more information.

The easiest way to write a mediator is to extend your mediator class from the `org.apache.synapse.mediators.AbstractMediator` class. For example, you can see the following articles in the WSO2 library:

- [Writing a Mediator in WSO2 ESB - Part 1](#)
- [Writing a Mediator in WSO2 ESB - Part 2](#)

You can use the `Class mediator` and `custom mediators` for user-specific custom developments when there is no built-in mediator that already provides the required functionality. However, class and custom mediators incur a high maintenance overhead. `Custom mediators` in particular might introduce version migration complications when upgrading the ESB. Therefore, avoid using them unless the scenario is frequently re-used and heavily user-specific. For best results, use [WSO2 Developer Studio](#) for debugging Class and custom mediators.

## **Building the mediator**

After you write the mediator, you must build it with WSO2 ESB and make it an OSGI bundle so that it will work with the ESB.

### **Basic approach**

Create a regular JAR that links to the Synapse core JAR and place it in the <ESB\_HOME>/repository/components/lib directory. The platform will automatically make it an OSGI bundle and deploy it to the server.

### **Advanced approach**

If you want to control the way your mediator is created as an OSGI bundle, you must write the POM files so that you can export and import the packages you need, as shown in the examples below.

Following is a POM file that creates the mediator using [Class Mediator](#).

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/maven-v4_0_0.xsd">

 <modelVersion>4.0.0</modelVersion>
 <groupId>org.test</groupId>
 <artifactId>org.test</artifactId>
 <version>1.0.0</version>
 <packaging>bundle</packaging>
 <name>My Samples - Test mediator</name>
 <url>http://www.test.com</url>

 <repositories>
 <repository>
 <id>wso2-maven2-repository</id>
 <url>http://dist.wso2.org/maven2</url>
 </repository>
 <repository>
 <id>apache-Incubating-repo</id>
 <name>Maven Incubating Repository</name>
 <url>http://people.apache.org/repo/m2-incubating-repository</url>
 </repository>
 <repository>
 <id>apache-maven2-repo</id>
 <name>Apache Maven2 Repository</name>
 <url>http://repo1.maven.org/maven2/</url>
 </repository>
 </repositories>

 <build>
 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-compiler-plugin</artifactId>
 <version>2.0</version>
 <configuration>
 <source>1.5</source>
 <target>1.5</target>
 </configuration>
 </plugin>
 </plugins>
 </build>

```

```
<plugin>
 <groupId>org.apache.felix</groupId>
 <artifactId>maven-bundle-plugin</artifactId>
 <version>1.4.0</version>
 <extensions>true</extensions>
 <configuration>
 <instructions>
 <Bundle-SymbolicName>org.test</Bundle-SymbolicName>
 <Bundle-Name>org.test</Bundle-Name>
 <Export-Package>
 org.test.mediator.*,
 </Export-Package>
 <Import-Package>
 *; resolution:=optional
 </Import-Package>
 </instructions>
 </configuration>
</plugin>
</plugins>
</build>

<dependencies>
 <dependency>
 <groupId>org.apache.synapse</groupId>
 <artifactId>synapse-core</artifactId>
 <version>2.1.1-wso2v5</version>
 </dependency>
```

```
</dependencies>
</project>
```

The Maven bundle plug-in was used for creating the OSGI bundle here. Make sure you export the correct package that contains the mediator code. Otherwise, your mediator will not work.

Following is a POM file that creates the mediator with its own XML configuration using the `Serialize` and `Factory` classes.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">

 <modelVersion>4.0.0</modelVersion>
 <groupId>org.test</groupId>
 <artifactId>org.test</artifactId>
 <version>1.0.0</version>
 <packaging>bundle</packaging>
 <name>My Samples - Test mediator</name>
 <url>http://www.test.com</url>

 <repositories>
 <repository>
 <id>wso2-maven2-repository</id>
 <url>http://dist.wso2.org/maven2</url>
 </repository>
 <repository>
 <id>apache-Incubating-repo</id>
 <name>Maven Incubating Repository</name>
 <url>http://people.apache.org/repo/m2-incubating-repository</url>
 </repository>
 <repository>
 <id>apache-maven2-repo</id>
 <name>Apache Maven2 Repository</name>
 <url>http://repo1.maven.org/maven2/</url>
 </repository>
 </repositories>

 <build>
 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-compiler-plugin</artifactId>
 <version>2.0</version>
 <configuration>
 <source>1.5</source>
 <target>1.5</target>
 </configuration>
 </plugin>
 <plugin>
 <groupId>org.apache.felix</groupId>
 <artifactId>maven-bundle-plugin</artifactId>
 <version>1.4.0</version>
 <extensions>true</extensions>
 </plugin>
 </plugins>
 </build>

```

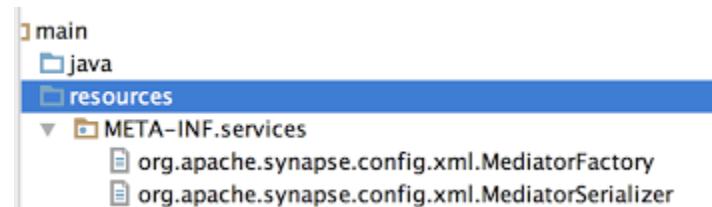
```
<configuration>
 <instructions>
 <Bundle-SymbolicName>org.test</Bundle-SymbolicName>
 <Bundle-Name>org.test</Bundle-Name>
 <Export-Package>
 org.test.mediator.*,
 </Export-Package>
 <Import-Package>
 *; resolution:=optional
 </Import-Package>
 <Fragment-Host>synapse-core</Fragment-Host>
 </instructions>
</configuration>
</plugin>
</plugins>
</build>

<dependencies>
 <dependency>
 <groupId>org.apache.synapse</groupId>
 <artifactId>synapse-core</artifactId>
 <version>2.1.1-wso2v5</version>
 </dependency>
```

```
</dependencies>
</project>
```

In this case, it is necessary to make the mediator an OSGI fragment of the synapse-core bundler. To achieve this, use the <Fragment-Host>synapse-core</Fragment-Host>.

Create <PROJECT\_HOME>/main/resources/META-INF.services/org.apache.synapse.config.xml.MediatorFactory and <PROJECT\_HOME>/main/resources/META-INF.services/org.apache.synapse.config.xml.MediatorSerializer files with the following content as shown below, to add service provider definitions to your Maven project.



#### Content of the org.apache.synapse.config.xml.MediatorFactory file

```
org.wso2.carbon.mediator.cache.config.xml.CacheMediatorFactory
```

#### Content of the org.apache.synapse.config.xml.MediatorSerializer file

```
org.wso2.carbon.mediator.cache.config.xml.CacheMediatorSerializer
```

After you create the mediator, place the JAR file in the <ESB\_HOME>\repository\components\dropins directory.

If you deploy a mediator through a Carbon application (CAR file), it can only be accessed from the sequences and proxy services within the same CAR file.

## Writing Custom Configuration Implementations for Mediators

You can write your own custom configurator for the Mediator implementation you write without relying on the Class Mediator or Spring extension for its initialization.

- **The MediatorFactory implementation** - Defines how to digest a custom XML configuration element to be used to create and configure the custom mediator instance.
- **The MediatorSerializer implementation** - Defines how a configuration should be serialized back into an XML configuration.

The custom MediatorFactory, MediatorSerializer implementations and the mediator class/es must be bundled as an OSGI bundle exporting these classes and placed into the ESB\_HOME/repository/components/dropins folder, so that the Synapse runtime could find and load the definition.

A custom JAR file must bundle your classes implementing the Mediator interface, MediatorSerializer and the MediatorFactory interfaces. It should also contain two text files named org.apache.synapse.config.xml.MediatorFactory and org.apache.synapse.config.xml.MediatorSerializer which will contain the fully qualified name(s) of your MediatorFactory and MediatorSerializer implementation classes. Any dependencies should be made available through OSGI bundles in the same plugins directory.

The MediatorFactory interface listing, which you should implement, is given below and its `getTagQName()` method must define the fully qualified element of interest for custom configuration. The Synapse initialization will call back to this MediatorFactory instance through the `createMediator(OMElement elem)` method passing in this XML element, so that an instance of the mediator could be created utilizing the custom XML specification and returned.

### [The MediatorFactory Interface](#)

```
package org.apache.synapse.config.xml;

import ...

/**
 * A mediator factory capable of creating an instance of a mediator through a given
 * XML should implement this interface
 */
public interface MediatorFactory {
 /**
 * Creates an instance of the mediator using the OMElement
 * @param elem
 * @return the created mediator
 */
 public Mediator createMediator(OMElement elem);

 /**
 * The QName of this mediator element in the XML config
 * @return QName of the mediator element
 */
 public QName getTagQName();
}
```

### [The MediatorSerializer Interface](#)

```

package org.apache.synapse.config.xml;

import ...

/**
 * Interface which should be implemented by mediator serializers. Does the
 * reverse of the MediatorFactory
 */
public interface MediatorSerializer {

 /**
 * Return the XML representation of this mediator
 * @param m mediator to be serialized
 * @param parent the OMEElement to which the serialization should be attached
 * @return the serialized mediator XML
 */
 public OMEElement serializeMediator(OMEElement parent, Mediator m);

 /**
 * Return the class name of the mediator which can be serialized
 * @return the class name
 */
 public String getMediatorClassName();
}

```

## Places for Putting Custom Mediators

There are three places for creating a mediator in WSO2 ESB. They are:

- <ESB\_HOME>\repository\components\extensions
- <ESB\_HOME>\repository\components\dropins
- <ESB\_HOME>\repository\components\lib

### *Extensions directory*

You can create a regular non-OSGI mediator JAR into this directory. The system will convert the mediator JAR into an OSGI JAR and deploy it into the server. This way is easy and simple to use. In this way user cannot specifically use the OSGI features for the mediator. For example, user cannot make certain packages private or import specific versions of packages. Also user can put mediators that are class mediator as well as mediators that have its own XML configuration.

**Tip:** The recommended way is to put the JAR into the `mediators` directory.

### *Dropins directory*

User need to build an OSGI bundle of the mediator and put it into the `dropins` directory. This requires a basic knowledge about the OSGI and maven bundle plug-in. So it can be difficult for a programmer who does not aware of the OSGI technologies. Since user have created the OSGI bundle he can use the OSGI features as he wishes. When creating the OSGI bundle? a mediator with a XML configuration should be a fragment of the synapse-core. If it is a class mediator, it can be a normal bundle.

### *Lib directory*

User can only put class mediator JARs into this directory. A regular JAR can be put into this directory and it will be automatically converted to an OSGI bundle by the system.

**Note:** In all these cases you have to restart the server after putting the mediator.

## Writing Custom Mediator Implementations

The following information concerning writing custom mediators implementations is available:

- [MessageContext Interface](#)
- [Mediator Interface](#)
- [Leaf and Node Mediators, List Mediators and Filter Mediators](#)

### **MessageContext Interface**

The [MessageContext](#) interface is the primary interface of the Synapse API. This essentially defines the per-message context passed through the chain of mediators, for each and every message received and processed by Synapse. Each message instance is wrapped within a [MessageContext](#) instance, and the message context is set with the references to the [SynapseConfiguration](#) and [SynapseEnvironment](#).

The [SynapseConfiguration](#) holds the global configuration model that defines mediation rules, local registry entries and other configuration.

The [SynapseEnvironment](#) gives access to the underlying SOAP implementation used - Apache Axis2.

A typical mediator would need to manipulate the [MessageContext](#) by referring to the [SynapseConfiguration](#).

### **Note**

It is strongly recommended that the [SynapseConfiguration](#) is not updated by mediator instances as it is shared by all messages and may be updated by Synapse administration or configuration modules.

Mediator instances may store local message properties into the [MessageContext](#) for later retrieval by successive mediators.

### **Tip**

Extending the [AbstractMediator](#) class is the easier way to write a new mediator rather than implementing the [Mediator](#) interface.

### **MessageContext Interface**

```
package org.apache.synapse;

import ...

public interface MessageContext {

 /**
 * Get a reference to the current SynapseConfiguration
 *
 * @return the current synapse configuration
 */
 public SynapseConfiguration getConfiguration();

 /**
 * Set or replace the Synapse Configuration instance to be used. May be used to

```

```

 * programmatically change the configuration at runtime etc.
 *
 * @param cfg The new synapse configuration instance
 */
public void setConfiguration(SynapseConfiguration cfg);

/**
 * Returns a reference to the host Synapse Environment
 * @return the Synapse Environment
 */
public SynapseEnvironment getEnvironment();

/**
 * Sets the SynapseEnvironment reference to this context
 * @param se the reference to the Synapse Environment
 */
public void setEnvironment(SynapseEnvironment se);

/**
 * Get the value of a custom (local) property set on the message instance
 * @param key key to look up property
 * @return value for the given key
 */
public Object getProperty(String key);

/**
 * Set a custom (local) property with the given name on the message instance
 * @param key key to be used
 * @param value value to be saved
 */
public void setProperty(String key, Object value);

/**
 * Returns the Set of keys over the properties on this message context
 * @return a Set of keys over message properties
 */
public Set getPropertyKeySet();

/**
 * Get the SOAP envelope of this message
 * @return the SOAP envelope of the message
 */
public SOAPEnvelope getEnvelope();

/**
 * Sets the given envelope as the current SOAPEnvelope for this message
 * @param envelope the envelope to be set
 * @throws org.apache.axis2.AxisFault on exception
 */
public void setEnvelope(SOAPEnvelope envelope) throws AxisFault;

/**
 * SOAP message related getters and setters
 */

```

```

 publicget/set()...
}

```

The `MessageContext` interface is based on the Axis2 `MessageContext` interface and uses the `EndpointReference` and `SOAPEnvelope` classes/interfaces. The purpose of this interface is to capture a message as it flows through the system. As you see the message payload is represented using the SOAP infoset. Binary messages can be embedded in the Envelope using MTOM (SOAP Message Transmission Optimization Mechanism) or SwA (SOAP with Attachments) using the AXIOM (AXIS Object Model) object model.

### **Mediator Interface**

The second key interface for mediator writers is the `Mediator` interface.

```

package org.apache.synapse;

import org.apache.synapse.MessageContext;

/**
 * All Synapse mediators must implement this Mediator interface. As a message passes
 * through the synapse system, each mediator's mediate() method is invoked in the
 * sequence/order defined in the SynapseConfiguration.
 */
public interface Mediator {

 /**
 * Invokes the mediator passing the current message for mediation. Each
 * mediator performs its mediation action, and returns true if mediation
 * should continue, or false if further mediation should be aborted.
 *
 * @param synCtx the current message for mediation
 * @return true if further mediation should continue
 */
 public boolean mediate(MessageContext synCtx);

 /**
 * This is used for debugging purposes and exposes the type of the current
 * mediator for logging and debugging purposes
 * @return a String representation of the mediator type
 */
 public String getType();

 /**
 * This is used to check whether the tracing should be enabled on the current
 * mediator or not
 * @return value that indicate whether tracing is on, off or unset
 */
 public int getTraceState();

 /**
 * This is used to set the value of tracing enable variable
 * @param traceState Set whether the tracing is enabled or not
 */
 public void setTraceState(int traceState);
}

```

A mediator can read and/or modify the message in any suitable manner - adjusting the routing headers or changing

the message body. If the `mediate()` method returns "false", it signals to the Synapse processing model to stop further processing of the message. For example, if the mediator is a security agent, it may decide that this message is dangerous and should not be processed further. This is generally the exception as mediators are usually designed to co-operate to process the message onwards.

#### ***Leaf and Node Mediators, List Mediators and Filter Mediators***

Mediators may be **Node mediators** (they can contain child mediators) or **Leaf mediators** (mediators that does not hold any other child mediators). A **Node mediator** must implement the `org.apache.synapse.mediators.ListMediator` interface listed below or extend from `org.apache.synapse.mediators.AbstractListMediator`.

#### ***The ListMediator Interface***

```

package org.apache.synapse.mediators;

import java.util.List;

/**
 * The List mediator executes a given sequence/list of child mediators
 */
public interface ListMediator extends Mediator {
 /**
 * Appends the specified mediator to the end of this mediator's (children) list
 * @param m the mediator to be added
 * @return true (as per the general contract of the Collection.add method)
 */
 public boolean addChild(Mediator m);

 /**
 * Appends all of the mediators in the specified collection to the end of this
 * mediator's (children)
 * list, in the order that they are returned by the specified collection's iterator
 * @param c the list of mediators to be added
 * @return true if this list changed as a result of the call
 */
 public boolean addAll(List c);

 /**
 * Returns the mediator at the specified position
 * @param pos index of mediator to return
 * @return the mediator at the specified position in this list
 */
 public Mediator getChild(int pos);

 /**
 * Removes the first occurrence in this list of the specified mediator
 * @param m mediator to be removed from this list, if present
 * @return true if this list contained the specified mediator
 */
 public boolean removeChild(Mediator m);

 /**
 * Removes the mediator at the specified position in this list
 * @param pos the index of the mediator to remove
 * @return the mediator previously at the specified position
 */
 public Mediator removeChild(int pos);

 /**
 * Return the list of mediators of this List mediator instance
 * @return the child/sub mediator list
 */
 public List getList();
}

```

A `ListMediator` implementation should call `super.mediate(synCtx)` to process its sub mediator sequence.

A `FilterMediator` is a `ListMediator` that executes its sequence of sub mediators on successful outcome of a test condition. The Mediator instance that performs filtering should implement the `FilterMediator` interface.

## FilterMediator Interface

```

package org.apache.synapse.mediators;

import org.apache.synapse.MessageContext;

/**
 * The filter mediator is a list mediator, which executes the given (sub) list of
mediators
 * if the specified condition is satisfied
 *
 * @see FilterMediator#test(org.apache.synapse.MessageContext)
 */
public interface FilterMediator extends ListMediator {

 /**
 * Should return true if the sub/child mediators should execute. i.e. if the
filter
 * condition is satisfied
 * @param synCtx
 * @return true if the configured filter condition evaluates to true
 */
 public boolean test(MessageContext synCtx);
}

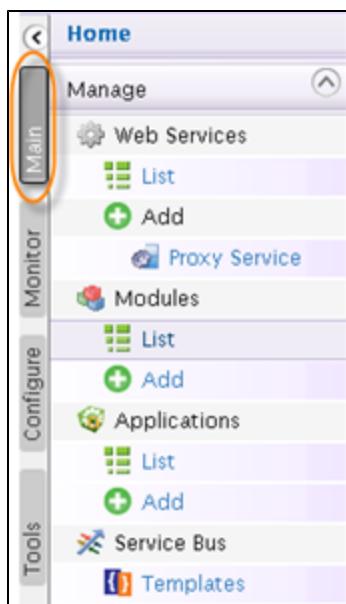
```

## Uploading Artifacts

ESB **artifacts** are custom developed extensions that can be deployed in the ESB. These extensions can contain custom **mediators**, **tasks**, configuration factories and serializers. ESB artifacts packed into .jar or .xar format could be hot-deployed in ESB using ESB Artifact Uploader feature.

Follow the instructions below to upload artifacts to the WSO2 ESB.

1. Sign in. Enter your user name and password to log on to the ESB Management Console.
2. Click on "Main" in the left menu to access the "Manage" menu.



3. In the "Manage" menu, click on "Add" under "ESB Artifacts."



4. In the "Add Artifacts" page, click on the "Browse" button to select file to upload.

**Add Artifacts**

ESB artifacts are custom developed extensions that can be deployed into the ESB. These extensions can contain custom mediators, tasks, configuration factories and serializers.

Upload Artifact

ESB Artifact .xar / jar   

5. Click on the "Upload" button to add an artifact to the ESB.

**Add Artifacts**

ESB artifacts are custom developed extensions that can be deployed into the ESB. These extensions can contain custom mediators, tasks, configuration factories and serializers.

Upload Artifact

ESB Artifact .xar / jar    

6. The XAR /JAR file appears in the "Artifacts" list.

**Deployed Artifacts**

Artifact Name	Action
my_mediator.jar <span style="border: 1px solid orange; border-radius: 50%; padding: 2px;"> </span>	<input type="button" value="Delete"/>

If you no longer want an artifact in your ESB instance, click **Delete** next to the artifact to remove it.

## Reference Guide

The following topics provide reference information for working with WSO2 ESB:

- Configuration Files
- Calling Admin Services from Apps
- Default Ports of WSO2 Products
- ESB Tools
- Properties Reference
- Synapse Configuration Reference
- Setting Up Host Names and Ports

## Configuration Files

The following sections explain the main configuration files of WSO2 ESB.

- XML files
- Properties Files

### XML files

This section explains the following xml configuration files used in carbon based products.

- Configuring axis2.xml
- Configuring carbon.xml

- Configuring catalina-server.xml
- Configuring config-validation.xml
- Configuring identity.xml
- Configuring master-datasources.xml
- Configuring registry.xml
- Configuring user-mgt.xml
- Configuring axis2\_blocking\_client.xml

#### Configuring axis2.xml

Users can change the default functionality-related configurations by editing the `<PRODUCT_HOME>/repository/conf/axis2/axis2.xml` file using the information given below. This information is provided as reference for users who are already familiar with the product features and want to know how to configure them. If you need introductory information on a specific concept, such as message receivers and formatters, see the relevant topics in the User Guide.

Click on the table and use the left and right arrow keys to scroll horizontally. For sample values, see the Example below the table.

#### **XML Elements**

XML element	Attributes	Description	Data type	Default value	Mandatory/Optional
<code>&lt;axisconfig&gt;</code>	<code>name</code>	The root element. The name is defined as: <code>name= "AxisJava2.0"</code>			Mandatory
<code>&lt;module&gt;</code>	<code>ref</code>	A globally engaged module. The <code>ref</code> attribute specifies the module name.			Mandatory
<code>&lt;parameter&gt;</code>	<code>name</code> <code>locked</code>	A parameter is a name-value pair. All top-level parameters (those that are direct sub-elements of the root element) will be transformed into properties in AxisConfiguration and can be accessed in the running system. The <code>name</code> attribute (required) specifies the parameter name. If you set the <code>locked</code> attribute to true (default is false), this parameter's value cannot be overridden by services and other configurations.			Mandatory

<listener>	class	<p>A registered listener that will be automatically informed whenever a change occurs in AxisConfiguration, such as when a service or module is deployed or removed. The <code>class</code> attribute specifies this listener's implementation class, which must implement the <code>AxisObserver</code> interface.</p> <p>Registering listeners is useful for additional features such as RSS feed generation, which will provide service information to subscribers.</p>		Optional
<messageReceivers>		The container element for messages receiver definitions.		Mandatory
<messageReceiver>	class mep	A message receiver definition. The <code>class</code> attribute (required) specifies the message receiver implementation class. The <code>mep</code> attribute (required) specifies the message exchange pattern supported by this message receiver. Each message receiver definition supports only one MEP.		Mandatory
<messageFormatters>		The container element for message formatter definitions, which are used to serialize outgoing messages to different formats (such as JSON). The format for a message can be specified by setting the "messageType" property in the <code>MessageContext</code> . It can also be specified as a parameter in <code>service.xml</code> (for service-based configuration) in addition to <code>axis2.xml</code> (for global configuration).		Optional

<messageFormatter>	contentType class	A message formatter definition. The <code>contentType</code> attribute specifies which message types are handled by this formatter, and the <code>class</code> attribute specifies the formatter implementation class.			Optional
<messageBuilders>		The container element for the message builder definitions, which are used to process the raw payload of incoming messages and convert them to SOAP.			Optional
<messageBuilder>	contentType class	A message builder definition. The <code>contentType</code> attribute specifies which message types are handled by this builder, and the <code>class</code> attribute specifies the builder implementation class.			Optional
<transportReceiver>	name class	A transport receiver definition, one for each transport type. The <code>name</code> attribute specifies the short name to use when referring to this transport in your configurations (http, tcp, etc.), and the <code>class</code> attribute specifies the receiver implementation class that provides the logic for receiving messages via this transport. You can specify <code>&lt;parameter&gt;</code> elements to pass any necessary information to the transport.			Mandatory
<transportSender>		Just like <code>&lt;transportReceiver&gt;</code> , except <code>&lt;transportSender&gt;</code> allows you to define transport senders, which are used to send messages via the transport.			Mandatory

<phaseOrder>	type	<p>Specifies the order of phases in the execution chain of a specific type of flow (specified by the <code>type</code> attribute), which can be one of the following:</p> <ul style="list-style-type: none"> <li>• InFlow</li> <li>• OutFlow</li> <li>• InFaultFlow</li> <li>• OutFaultFlow</li> </ul> <p>You add phases using the <code>&lt;phase&gt;</code> sub-element. In the In phase orders, all phases before the Dispatch phase are global phases and after Dispatch are operation phases. In the Out phase orders, phases before the MessageOut phase are global phases and after MessageOut are operation phases.</p>			Mandatory
<phase>	name	The phase definition. The <code>name</code> attribute specifies the phase name. You can add the <code>&lt;handler&gt;</code> sub-element to execute a specific handler during this phase.			Mandatory
<handler>	name class	<p>The handler (message processing functionality) to execute during this phase. Handlers are combined into chains and phases to provide customizable functionality such as security, reliability, etc. Handlers must be multi-thread safe and should keep all their state in Context objects (see the <code>org.apache.axis2.context</code> package).</p>			Optional
<order>	phase				Optional
<clustering>	class enable	<p>Used to enable clustering. The <code>class</code> attribute specifies the clustering agent class. The <code>enable</code> attribute is false by default; set it to true to enable clustering.</p>			Optional
<property>					Optional

	<b>name</b>			
	<b>value</b>			
<members>		The list of static or well-known members. These entries will only be valid if the "membershipScheme" above is set to "wka"	N/A	Optional
<member>			N/A	Optional
<hostName>			N/A	Optional
<port>			N/A	Optional
<groupManagement>		Enable the groupManagement entry if you need to run this node as a cluster manager. Multiple application domains with different GroupManagementAgent implementations can be defined in this section.		Optional
	<b>enable</b>		FALSE	
<applicationDomain >			N/A	Optional
	<b>name</b>			
	<b>port</b>			
	<b>subDomain</b>			
	<b>agent</b>			
	<b>description</b>			

### Example

The following example shows excerpts from an axis2.xml file.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
...
<axisconfig name="AxisJava2.0">

 <!-- ===== -->
 <!-- Parameters -->
 <!-- ===== -->

 ...

 <!-- If you want to enable file caching for attachments change this to true -->
 <parameter name="cacheAttachments" locked="false">false</parameter>
 <!-- Attachment file caching location relative to CARBON_HOME -->

```

```

<parameter name="attachmentDIR" locked="false">work/mtom</parameter>
<!-- Attachment file cache threshold size -->
<parameter name="sizeThreshold" locked="false">4000</parameter>

...

<!-- ===== -->
<!-- Listeners -->
<!-- ===== -->

<!-- This deployment interceptor will be called whenever before a module is
initialized or -->
<!-- service is deployed -->
<listener class="org.wso2.carbon.core.deployment.DeploymentInterceptor"/>

<!-- ===== -->
<!-- Deployers -->
<!-- ===== -->

<!-- Deployer for the dataservice. -->
<!--<deployer extension="dbs" directory="dataservices"
class="org.wso2.dataservices.DBDeployer"/>-->

<!-- Axis1 deployer for Axis2 -->
<!--<deployer extension="wsdd" class="org.wso2.carbon.axis1services.Axis1Deployer"
directory="axis1services"/>-->

...

<!-- ===== -->
<!-- Message Receivers -->
<!-- ===== -->

<!-- This is the set of default Message Receivers for the system, if you want to
have -->
<!-- message receivers for any of the other Message exchange Patterns (MEP)
implement it -->
<!-- and add the implementation class to here, so that you can refer from any
operation -->
<!-- Note : You can override this for particular service by adding this same
element to the -->
<!-- services.xml with your preferences -->
<messageReceivers>
 <messageReceiver mep="http://www.w3.org/ns/wsdl/in-only"
class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver"/>
 <messageReceiver mep="http://www.w3.org/ns/wsdl/robust-in-only"
class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver"/>
 <messageReceiver mep="http://www.w3.org/ns/wsdl/in-out"
class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
</messageReceivers>

<!-- ===== -->
<!-- Message Formatters -->
<!-- ===== -->

<!-- Following content type to message formatter mapping can be used to implement
support -->

```

```

<!-- for different message format serializations in Axis2. These message formats
are -->
<!-- expected to be resolved based on the content type. -->
<messageFormatters>
 <messageFormatter contentType="application/x-www-form-urlencoded"
 class="org.apache.axis2.transport.http.XFormURLEncodedFormatter"/>
 <messageFormatter contentType="multipart/form-data"
 class="org.apache.axis2.transport.http.MultipartFormDataFormatter"/>
 <messageFormatter contentType="application/xml"
 class="org.apache.axis2.transport.http.ApplicationXMLFormatter"/>
 <messageFormatter contentType="text/xml"
 class="org.apache.axis2.transport.http.SOAPMessageFormatter"/>
 <messageFormatter contentType="application/soap+xml"
 class="org.apache.axis2.transport.http.SOAPMessageFormatter"/>
 <messageFormatter contentType="text/plain"
 class="org.apache.axis2.format.PlainTextFormatter" />

 ...
</messageFormatters>

<!-- ===== -->
<!-- Message Builders -->
<!-- ===== -->

<!-- Following content type to builder mapping can be used to implement support
for -->
<!-- different message formats in Axis2. These message formats are expected to be
-->
<!-- resolved based on the content type. -->
<messageBuilders>
 <messageBuilder contentType="application/xml"
 class="org.apache.axis2.builder.ApplicationXMLBuilder"/>
 <messageBuilder contentType="application/x-www-form-urlencoded"
 class="org.apache.synapse.commons.builders.XFormURLEncodedBuilder"/>
 <messageBuilder contentType="multipart/form-data"
 class="org.apache.axis2.builder.MultipartFormDataBuilder"/>
 <messageBuilder contentType="text/plain"
 class="org.apache.axis2.format.PlainTextBuilder" />

 ...
</messageBuilders>

<!-- ===== -->
<!-- Transport Ins (Listeners) -->
<!-- ===== -->

 <transportReceiver name="http"
class="org.apache.synapse.transport.passthru.PassThroughHttpListener">
 <parameter name="port" locked="false">8280</parameter>
 <parameter name="non-blocking" locked="false">true</parameter>
 <!--parameter name="bind-address" locked="false">hostname or IP

```

```

address</parameter-->
 <!--parameter name="WSDLEPRPrefix"
locked="false">https://apachehost:port/somepath</parameter-->
 <parameter name="httpGetProcessor"
locked="false">org.wso2.carbon.transport.nhttp.api.PassThroughNHttpGetProcessor</param
eter>
 <!--<parameter name="priorityConfigFile" locked="false">location of priority
configuration file</parameter>-->
 </transportReceiver>

 ...

<!-- ===== -->
<!-- Transport Outs (Senders) -->
<!-- ===== -->

<transportSender name="http"
class="org.apache.synapse.transport.passthru.PassThroughHttpSender">
 <parameter name="non-blocking" locked="false">true</parameter>
 <!--<parameter name="warnOnHTTP500" locked="false">*</parameter>-->
 <!--parameter name="http.proxyHost" locked="false">localhost</parameter-->
 <!--<parameter name="http.proxyPort" locked="false">3128</parameter>-->
 <!--<parameter name="http.nonProxyHosts"
locked="false">localhost|moon|sun</parameter>-->
</transportSender>

 ...

<!-- ===== -->
<!-- Global Engaged Modules -->
<!-- ===== -->

<!-- Comment this out to disable Addressing -->
<module ref="addressing"/>

<!-- ===== -->
<!-- Clustering -->
<!-- ===== -->
<!--
 To enable clustering for this node, set the value of "enable" attribute of the
"class"
 element to "true". The initialization of a node in the cluster is handled by the
class
 corresponding to the "class" attribute of the "clustering" element. It is also
responsible for
 getting this node to join the cluster.
-->
<clustering
class="org.wso2.carbon.core.clustering.hazelcast.HazelcastClusteringAgent"
 enable="false">

 <!--
 This parameter indicates whether the cluster has to be automatically
initialized
 when the AxisConfiguration is built. If set to "true" the initialization
will not be
 done at that stage, and some other party will have to explicitly initialize
the cluster.
-->

```

```

<parameter name="AvoidInitiation">true</parameter>

...

<!--
 The list of static or well-known members. These entries will only be valid
if the
 "membershipScheme" above is set to "wka"
-->
<members>
 <member>
 <hostName>127.0.0.1</hostName>
 <port>4000</port>
 </member>
</members>

<!--
 Enable the groupManagement entry if you need to run this node as a cluster
manager.
 Multiple application domains with different GroupManagementAgent
implementations
 can be defined in this section.
-->
<groupManagement enable="false">
 <applicationDomain name="wso2.esb.domain"
 description="ESB group"

agent="org.wso2.carbon.core.clustering.hazelcast.HazelcastGroupManagementAgent"
 subDomain="worker"
 port="2222"/>
 </groupManagement>
</clustering>

<!-- ===== -->
<!-- Transactions -->
<!-- ===== -->

<!--
 Uncomment and configure the following section to enable transactions support
-->
<!--<transaction timeout="30000">
 <parameter
name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFact
ory</parameter>
 <parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
 <parameter name="UserTransactionJNDIName">UserTransaction</parameter>
 <parameter name="TransactionManagerJNDIName">TransactionManager</parameter>
</transaction>-->

<!-- ===== -->
<!-- Phases -->
<!-- ===== -->

<phaseOrder type="InFlow">
 <!-- System pre defined phases -->
 <!--
 The MsgInObservation phase is used to observe messages as soon as they are
 received. In this phase, we could do some things such as SOAP message
tracing & keeping

```

```

track of the time at which a particular message was received

NOTE: This should be the very first phase in this flow
-->
<phase name="MsgInObservation">
 <handler name="TraceMessageBuilderDispatchHandler"
 class="org.apache.synapse.transport.passthru.util.TraceMessageBuilderDispatchHandler"/>
</phase>
<phase name="Validation"/>
<phase name="Transport">
 <handler name="RequestURIBasedDispatcher"
 class="org.apache.axis2.dispatchers.RequestURIBasedDispatcher">
 <order phase="Transport"/>
 </handler>
 <handler name="CarbonContextConfigurator"
 class="org.wso2.carbon.mediation.initializer.handler.CarbonContextConfigurator"/>
 <handler name="RelaySecurityMessageBuilderDispatchHandler"
 class="org.apache.synapse.transport.passthru.util.RelaySecurityMessageBuilderDispatchHandler"/>
 <handler name="SOAPActionBasedDispatcher"
 class="org.apache.axis2.dispatchers.SOAPActionBasedDispatcher">
 <order phase="Transport"/>
 </handler>
 <!--handler name="SMTPFaultHandler"
 class="org.wso2.carbon.core.transports.smtp.SMTPFaultHandler">
 <order phase="Transport"/>
 </handler-->
 <handler name="CacheMessageBuilderDispatchHandler"
 class="org.wso2.carbon.mediation.initializer.handler.CacheMessageBuilderDispatchHandler"/>
 </phase>
 ...
 </phaseOrder>

<phaseOrder type="OutFlow">
 <!-- Handlers related to unified-endpoint component are added to the UEPPhase
-->
 <phase name="UEPPhase" />
 <!-- user can add his own phases to this area -->
 <phase name="RMPhase" />
 ...
</phaseOrder>

```

```

...
</axisconfig>

```

### Configuring carbon.xml

Users can change the configurations related to the default Carbon functionality by editing the `<PRODUCT_HOME>/repository/conf/carbon.xml` file using the information given below.

Click on the table and use the left and right arrow keys to scroll horizontally.

#### **XML Elements**

XML element	Attribute	Description	Data type
<code>&lt;Server&gt;</code>			
	<code>xmlns</code>		
<code>&lt;Name&gt;</code>		Product Name.	String
<code>&lt;ServerKey&gt;</code>		Machine readable unique key to identify each product.	String
<code>&lt;Version&gt;</code>		Product Version.	String
<code>&lt;HostName&gt;</code>		Host name or IP address of the machine hosting this server e.g. www.wso2.org, 192.168.1.10 This is will become part of the End Point Reference of the services deployed on this server instance.	String
<code>&lt;MgtHostName&gt;</code>		Host name to be used for the Carbon management console.	String
<code>&lt;ServerURL&gt;</code>		The URL of the back end server. This is where the admin services are hosted and will be used by the clients in the front end server. This is required only for the Front-end server. This is used when separating the BE server from the FE server.	String
<code>&lt;IndexPageURL&gt;</code>		The URL of the index page. This is where the user will be redirected after signing in to the carbon server.	String
<code>&lt;ServerRoles&gt;</code>		For cApp deployment, we have to identify the roles that can be acted by the current server. The following property is used for that purpose. Any number of roles can be defined here. Regular expressions can be used in the role. Ex : <code>&lt;Role&gt;.*&lt;/Role&gt;</code> means this server can act as any role.	String
<code>&lt;Role&gt;</code>			
<code>&lt;BamServerURL&gt;</code>			
<code>&lt;Package&gt;</code>		The fully qualified name of the server.	String
<code>&lt;WebContextRoot&gt;</code>		Webapp context root of WSO2 Carbon.	String

<RegistryHttpPort>		In-order to get the registry http Port from the back-end when the default http transport is not the same.	Int
<ItemsPerPage>		Number of items to be displayed on a management console page. This is used at the backend server for pagination of various items.	Int
<InstanceMgtWSEndpoint>		The endpoint URL of the cloud instance management Web service.	String
<Ports>		Ports used by this server	
<Offset>		Ports offset. This entry will set the value of the ports defined below to the define value + Offset. e.g. Offset=2 and HTTPS port=9443 will set the effective HTTPS port to 9445.	Int
<JMX>		The JMX Ports.	Int
<RMIRegistryPort>		The port RMI registry is exposed.	Int
<RMIServerPort>		The port RMI server should be exposed.	Int
<EmbeddedLDAP>		Embedded LDAP server specific ports.	Int
<LDAPServerPort>		Port which embedded LDAP server runs.	Int
<KDCServerPort>		Port which KDC (Kerberos Key Distribution Center) server runs.	Int
<EmbeddedQpid>		Embedded Qpid broker ports.	Int
<BrokerPort>		Broker TCP Port.	Int
<BrokerSSLPort>		SSL Port.	Int
<JNDIProviderPort>		Override datasources JNDIproviderPort defined in bps.xml and datasources.properties files.	
<ThriftEntitlementReceivePort>		Override receive port of thrift based entitlement service.	Int
<JNDI>		JNDI Configuration.	Int
<DefaultInitialContextFactory>		The fully qualified name of the default initial context factory.	String
<Restrictions>		The restrictions that are done to various JNDI Contexts in a Multi-tenant environment.	
<AllTenants>		Contexts that are common to all tenants.	String
<UrlContexts>			
<UrlContext>			
<Scheme>			
<SuperTenantOnly>		Contexts that will be available only to the super-tenant.	String
<UrlContexts>			
<UrlContext>			
<Scheme>			

<IsCloudDeployment>		Property to determine if the server is running on a cloud deployment environment. This property should only be used to determine deployment specific details that are applicable only in a cloud deployment, i.e when the server is deployed *-as-a-service.	Bool
<EnableMetering>		Property to determine whether usage data should be collected for metering purposes.	Bool
<MaxThreadExecutionTime>		The Max time a thread should take for execution in seconds.	Int
<GhostDeployment>		A flag to enable or disable Ghost Deployer. By default this is set to false. That is because the Ghost Deployer works only with the HTTP/S transports. If you are using other transports, don't enable Ghost Deployer.	Bool
<Enabled>		When <GhostDeployment> is enabled, the lazy loading feature will apply to artifacts deployed. That is, when a tenant loads, only the specific artifact requested by the service will be loaded.	
<PartialUpdate>		<PartialUpdate> is a further enhancement to lazy loading of artifacts, which applies when <DeploymentSynchronizer> is enabled in a <a href="#">clustered</a> environment.	
<Axis2Config>		Axis2 related configurations.	
<RepositoryLocation>		Location of the Axis2 Services & Modules repository This can be a directory in the local file system, or a URL. e.g. 1. /home/wso2wsas/repository/ - An absolute path 2. repository - In this case, the path is relative to CARBON_HOME 3. file:///home/wso2wsas/repository/ 4. http://wso2wsas/repository/.	String
<DeploymentUpdateInterval>		Deployment update interval in seconds. This is the interval between repository listener executions.	Int
<ConfigurationFile>		Location of the main Axis2 configuration descriptor file, a.k.a. axis2.xml file This can be a file on the local file system, or a URL e.g. 1. /home/repository/axis2.xml - An absolute path 2. conf/axis2.xml - In this case, the path is relative to CARBON_HOME 3. file:///home/carbon/repository/axis2.xml 4. http://repository/conf/axis2.xml	String
<ServiceGroupContextIdleTime>		ServiceGroupContextIdleTime, which will be set in ConfigurationContex for multiple clients which are going to access the same ServiceGroupContext Default Value is 30 Sec.	String
<ClientRepositoryLocation>		This repository location is used to crete the client side configuration context used by the server when calling admin services.	String
<clientAxis2XmlLocation>		This axis2 xml is used in creating the configuration context by the FE server calling to BE server.	String

<HideAdminServiceWSDLs>		If this parameter is set, the WSDL file on an admin service will not give the admin service WSDL. By default, this parameter is set to "true". Note that setting this parameter to false will expose WSO2 Storage Server operations through a WSDL.	String
<HttpAdminServices>		WARNING-Use With Care! Uncommenting bellow parameter would expose all AdminServices in HTTP transport. With HTTP transport your credentials and data routed in public channels are vulnerable for sniffing attacks. Use this parameter ONLY if your communication channels are confirmed to be secured by other means.	String
<ServiceUserRoles>		The default user roles which will be created when the server is started up for the first time.	String
<EnableEmailUserName>		Enable following config to allow Emails as usernames.	Boolean
<Security>		Security configurations.	
<KeyStore>		KeyStore which will be used for encrypting/decrypting passwords and other sensitive information.	String
<Location>		Keystore file location.	String
<Type>		Keystore type (JKS/PKCS12 etc.)	String
<Password>		Keystore password.	String
<KeyAlias>		Private Key alias.	String
<KeyPassword>		Private Key password.	String
<TrustStore>		System wide trust-store which is used to maintain the certificates of all the trusted parties.	String
<Location>		Trust-store file location.	String
<Type>		Trust-store type.	String
<Password>		Trust-store password.	String
<NetworkAuthenticatorConfig>		The Authenticator configuration to be used at the JVM level. We extend the java.net.Authenticator to make it possible to authenticate to given servers and proxies.	String
<Credential>			String
<Pattern>		The pattern that would match a subset of URLs for which this authenticator would be used.	String
<Type>		The type of this authenticator. Allowed values are: 1. server 2. proxy.	String
<Username>		The username used to log in to server/proxy.	String
<Password>		The password used to log in to server/proxy.	String

<TomcatRealm>		The Tomcat realm to be used for hosted Web applications. Allowed values are; 1. UserManager 2. Memory If this is set to 'UserManager', the realm will pick users & roles from the system's WSO2 User Manager. If it is set to 'memory', the realm will pick users & roles from CARBON_HOME/repository/conf/tomcat/tomcat-users.xml.	String
<DisableTokenStore>		Option to disable storing of tokens issued by STS.	Boolean
<TokenStoreClassName>		Security token store class name. If this is not set, default class will be org.wso2.carbon.security.util.SecurityTokenStore	String
<WorkDirectory>		The temporary work directory.	String
<HouseKeeping>		House-keeping configuration.	String
<AutoStart>		True - Start House-keeping thread on server startup false - Do not start House-keeping thread on server startup. The user will run it manually as and when he wishes.	Boolean
<Interval>		The interval in *minutes*, between house-keeping runs.	Integer
<MaxTempFileLifetime>		The maximum time in *minutes*, temp files are allowed to live in the system. Files/directories which were modified more than "MaxTempFileLifetime" minutes ago will be removed by the house-keeping task.	Integer
<FileUploadConfig>		Configuration for handling different types of file upload and other file uploading related config parameters. To map all actions to a particular FileUploadExecutor, use <Action>*</Action>.	String
<TotalFileSizeLimit>		The total file upload size limit in MB.	Integer
<Mapping>			String
<Actions>			String
<Action>			String
<Class>			String
<HttpGetRequestProcessors>		Processors which process special HTTP GET requests such as ?wsdl, ?policy etc. In order to plug in a processor to handle a special request, simply add an entry to this section. The value of the Item element is the first parameter in the query string(e.g. ?wsdl) which needs special processing The value of the Class element is a class which implements org.wso2.carbon.transport.HttpGetRequestProcessor	String
<Processor>			String
<Item>			String
<Class>			String

<DeploymentSynchronizer>		Deployment Synchronizer Configuration. Enabled when running with "svn based" dep sync. In master nodes you need to set both AutoCommit and AutoCheckout to true and in worker nodes set only AutoCheckout to true.	String
<Enabled>			Boolean
<AutoCommit>			Boolean
<AutoCheckout>			Boolean
<RepositoryType>			String
<SvnUrl>			String
<SvnUser>			String
<SvnPassword>			String
<SvnUrlAppendTenantId>			Boolean
<MediationConfig> <LoadFromRegistry> <SaveToFile> <Persistence> <RegistryPersistence>		Mediation persistence configurations. Only valid if mediation features are available i.e. ESB.	String
<ServerInitializers>		Server initializing code, specified as implementation classes of org.wso2.carbon.core.ServerInitializer. This code will be run when the Carbon server is initialized.	String
<Initializers>			
<RequireCarbonServlet>		Indicates whether the Carbon Servlet is required by the system, and whether it should be registered.	Boolean
<H2DatabaseConfiguration> <Property name>		Carbon H2 OSGI Configuration By default non of the servers start. name="web" - Start the web server with the H2 Console name="webPort" - The port (default: 8082) name="webAllowOthers" - Allow other computers to connect name="webSSL" - Use encrypted (HTTPS) connections name="tcp" - Start the TCP server name="tcpPort" - The port (default: 9092) name="tcpAllowOthers" - Allow other computers to connect name="tcpSSL" - Use encrypted (SSL) connections name="pg" - Start the PG server name="pgPort" - The port (default: 5435) name="pgAllowOthers" - Allow other computers to connect name="trace" - Print additional trace information; for all servers name="baseDir" - The base directory for H2 databases; for all servers.	String
<StatisticsReporterDisabled>		Disables the statistics reporter by default.	String
<EnableHTTPAdminConsole>		Enables HTTP for WSO2 servers so that you can access the Admin Console via HTTP.	String
<FeatureRepository>		Default Feature Repository of WSO2 Carbon.	String
<RepositoryName>			String

<RepositoryURL>			String
<APIManagement>		Configure API Management.	String
<Enabled>		Uses the embedded API Manager by default. If you want to use an external API Manager instance to manage APIs, configure below externalAPIManager.	Boolean
<ExternalAPIMangers> <APIGatewayURL> <APIPublisherURL>		Uncomment and configure API Gateway and Publisher URLs to use external API Manager instance.	String
<LoadAPIContextsInServerStartup>			Boolean

### Configuring catalina-server.xml

Users can change the default configurations by editing the <PRODUCT\_HOME>/repository/conf/tomcat/catalina-server.xml file using the information given below.

Click on the table and use the left and right arrow keys to scroll horizontally.

### XML Elements

XML element	Attribute	Description	Data type	Default value
<Server>		A Server element represents the entire Catalina servlet container. Therefore, it must be the single outermost element in the conf/server.xml configuration file. Its attributes represent the characteristics of the servlet container as a whole.		
	shutdown	The command string that must be received via a TCP/IP connection to the specified port number, in order to shut down Tomcat.	String	SHUTDOWN

	port	<p>The TCP/IP port number on which this server waits for a shutdown command. Set to -1 to disable the shutdown port.</p> <p>Note: Disabling the shutdown port works well when Tomcat is started using Apache Commons Daemon (running as a service on Windows or with jsvc on un*xes). It cannot be used when running Tomcat with the standard shell scripts though, as it will prevent shutdown.bat .sh and catalina.bat .sh from stopping it gracefully.</p>	Int	8005
<Service>		A Service element represents the combination of one or more Connector components that share a single Engine component for processing incoming requests. One or more Service elements may be nested inside a Server element.		
	name	The display name of this Service, which will be included in log messages if you utilize standard Catalina components. The name of each Service that is associated with a particular Server must be unique.	String	Catalina
	className	Java class name of the implementation to use. This class must implement the org.apache.catalina.Service interface. If no class name is specified, the standard implementation will be used.	String	org.wso2.carbon.tomca
<Connect or>				

	port	The TCP port number on which this Connector will create a server socket and await incoming connections. Your operating system will allow only one server application to listen to a particular port number on a particular IP address. If the special value of 0 (zero) is used, then Tomcat will select a free port at random to use for this connector. This is typically only useful in embedded and testing applications.	Int	9763
	URIEncoding	This specifies the character encoding used to decode the URI bytes, after %xx decoding the URL.	Int	UTF-8
	compressableMimeType	The value is a comma separated list of MIME types for which HTTP compression may be used.	String	text/html,text/javascript,
	noCompressionUserAgents	The value is a regular expression (using java.util.regex) matching the user-agent header of HTTP clients for which compression should not be used, because these clients, although they do advertise support for the feature, have a broken implementation.	String	ozilla, traviata
	compressionMinSize	If compression is set to "on" then this attribute may be used to specify the minimum amount of data before the output is compressed.	Int	2048

	compression	<p>The Connector may use HTTP/1.1 GZIP compression in an attempt to save server bandwidth. The acceptable values for the parameter is "off" (disable compression), "on" (allow compression, which causes text data to be compressed), "force" (forces compression in all cases), or a numerical integer value (which is equivalent to "on", but specifies the minimum amount of data before the output is compressed). If the content-length is not known and compression is set to "on" or more aggressive, the output will also be compressed. If not specified, this attribute is set to "off".</p> <p>Note: There is a tradeoff between using compression (saving your bandwidth) and using the sendfile feature (saving your CPU cycles). If the connector supports the sendfile feature, e.g. the NIO connector, using sendfile will take precedence over compression. The symptoms will be that static files greater than 48 Kb will be sent uncompressed. You can turn off sendfile by setting useSendfile attribute of the connector, as documented below, or change the sendfile usage threshold in the configuration of the DefaultServlet in the default conf/web.xml or in the web.xml of your web application.</p>	String	on
	server	Overrides the Server header for the http response. If set, the value for this attribute overrides the Tomcat default and any Server header set by a web application. If not set, any value specified by the application is used. Most often, this feature is not required.	String	WSO2 Carbon Server
	acceptCount	The maximum queue length for incoming connection requests when all possible request processing threads are in use. Any requests received when the queue is full will be refused.	Int	200

	maxKeepAliveRequests	The maximum number of HTTP requests which can be pipelined until the connection is closed by the server. Setting this attribute to 1 will disable HTTP/1.0 keep-alive, as well as HTTP/1.1 keep-alive and pipelining. Setting this to -1 will allow an unlimited amount of pipelined or keep-alive HTTP requests.	Int	200
	connectionUploadTimeout	Specifies the timeout, in milliseconds, to use while a data upload is in progress. This only takes effect if disableUploadTimeout is set to false.	Int	120000
	disableUploadTimeout	This flag allows the servlet container to use a different, usually longer connection timeout during data upload.	Boolean	false
	minSpareThreads	The minimum number of threads always kept running.	Int	50
	maxThreads	The maximum number of request processing threads to be created by this Connector, which therefore determines the maximum number of simultaneous requests that can be handled. If an executor is associated with this connector, this attribute is ignored as the connector will execute tasks using the executor rather than an internal thread pool.	Int	250
	acceptorThreadCount	The number of threads to be used to accept connections. Increase this value on a multi CPU machine, although you would never really need more than 2. Also, with a lot of non keep alive connections, you might want to increase this value as well.	Int	2
	maxHttpHeaderSize	The maximum size of the request and response HTTP header, specified in bytes.	Int	8192
	bindOnInit	Controls when the socket used by the connector is bound. By default it is bound when the connector is initiated and unbound when the connector is destroyed. If set to false, the socket will be bound when the connector is started and unbound when it is stopped.	Boolean	false

	redirectPort	If this Connector is supporting non-SSL requests, and a request is received for which a matching <security-constraint> requires SSL transport, Catalina will automatically redirect the request to the port number specified here.	Int	9443
	protocol	Sets the protocol to handle incoming traffic.	String	org.apache.coyote.http'
	SSLEnabled	Use this attribute to enable SSL traffic on a connector. To turn on SSL handshake/encryption/decryption on a connector set this value to true. The default value is false. When turning this value to true you will want to set the scheme and the secure attributes as well to pass the correct request.getScheme() and request.isSecure() values to the servlets See SSL Support for more information.	Boolean	true
	secure	Set this attribute to true if you wish to have calls to request.isSecure() to return true for requests received by this Connector. You would want this on an SSL Connector or a non SSL connector that is receiving data from a SSL accelerator, like a crypto card, a SSL appliance or even a webserver.	Boolean	true
	scheme	Set this attribute to the name of the protocol you wish to have returned by calls to request.getScheme(). For example, you would set this attribute to "https" for an SSL Connector.	String	https
	clientAuth	Set to true if you want the SSL stack to require a valid certificate chain from the client before accepting a connection. Set to false if you want the SSL stack to request a client Certificate, but not fail if one isn't presented. A false value will not require a certificate chain unless the client requests a resource protected by a security constraint that uses CLIENT-CERT authentication.	Boolean	false

	enableLookups	Set to true if you want calls to request.getRemoteHost() to perform DNS lookups in order to return the actual host name of the remote client. Set to false to skip the DNS lookup and return the IP address in String form instead (thereby improving performance). By default, DNS lookups are disabled.	Boolean	false
	sslProtocol	The SSL protocol(s) to use (a single value may enable multiple protocols - see the JVM documentation for details). The permitted values may be obtained from the JVM documentation for the allowed values for algorithm when creating an SSLContext instance e.g. Oracle Java 6 and Oracle Java 7.  Note: There is overlap between this attribute and sslEnabledProtocols.	String	TLS
	keystoreFile keystorePass	This setting allows you to use separate keystore and security certificates for SSL connections. The location of the keystore file and the keystore password can be given for these parameters. Note that by default, these parameters point to the location and password of the default keystore in the Carbon server.		
<Engine>		The Engine element represents the entire request processing machinery associated with a particular Catalina Service. It receives and processes all requests from one or more Connectors, and returns the completed response to the Connector for ultimate transmission back to the client. Exactly one Engine element MUST be nested inside a Service element, following all of the corresponding Connector elements associated with this Service.		
	name	Logical name of this Engine, used in log and error messages. When using multiple Service elements in the same Server, each Engine MUST be assigned a unique name.	String	Catalina

	defaultHost	The default host name, which identifies the Host that will process requests directed to host names on this server, but which are not configured in this configuration file. This name MUST match the name attributes of one of the Host elements nested immediately inside.	String	localhost
<Realm>		A Realm element represents a "database" of usernames, passwords, and roles (similar to Unix groups) assigned to those users. Different implementations of Realm allow Catalina to be integrated into environments where such authentication information is already being created and maintained, and then utilize that information to implement Container Managed Security as described in the Servlet Specification. You may nest a Realm inside any Catalina container Engine, Host, or Context). In addition, Realms associated with an Engine or a Host are automatically inherited by lower-level containers, unless explicitly overridden.		
	className	Java class name of the implementation to use. This class must implement the org.apache.catalina.Realminterface.	String	org.wso2.carbon.tomca

<Host>	<p>The Host element represents a virtual host, which is an association of a network name for a server (such as "www.mycompany.com" with the particular server on which Tomcat is running. For clients to be able to connect to a Tomcat server using its network name, this name must be registered in the Domain Name Service (DNS) server that manages the Internet domain you belong to - contact your Network Administrator for more information.</p> <p>In many cases, System Administrators wish to associate more than one network name (such as www.mycompany.com and company.com) with the same virtual host and applications. This can be accomplished using the Host Name Aliases feature discussed below.</p> <p>One or more Host elements are nested inside an Engine element. Inside the Host element, you can nest Context elements for the web applications associated with this virtual host. Exactly one of the Hosts associated with each Engine MUST have a name matching the defaultHost attribute of that Engine.</p> <p>Clients normally use host names to identify the server they wish to connect to. This host name is also included in the HTTP request headers. Tomcat extracts the host name from the HTTP headers and looks for a Host with a matching name. If no match is found, the request is routed to the default host. The name of the default host does not have to match a DNS name (although it can) since any request where the DNS name does not match the name of a Host element will be routed to the default host.</p>		
--------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--	--

	<code>name</code>	Usually the network name of this virtual host, as registered in your Domain Name Service server. Regardless of the case used to specify the host name, Tomcat will convert it to lower case internally. One of the Hosts nested within an Engine MUST have a name that matches the defaultHost setting for that Engine. See Host Name Aliases for information on how to assign more than one network name to the same virtual host.	String	localhost
	<code>appBase</code>	The Application Base directory for this virtual host. This is the pathname of a directory that may contain web applications to be deployed on this virtual host. You may specify an absolute pathname, or a pathname that is relative to the \$CATALINA_BASE directory. See Automatic Application Deployment for more information on automatic recognition and deployment of web applications. If not specified, the default of webapps will be used.	String	\${carbon.home}/repository
	<code>autoDeploy</code>	This flag value indicates if Tomcat should check periodically for new or updated web applications while Tomcat is running. If true, Tomcat periodically checks the appBase and xmlBase directories and deploys any new web applications or context XML descriptors found. Updated web applications or context XML descriptors will trigger a reload of the web application. See Automatic Application Deployment for more information.	Boolean	false
	<code>deployOnStartup</code>	This flag value indicates if web applications from this host should be automatically deployed when Tomcat starts. See Automatic Application Deployment for more information.	Boolean	false

	unpackWARs	Set to true if you want web applications that are placed in the appBase directory as web application archive (WAR) files to be unpacked into a corresponding disk directory structure, false to run such web applications directly from a WAR file. WAR files located outside of the Host's appBase will not be expanded.	Boolean	true
>	<Valve	<p>The Access Log Valve creates log files in the same format as those created by standard web servers. These logs can later be analyzed by standard log analysis tools to track page hit counts, user session activity, and so on. The files produced by this Valve are rolled over nightly at midnight. This Valve may be associated with any Catalina container (Context, Host, orEngine), and will record ALL requests processed by that container.</p> <p>Some requests may be handled by Tomcat before they are passed to a container. These include redirects from /foo to /foo/ and the rejection of invalid requests. Where Tomcat can identify the Context that would have handled the request, the request/response will be logged in the AccessLog(s) associated Context, Host and Engine. Where Tomcat cannot identify theContext that would have handled the request, e.g. in cases where the URL is invalid, Tomcat will look first in the Engine, then the default Host for the Engine and finally the ROOT (or default) Context for the default Host for an AccessLog implementation. Tomcat will use the first AccessLog implementation found to log those requests that are rejected before they are passed to a container.</p> <p>The output file will be placed in the directory given by the directory attribute. The name of the file is composed by concatenation of the configured prefix, timestamp and suffix. The format of the timestamp in the file name can be set using the fileDateFormat attribute. This timestamp will be omitted if the file</p>		

rotation is switched off by setting rotatable to false.

**Warning:** If multiple AccessLogValve instances are used, they should be configured to use different output files.

If sendfile is used, the response bytes will be written asynchronously in a separate thread and the access log valve will not know how many bytes were actually written. In this case, the number of bytes that was

		passed to the sendfile thread for writing will be recorded in the access log valve.		
	className	Java class name of the implementation to use.	String	org.wso2.carbon.tomca
	pattern	A formatting layout identifying the various information fields from the request and response to be logged, or the word common or combined to select a standard format.	String	combined
	suffix	The suffix added to the end of each log file name.	String	.log
	prefix	The prefix added to the start of each log file name.	String	http_access_
	directory	Absolute or relative path name of a directory in which log files created by this valve will be placed. If a relative path is specified, it is interpreted as relative to \$CATALINA_BASE. If no directory attribute is specified, the default value is "logs" (relative to \$CATALINA_BASE).	String	\${carbon.home}/repository
	threshold	Minimum duration in seconds after which a thread is considered stuck. If set to 0, the detection is disabled.  Note: since the detection is done in the background thread of the Container (Engine, Host or Context) declaring this Valve, the threshold should be higher than the backgroundProcessorDelay of this Container.	Int	600

#### Configuring config-validation.xml

The <PRODUCT\_HOME>/repository/conf/etc/config-validation.xml file contains the recommended system configurations for a server. When you start the server, the system configurations will be validated against these recommendations, and warnings will be published if conflicts are found. See more details on system requirements for your product on [Installation Prerequisites](#).

Given below are the default recommendations in the config-validation.xml file. If required, you may change some of these recommendations on this file according to the conditions in your production environment.

#### **System Validation**

Following are the system parameters recommended by default.

Parameter	Parameter Value
CPU	800

RAM	2048
swap	2048
freeDisk	1024
ulimit	4096

### JVM Validation

The following JVM parameters are recommended by default.

Parameter	Parameter Value
initHeapSize	256
maxHeapSize	512
maxPermGenSize	256

### System Property Validation

The following properties are required to be setup in your system. Therefore, it is not recommended to remove these validations from the `config-validation.xml` file.

- carbon.home
- carbon.config.dir.path
- axis2.home

### Supported OS Validation

The product has been tested for compatibility with the following operating systems. Therefore, by default, the system is validated against these operating systems.

- Linux
- Unix
- Mac OS
- Windows Server 2003
- Windows XP
- Windows Vista
- Windows 7
- Mac OS X
- Windows Server 2008
- Windows Server 2008 R2
- AIX

### Configuring identity.xml

Users can change the default configurations by editing the `<PRODUCT_HOME>/repository/conf/identity.xml` file using the information given below.

Click on the table and use the left and right arrow keys to scroll horizontally.

### XML Elements

XML element	Attribute	Description	Data type	Default
<Server>				

	xmlns			
<JDBCPersistenceManager>		Identity related data source configuration.		
<DataSource>				
<Name>		Include a data source name (jndiConfigName) from the set of data sources defined in <b>master-datasources.xml</b> .	String	N/A
<SkipDBSchemaCreation>		If the identity database is created from another place and if it is required to skip schema initialization during the server start up, set the property to "true".	Boolean	FALSE
<OpenID>		OpenID related configurations.		
<OpenIDServerUrl>		This is the URL that the OpenID server (servlet) is running in.	String	N/A
<OpenIDUserPattern>		URL of the pattern that can be configured for the user's OpenID.	String	N/A
<OpenIDSkipUserConsent>		Set to false if the users must be prompted for approval.	Boolean	FALSE
<OpenIDRememberMeExpiry>		Expiry time of the OpenID RememberMe token in minutes.	Int	0 Minutes
<UseMultifactorAuthentication>		Multifactor authentication configuration.	Boolean	FALSE
<DisableOpenIDDumbMode>		To enable or disable OpenID dumb mode.	Boolean	FALSE
<SessionTimeout>		OpenID session timeout in seconds.	Int	36000 Seconds
<AcceptSAMLSSOLogin>		Skips authentication if the valid SAML2 Web SSO browser session is available.	Boolean	FALSE
<ClaimsRetrieverImplClass>		User claim retrieving module for OpenID.		
<OAuth>		OAuth related configurations.		
<AuthorizationCodeDefaultValidityPeriod>		Default validity period for Authorization Code in seconds.	Int	300 Seconds
<AccessTokenDefaultValidityPeriod>		Default validity period for Access Token in seconds.	Int	3600 Seconds
<TimestampSkew>		Timestamp skew in seconds.	Int	300 Seconds
<EnableOAuthCache>		Enable OAuth caching. This cache has the replication support.	Boolean	TRUE

<TokenPersistencePreprocessor>		Configure the security measures needed to be done prior to storing the token in the database, such as hashing, encrypting, etc.	String	org.wso:
<SupportedResponseTypes>		Supported OAuth2.0 response types.	String values with Comma separated	token, c
<SupportedGrantTypes>		Supported OAuth2.0 grant types.	String values with Comma separated	authoriz :grant-ty
<OAuthCallbackHandlers>				
<OAuthCallbackHandler>		OAuth callback handler module class name.	String	N/A
<EnableAssertions>		Assertions can be used to embed parameters into the access token.		
<UserName>		This enables you to add the user name as an additional parameter if you require it.	Boolean	FALSE
<EnableAccessTokenPartitioning>		This should be set to true when using multiple user stores and keys should be saved into different tables according to the user store. By default, all the application keys are saved into the same table. UserName Assertion should be 'true' to use this.	Boolean	FALSE
<AccessTokenPartitioningDomains>		This includes the user store domain names and mapping to the new table name. E.g., if you provide 'A:foo.com', foo.com should be the user store domain name and 'A' represents the relevant mapping of the token store table, i.e., tokens will be added to a table called IDN_OAUTH2_ACCESS_TOKEN_A.	String values with Comma separated	N/A
<AuthorizationContextTokenGeneration>				
<Enabled>		This mentions whether token generation is enabled or not.	Boolean	FALSE
<TokenGeneratorImplClass>		Token generation class name.	String	org.wso:

<ClaimsRetrieverImplClass>		Claim retrieving class name for generating a token.		org.wso:
<ConsumerDialectURI>		Claim Dialect URI that is used for claim retrieving.		http://ws
<SignatureAlgorithm>		Signature algorithm used for sign the token.		SHA256
<AuthorizationContextTTL>		Token time to live value.	Long	15 Minu
<SAML2Grant>		Configuration related to SAML2 Grant type.		
<OpenIDConnect>				
<IDTokenBuilder>		IDToken generator implementation class name.	String	org.wso:
<IDTokenIssuerID>		The value of TokenIssuerID of the IDToken. This is a unique value and should be changed according to the deployment values.	String	OIDCAu
<IDTokenSubjectClaim>		This is the claim used as the subject of the IDToken. You can use different claims such as <a href="http://wso2.org/claims/emailaddress">http://wso2.org/claims/emailaddress</a> .	String	http://ws
<IDTokenCustomClaimsCallBackHandler>		Claim callback implementation class name. This is used to return custom claims with the IDToken.	String	org.wso:
<IDTokenExpiration>		The expiration value of the IDToken in seconds.	Int	300 Sec
<UserInfoEndpointClaimDialect>		Defines which claim dialect should be returned from the User Endpoint.	String	http://ws
<UserInfoEndpointClaimRetriever>		Defines the implementation name of the class which builds the claims for the user info endpoint's response.	String	org.wso:
> <UserInfoEndpointRequestValidator>		Implementation name of the class that validates the user info request against the specification.	String	org.wso:
> <UserInfoEndpointAccessTokenValidator>		Implementation name of the class that validates the access token.	String	org.wso:
> <UserInfoEndpointResponseBuilder>		Implementation name of the class that builds the user info request.	String	org.wso:
<SkipUserConsent>		Set to false if the users must be prompted for approval.	Boolean	FALSE
<MultifactorAuthentication>				

<XMPPSettings>		XMPP setting for multifactor authentication.		
<XMPPConfig>				
<XMPPProvider>		XMPP provider name.	String	N/A
<XMPPServer>		XMPP server name.	String	N/A
<XMPPPort>		XMPP server's port.	Int	N/A
<XMPPExt>		XMPP domain.	String	N/A
<XMPPUserName>		User name for login to XMPP server.	String	N/A
<XMPPPassword>		Password for login to XMPP server.	String	N/A
<SSOService>				
<IdentityProviderURL>		Unique identifier for IDP. This would be passed as Issuer in SAML2 response.	String	N/A
<SingleLogoutRetryCount>		Number of retries that must be done if a single logout request is not received from the SP.	Int	
<SingleLogoutRetryInterval>		Interval between two re-tries.	Int	60 Seco
<TenantPartitioningEnabled>		This would add the tenant domain as parameter into the ACS URL.	Boolean	FALSE
<SessionTimeout>		Remember me session timeout in seconds.	Int	36000 S
<AttributesClaimDialect>		Claim Dialect URI that is used for claim retrieving.	String	http://ws
<AcceptOpenIDLogin>		Skips authentication if the valid OpenID login session is available.	Boolean	FALSE
<ClaimsRetrieverImplClass>		Claim retrieving class name for generating a token.	String	N/A
<SAMLResponseValidityPeriod>		SAML Token validity period in minutes.	Int	5 Minute
<UseAuthenticatedUserDomainCrypto>		When set to true, this is useful in tenant mode setup with older versions of API Manager. This indicates that the SAML2 SSO SAML Response must be signed using the authenticated user's tenant keystore.	Boolean	FALSE
<EntitlementSettings>				
<ThriftBasedEntitlementConfig>		Thrift transport configurations for entitlement service.		
<EnableThriftService>		Enable thrift transport.	Boolean	FALSE

<ReceivePort>		Thrift listening port.	Int	N/A
<ClientTimeout>		Thrift session time out in seconds.	Int	N/A
<KeyStore>		Thrift key store configurations used for SSL.		
<Location>		Key store location	String	N/A
<Password>		Key store password	String	N/A
<SCIMAuthenticators>				
<Authenticator>		Defines implementations of SCIM authenticator.	String	org.wso2.securevault.org.wso2.securevault
<Property>		Configuration properties of each authenticator implementation.	String	N/A

#### Configuring master-datasources.xml

Users can change the default configurations by editing the <PRODUCT\_HOME>/repository/conf/datasources/master-datasources.xml file using the information in the following table.

#### XML Elements

Click on the table and use the left and right arrow keys to scroll horizontally. For sample values, see the Example below the table.

XML element	Attribute	Description	Data type
<datasources-configuration>	xmlns	The root element. The namespace is specified as: xmlns:svns="http://org.wso2.securevault/configuration"	
<providers>		The container element for the datasource providers.	
<provider>		The datasource provider, which should implement org.wso2.carbon.ndatasource.common.spi.DataSourceReader. The datasources follow a pluggable model in providing datasource type implementations using this approach.	Fully qualified Java class
<datasources>		The container element for the datasources.	
<datasource>		The root element of a datasource.	
<name>		Name of the datasource.	String
<description>		Description of the datasource.	String
<jndiConfig>		The container element that allows you to expose this datasource as a JNDI datasource.	
<name>		The JNDI resource name to which this datasource will be bound.	String

<environment>		The container element in which you specify the following JNDI properties: <ul style="list-style-type: none"><li>• <code>java.naming.factory.initial</code>: Selects the registry service provider as the initial context.</li><li>• <code>java.naming.provider.url</code>: Specifies the location of the registry when the registry is being used as the initial context.</li></ul>	Fully qualified Java class
<definition>	type	The container element for the data source definition. Set the type attribute to RDBMS, or to custom if you're creating a custom type. The "RDBMS" data source reader expects a "configuration" element with the sub-elements listed below.	String
<configuration>		The container element for the RDBMS properties.	
<url>		The connection URL to pass to the JDBC driver to establish the connection.	URL
<username>		The connection user name to pass to the JDBC driver to establish the connection.	String
<password>		The connection password to pass to the JDBC driver to establish the connection.	String
<driverClassName>		The class name of the JDBC driver to use.	Fully qualified Java class
<maxActive>		The maximum number of active connections that can be allocated from this pool at the same time.	Integer
<maxWait>		Maximum number of milliseconds that the pool waits (when there are no available connections) for a connection to be returned before throwing an exception.	Integer
<testOnBorrow>		Specifies whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and we will attempt to borrow another. When set to true, the <code>validationQuery</code> parameter must be set to a non-null string.	Boolean
<validationQuery>		The SQL query used to validate connections from this pool before returning them to the caller. If specified, this query does not have to return any data, it just can't throw a SQLException. The default value is null. Example values are <code>SELECT 1(mysql)</code> , <code>select 1 from dual(oracle)</code> , <code>SELECT 1(MS Sql Server)</code> .	String

<validationInterval>		To avoid excess validation, only run validation at most at this frequency (interval time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again. The default value is 30000 (30 seconds).	Long
----------------------	--	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

### Example

```

<datasources-configuration xmlns:svns="http://org.wso2.securevault/configuration">
 <providers>
 <provider>
 org.wso2.carbon.ndatasource.rdbms.RDBMSDataSourceReader
 </provider>
 </providers>
 <datasources>
 <datasource>
 <name>WSO2_CARBON_DB</name>
 <description>The datasource used for registry and user manager</description>
 <jndiConfig>
 <name>jdbc/WSO2CarbonDB</name>
 </jndiConfig>
 <definition type="RDBMS">
 <configuration>
 <url>
 jdbc:h2:repository/database/WSO2CARBON_DB;DB_CLOSE_ON_EXIT=FALSE;LOCK_TIMEOUT=60000
 </url>
 <username>wso2carbon</username>
 <password>wso2carbon</password>
 <driverClassName>org.h2.Driver</driverClassName>
 <maxActive>50</maxActive>
 <maxWait>60000</maxWait>
 <testOnBorrow>true</testOnBorrow>
 <validationQuery>SELECT 1</validationQuery>
 <validationInterval>30000</validationInterval>
 </configuration>
 </definition>
 </datasource>
 </datasources>
</datasources-configuration>

```

### Configuring registry.xml

Users can change the default configurations by editing the <PRODUCT\_HOME>/repository/conf/registry.xml file using the information given below.

Click on the table and use the left and right arrow keys to scroll horizontally.

### XML Elements

XML element	Attribute	Description
<wso2registry>		

<currentDBConfig>		The server can only handle one active configuration at a time. The currentDBConfig element is used to specify the database configuration that is active at present. This setting is valid on a global level.  For more information, see the Governance Registry documentation here: <a href="#">base+Configuration+Details</a>
<readOnly>		To run the registry in read-only mode, set the readOnly element to true. This setting is valid on a global level.  For more information, see the Governance Registry documentation here: <a href="#">stry+Configuration+Details</a>
<enableCache>		To enable registry caching, set the enableCache element to true. Once enabled, queries will be executed against the cache instead of the database. This setting is valid on a global level.
<registryRoot>		The registryRoot parameter can be used to define the apparent root of the registry. This setting is valid on a global level.  For more information, see the Governance Registry documentation here: <a href="#">stry+Configuration+Details</a>
<dbConfig>		
	name	
<dataSource>		
<handler>		Handlers are pluggable components, that contain custom processing logic. They inherit from an abstract class named Handler, which provides default implementations of various utilities useful for concrete handler implementations.  Handler implementations can provide alternative behaviors for basic registry operations by overriding methods in the Handler class.  For more information, see the Governance Registry documentation here: <a href="#">ler+Configuration+Details</a>
	class	
<filter>		
	class	
<remoteInstance>		In order to mount an external registry, you have to define the remote instance configuration, the Atom-based configuration model or the WebService-based configuration model.  For more information, see the Governance Registry documentation here: <a href="#">ote+Instance+and+Mount+Configuration+Details</a>
	url	The URL of the remote instance.
<ID>		Remote instance ID.
<username>		Username of the remote registry login.
<password>		Password of the remote registry login.
<dbConfig>		The database configuration to use.

<readOnly>		To run the registry in read-only mode set the readOnly element to true immutable instance of registry repository. This setting is valid only for the specific remote instance. For more information, see the Governance Registry documentation here: <a href="#">Governance Registry Configuration+Details</a>
<enableCache>		To enable registry caching, set the enableCache element to true. Once a remote instance has been defined, a collection on the remote instance will be executed against the cache instead of the database. This setting is valid only for the specific remote instance.
<registryRoot>		The registryRoot parameter can be used to define whether the appare setting is valid only for the specific remote instance. For more information, see the Governance Registry documentation here: <a href="#">Governance Registry Configuration+Details</a>
<mount>		Once a remote instance has been defined, a collection on the remote instance will be added to the specified path. For more information, see the Governance Registry documentation here: <a href="#">Governance Registry Configuration+Details</a>
	path	The path to which the mount will be added to.
	overwrite	Whether an existing collection at the given path would be overwritten or not.
<instanceID>		Remote instance ID.
<targetPath>		The path on the remote registry.
<versionResourcesOnChange>		You can configure whether you want to auto-version the resources (node element to true. In this configuration it will create a version for the resource.) For more information, see the Governance Registry documentation here: <a href="http://docs.wso2.org/display/Governance501/One-time%29+and+Auto+Versioning+Resources">http://docs.wso2.org/display/Governance501/One-time%29+and+Auto+Versioning+Resources</a>
<staticConfiguration>		While most configuration options can be changed after the first run of the server, the static configuration section under the staticConfiguration parameter Static Configuration and expect it to take effect, you will have to either stop the server passing the -Dsetup system property which will re-generate the static configuration file.
		You are supposed to change the static configuration section only before the first start-up.) For more information, see the Governance Registry documentation here: <a href="http://docs.wso2.org/display/Governance501/Configuration+for+Static+%28One-time%29+and+Auto+Versioning+Resources">http://docs.wso2.org/display/Governance501/Configuration+for+Static+%28One-time%29+and+Auto+Versioning+Resources</a>
<versioningProperties>		Whether the properties are versioned when a snapshot is created.
<versioningComments>		Whether the comments are versioned when a snapshot is created.
<versioningTags>		Whether the tags are versioned when a snapshot is created.
<versioningRatings>		Whether the ratings are versioned when a snapshot is created.

#### Configuring user-mgt.xml

Users can change the default user management functionality related configurations by editing the <PRODUCT\_HOME>/repository/conf/user-mgt.xml file using the information given below.

Click on the table and use the left and right arrow keys to scroll horizontally.

#### XML Elements

XML element	Attribute	Description	Data type	Default value	Mandatory/Optional

<UserManager>	User kernel configuration for Carbon server.			
<Realm>	Realm configuration.			
<Configuration>				
<AddAdmin>	Specifies whether the admin user and admin role will be created in the primary user store. This element enables the user to create additional admin users in the user store. If the <AdminUser> element does not exist in the external user store, it will be automatically created only if this property is set to true. If the value is set to false, the given admin user and role should already exist in the external user store.	Boolean	true	Mandatory
<AdminRole>	The role name that is used as an admin role for the Carbon server.	String	N/A	Mandatory
<AdminUser>				
> <UserName>	User name that is used to represent an admin user for the Carbon server.	String	N/A	Mandatory
<Password>	Password of the admin user, If the admin user needs to be created in the Carbon server.	String	N/A	Optional
<EveryOneRoleName>	By default, every user in the user store is assigned to this role.	String	N/A	Mandatory
<Property>	User realm configuration specific property values.	String	N/A	Mandatory

<code>&lt;UserStoreManager&gt;</code>	<p>User Store manager implementation classes and their configurations for user realm. Use the <code>ReadOnlyLDAPUserStoreManager</code> to do read-only operations for external LDAP user stores.</p> <p>To do both read and write operations, use the <code>ReadWriteLDAPUserStoreManager</code> for external LDAP user stores.</p> <p>If you wish to use an Active Directory Domain Service (AD DS) or Active Directory Lightweight Directory Service (AD LDS), use the <code>ActiveDirectoryUserStoreManager</code>. This can be used for both read-only and read/write operations.</p> <p>Use <code>JDBCUserStoreManager</code> for both internal and external JDBC user stores.</p>	String	N/A	Mandatory
	class			
<code>&lt;Property&gt;</code>	User store configuration specific property values. See <a href="#">Working with Properties of Primary User Stores</a> for more information.	String	N/A	Optional
<code>&lt;AuthorizationManager&gt;</code>	Authorization manager implementation class and its configuration for user realm.	String	N/A	Mandatory
	class			
<code>&lt;Property&gt;</code>	Authorization manager configuration specific property values.	String	N/A	Optional

### Configuring axis2\_blocking\_client.xml

This file contains parameters which apply to scenarios where blocking transports are used i.e. Message Processors and the [Callout Mediator](#). These parameters are grouped as follows.

#### Parameters

This section is used to add parameters to be applied to scenarios in which blocking transports are used.

A parameter is a name-value pair. All top-level parameters (those that are direct sub-elements of the root element) will be transformed into properties in the AxisConfiguration and can be accessed in the running system. The `name` attribute (required) specifies the parameter name. If you set the `locked` attribute to `true` (the default value is `false`), this parameter's value cannot be overridden by services and other configurations.

#### Message Receivers

This section is used to add message receivers that can be used in scenarios where blocking transports are used. The `class` attribute specifies the implementation class of the message receiver. The `mep` attribute specifies the message exchange pattern supported by the message receiver.

The following example shows the format in which a message receiver can be added.

```
<messageReceivers>
 <messageReceiver mep="http://www.w3.org/ns/wsdl/in-only"
 class="org.apache.axis2.receivers.RawXMLINOnlyMessageReceiver"/>
</messageReceivers>
```

## **Message Formatters**

This section is used to define message formatters to be used in scenarios where blocking transports are used. A message formatter is used to build the outgoing stream of a message. The `contentType` attribute specifies which message types are handled by this formatter, and the `class` attribute specifies the formatter implementation class. See [Working with Message Builders and Formatters](#) for more information.

Message formatters can be added using the format shown in the example below.

```
<messageFormatters>
 <messageFormatter contentType="application/x-www-form-urlencoded"
 class="org.apache.axis2.transport.http.XFormURLEncodedFormatter"/>
</messageFormatters>
```

## **Message Builders**

This section is used to define message builders to be used in scenarios where blocking transports are used. A message builder is used by transport receivers to process the raw data in the payload of a received message and convert it to the SOAP format. The `contentType` attribute specifies which message types are handled by this builder, and the `class` attribute specifies the builder implementation class. See [Working with Message Builders and Formatters](#) for more information.

Message builders can be added using the format shown in the example below.

```
<messageBuilders>
 <messageBuilder contentType="application/xml"
 class="org.apache.axis2.builder.ApplicationXMLBuilder"/>
</messageBuilders>
```

## **Transport Ins**

Transport receivers can be added in this section using the format shown in the example below.

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory">
 <parameter
name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFact
ory</parameter>
 <parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
 <parameter
name="transport.jms.ConnectionFactoryJNDIName">TopicConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 </parameter>
</transportReceiver>

```

Configurable parameters for each transport receiver are as follows.

Parameter Name	Description
java.naming.factory.initial	JNDI initial context factory class. The class must implement the java.naming.spi.InitialContextFactory interface.
java.naming.provider.url	The URL of the JNDI provider.
transport.jms.ConnectionFactoryJNDIName	The JNDI name of the connection factory.
transport.jms.ConnectionFactoryType	The type of the connection factory.

### Transport Outs

Transport senders can be added in this section using the format shown in the example below.

```

<transportSender name="http"

class="org.apache.axis2.transport.http.CommonsHTTPTransportSender">
 <parameter name="PROTOCOL">HTTP/1.1</parameter>
 <parameter name="Transfer-Encoding">chunked</parameter>
 <parameter name="cacheHttpClient">true</parameter>
 <parameter name="defaultMaxConnectionsPerHost">200</parameter>
 <!-- If following is set to 'true', optional action part of the Content-Type
will not be added to the SOAP 1.2 messages -->
 <!-- <parameter name="OmitSOAP12Action">true</parameter> -->
</transportSender>

```

Configurable parameters for each transport sender are as follows.

Parameter Name	Description
PROTOCOL	The transport protocol.

Transfer-Encoding	This parameter enables you to specify whether the data sent should be chunked instead of the Content-Length header if you want to upload files without having to know the amount of data to be uploaded in advance.
cacheHttpClient	This parameter is used to specify whether the HTTP client should save cached entries and the cached responses in the JVM memory or not.
defaultMaxConnectionsPerHost	The maximum number of connections that will be created per host server by the client. If the backend server is slow, the connections in use at a given time may take a long time to be released and added back to the connection pool. As a result, connections may not be available for some requests and you may get org.apache.commons.httpclient.ConnectionPoolTimeoutException: Timeout waiting for connection error. In such situations, it is recommended to increase the value for this parameter.

## Global Modules

This section is used to engage a module. The `ref` attribute specifies the module name.

```
<module ref="addressing"/>
```

## Clustering

This section is used to enable clustering. The `class` attribute specifies the clustering agent class. The `enable` attribute can be set to `true` to enable clustering. It is set to `false` by default.

## Phases

This section is used to specify the order of phases in the execution chains of different types of flow. The type of flow to which the phasing is defined is specified by the attribute and it can be one of the following.

- InFlow
- OutFlow
- InFaultFlow
- OutFaultFlow

The following configuration is an example of how phases are configured.

```
<phaseOrder type="InFlow">
 <!-- System pre defined phases -->
 <phase name="Transport">
 <handler name="RequestURIBasedDispatcher"
 class="org.apache.axis2.dispatchers.RequestURIBasedDispatcher">
 <order phase="Transport"/>
 </handler>
 <handler name="SOAPActionBasedDispatcher"
 class="org.apache.axis2.dispatchers.SOAPActionBasedDispatcher">
 <order phase="Transport"/>
 </handler>
 </phase>
 <phase name="Addressing">
 <handler name="AddressingBasedDispatcher"
 class="org.apache.axis2.dispatchers.AddressingBasedDispatcher">
 <order phase="Addressing"/>
 </handler>
 </phase>
</phaseOrder>
```

```

<phase name="Security"/>
<phase name="PreDispatch"/>
<phase name="Dispatch" class="org.apache.axis2.engine.DispatchPhase">
 <handler name="RequestURIBasedDispatcher"
 class="org.apache.axis2.dispatchers.RequestURIBasedDispatcher"/>
 <handler name="SOAPActionBasedDispatcher"
 class="org.apache.axis2.dispatchers.SOAPActionBasedDispatcher"/>
 <handler name="RequestURIOperationDispatcher"

class="org.apache.axis2.dispatchers.RequestURIOperationDispatcher"/>
 <handler name="SOAPMessageBodyBasedDispatcher"

class="org.apache.axis2.dispatchers.SOAPMessageBodyBasedDispatcher"/>
 <handler name="HTTPLocationBasedDispatcher"

class="org.apache.axis2.dispatchers.HTTPLocationBasedDispatcher"/>
 </phase>
 <phase name="RMPhase"/>
 <!-- System predefined phases -->
 <!-- After Postdispatch phase module author or service author can add any
phase he want -->
 <phase name="OperationInPhase"/>
 <phase name="soapmonitorPhase"/>
</phaseOrder>
<phaseOrder type="OutFlow">
 <!-- user can add his own phases to this area -->
 <phase name="soapmonitorPhase"/>
 <phase name="OperationOutPhase"/>
 <!--system predefined phase-->
 <!--these phase will run irrespective of the service-->
 <phase name="RMPhase"/>
 <phase name="PolicyDetermination"/>
 <phase name="MessageOut"/>
 <phase name="Security"/>
</phaseOrder>
<phaseOrder type="InFaultFlow">
 <phase name="Addressing">
 <handler name="AddressingBasedDispatcher"
 class="org.apache.axis2.dispatchers.AddressingBasedDispatcher">
 <order phase="Addressing"/>
 </handler>
 </phase>
 <phase name="Security"/>
 <phase name="PreDispatch"/>
 <phase name="Dispatch" class="org.apache.axis2.engine.DispatchPhase">
 <handler name="RequestURIBasedDispatcher"
 class="org.apache.axis2.dispatchers.RequestURIBasedDispatcher"/>
 <handler name="SOAPActionBasedDispatcher"
 class="org.apache.axis2.dispatchers.SOAPActionBasedDispatcher"/>
 <handler name="RequestURIOperationDispatcher"

class="org.apache.axis2.dispatchers.RequestURIOperationDispatcher"/>
 <handler name="SOAPMessageBodyBasedDispatcher"

class="org.apache.axis2.dispatchers.SOAPMessageBodyBasedDispatcher"/>
 <handler name="HTTPLocationBasedDispatcher"

class="org.apache.axis2.dispatchers.HTTPLocationBasedDispatcher"/>
 </phase>

```

```
<phase name="RMPhase"/>
<!-- user can add his own phases to this area -->
<phase name="OperationInFaultPhase"/>
<phase name="soapmonitorPhase"/>
</phaseOrder>
<phaseOrder type="OutFaultFlow">
<!-- user can add his own phases to this area -->
<phase name="soapmonitorPhase"/>
<phase name="OperationOutFaultPhase"/>
<phase name="RMPhase"/>
<phase name="PolicyDetermination"/>
```

```
<phase name="MessageOut" />
<phase name="Security" />
</phaseOrder>
```

The configurable parameters for this section are as follows.

Parameter Name	Attribute	Description
<phase>	name	<p>You add phases using the &lt;phase&gt; sub-element. In the In phase orders, all phases before the Dispatch phase are global phases, and phases after the Dispatch phase are operation phases.</p> <p>In the Out phase orders, phases before the MessageOut phase are global phases, and phases after the MessageOut phase are operation phases.</p> <p>The name attribute specifies the phase name. You can add the &lt;handler&gt; sub-element under &lt;phase&gt; to execute a specific handler during a phase.</p>
<handler>	name class	The handler (message processing functionality) to execute during this phase. Handlers are combined into chains and phases to provide customizable functionality such as security, reliability, etc. Handlers must be multi-thread safe and should keep all their state in Context objects (see the org.apache.axis2.context package).

## Properties Files

This section provides detailed definitions of the following properties configuration files of WSO2 ESB.

- Configuring passthru-http.properties
- Configuring nhttp.properties
- Configuring synapse.properties

### Configuring passthru-http.properties

The <ESB\_Home>/repository/conf/passthru-http.properties file contains tuning parameters related to the HTTP Pass Through transport. This is the default transport of the WSO2 ESB since it optimises the performance of the ESB in most scenarios. The performance can be further optimised when this transport is used with header based routing or smaller sized messages.

These parameters can be modified as required based on your production environment. This information is provided as a reference for users who are already familiar with the product features and want to know how to configure them.

#### ***Pass-through HTTP transport specific tuning parameters***

Parameter Name	Description	Default Value
worker_pool_size_core	Initial number of threads in the worker pool. If the number of live threads is less than the value defined here, a new thread is created to process the response.	400

worker_pool_size_max	Maximum number of threads in the worker pool.  If the queue ( <code>worker_pool_queue_length</code> ) is full and the number of live threads is less than the value defined here, a new thread will be created to process the response.	500
http.socket.timeout	Maximum period of inactivity between two consecutive data packets. Given in milliseconds.	180000
worker_thread_keepalive_sec	The keep-alive time in seconds for idle threads in the worker pool. Once this time has elapsed for an idle thread, it will be destroyed.  The purpose of this parameter is to optimize the usage of resources by avoiding wastage that results from having idle threads.	60
worker_pool_queue_length	The length of the queue that is used to hold runnable tasks to be executed by the worker pool.  If the number of live threads is equal to the core pool size ( <code>worker_pool_size_core</code> ) and all threads are busy, new tasks will be pushed into the queue so that they can be processed when a thread becomes available.  If you need new threads to be created beyond the core pool size ( <code>worker_pool_size_core</code> ), ensure you change the default value of this parameter so that it becomes a bounded queue  If the queue is full, all threads are busy and the number of live threads is equal to the max pool size ( <code>worker_pool_size_max</code> ), new tasks will be rejected by the worker.	-1
io_threads_per_reactor	Defines the number of IO dispatcher threads used per reactor. The value for this property should not exceed the number of cores in the server.	2
io_buffer_size	Size in bytes of the buffer through which data passes.	16384
http.max.connection.per.host.port	Defines the maximum number of connections allowed per host port.	32767
http.socket.reuseaddr	If this parameter is set to <code>true</code> , it is possible to open another socket on the same port as that of the socket which is currently used by the ESB server to listen to connections. This is useful when recovering from a crash. On such occasions, if the socket is not properly closed, a new socket can be opened in order to continue with the listening.	<code>true</code>
http.socket.buffer-size	This is used to configure the <code>SessionInputBuffer</code> size of http core. The <code>SessionInputBuffer</code> is used to fill data that is read from the OS socket. This parameter does not affect the OS socket buffer size.	8192

### Other parameters

Parameter Name	Description	Default Value
http.block_service_list	If this parameter is set to <code>true</code> , all the services deployed to the WSO2 ESB cannot be accessed from the <code>http(s):&lt;esb&gt;:8280/services/</code> URL.	<code>true</code>
http.origin-server	This parameter is used to customize the name of the origin server in the HTTP response header.	
http.user.agent.preserve	If this parameter is set to <code>true</code> , the <code>User-Agent</code> HTTP header of messages passing through the ESB will be preserved and printed in the outgoing message.	<code>false</code>
http.server.preserve	If this parameter is set to <code>true</code> , the <code>Server</code> HTTP header of messages passing through the ESB will be preserved and printed in the outgoing message.	<code>false</code>
http.connection.disable.keepalive	If this parameter is set to <code>true</code> , the HTTP connections with the back end service will be closed soon after the request is served. It is recommended to set this property to <code>false</code> so that the ESB does not have to create a new connection every time it sends a request to a back end service. However, you may need to close connections after they are used if the back end service does not provide sufficient support for keep alive connections.	<code>false</code>
rest.dispatcher.service	This parameter determines how requests from clients to the ESB are dispatched to the REST APIs of the ESB. The <code>MultitenantDispatcherService</code> which is selected by default for this parameter dispatches requests from multiple clients at a given time.	<code>MultitenantDispatcherService</code>

### Example

The following example shows excerpts from a `passthru-http.properties` file.

```

This file contains the configuration parameters used by the Pass-through HTTP
transport
Pass-through HTTP transport specific tuning parameters
worker_pool_size_core=400
worker_pool_size_max=500
#worker_thread_keepalive_sec=60
#worker_pool_queue_length=-1
#io_threads_per_reactor=2
io_buffer_size=16384
#http.max.connection.per.host.port=32767
This property is crucial for automated tests
http.socket.reuseaddr=true
Other parameters
http.block_service_list=true
#http.user.agent.preserve=false
#http.server.preserve=true
#http.connection.disable.keepalive=false
rest.dispatcher.service=__MultitenantDispatcherService

```

### Configuring nhttp.properties

The <ESB\_Home>/repository/conf/nhttp.properties file contains tuning parameters related to non-blocking HTTP transport. These parameters can be modified as required based on your production environment. This information is provided as reference for users who are already familiar with the product features and want to know how to configure them.

#### **Parameters used by non-blocking HTTP transport**

Parameter Name	Description	Default Value
http.socket.timeout	Maximum period of inactivity between two consecutive data packets. Given in milliseconds. This parameter is also defined as SO_TIMEOUT.	60000
nhttp_buffer_size	The size of the buffer through which data passes when receiving/transmitting NHTTP requests. This is given in bytes.	8192
http.tcp.nodelay	This determines whether Nagle's algorithm (for improving the efficiency of TCP/IP by reducing the number of packets sent over the network) is used. Value 0 is used to enable this algorithm and value 1 is used to disable it. The algorithm should be enabled if you need to reduce bandwidth consumption.	1
http.connection.stalecheck	This determines whether stale connection check is used. Value 0 is used to enable stale connection check and value 1 is used to disable it. When this parameter is enabled, connections which are no longer used will be identified and disabled before each request execution. Stale connection check should be disabled when performing critical operations.	0

#### **Properties for AIX based deployment**

Parameter Name	Description	Default Value
http.nio.interest-ops-queueing	Determines whether interestOps() queueing is enabled for the I/O reactors.	true

### HTTP Sender (Client Worker) thread pool parameters

The following parameters relate to worker thread pools that are used to process responses in non-blocking HTTP transport:

Parameter Name	Description	Default Value
lst_t_core	Initial number of threads in the response processing worker (Client Worker) pool.  If the number of live threads is less than the value defined here, a new thread is created to process the response.	20
lst_t_max	Maximum number of threads in the response processing worker (Client Worker) pool.  If the queue (lst_qlen) is full and the number of live threads is less than the value defined here, a new thread will be created to process the response.	100
lst_alive_sec	The keep-alive time in seconds for idle threads in the response processing worker pool. Once this time has elapsed for an idle thread, it will be destroyed.  The purpose of this parameter is to optimize the usage of resources by avoiding wastage that results from having idle threads.	5
lst_qlen	The length of the queue that is used to hold runnable tasks to be executed by the response processing worker pool.  If the number of live threads is equal to the core pool size (lst_t_core) and all threads are busy, new responses will be pushed into the queue so that they can be processed when a thread becomes available.  If you need new threads to be created beyond the core pool size (lst_t_core), ensure you change the default value of this parameter so that it becomes a bounded queue.  If the queue is full, all threads are busy and the number of live threads is equal to the max pool size (lst_t_max), new responses will be rejected by the response processing worker.	-1 (Unbounded queue)
lst_io_threads	Sender-side IO workers.	2

When there is an increased load, it is recommended to increase the number of threads mentioned in the parameters above to balance it.

### HTTP Listener (Server Worker) thread pool parameters

The following parameters relate to worker thread pools that are used to process requests in non-blocking HTTP transport:

Listener-side parameters should generally have the same values as the sender-side parameters.

Parameter Name	Description	Default Value
snd_t_core	Initial number of threads in the worker thread pool.  If the number of live threads is less than the value defined here, a new thread is created to process the request.	20
snd_t_max	Maximum number of threads in the request processing worker (Server Worker) pool.  If the queue (snd_qlen) is full and the number of live threads is less than the value defined here, a new thread will be created to process the request.	100
snd_alive_sec	Keep-alive time in seconds for idle threads in the worker pool. Once this time has elapsed for an idle thread, it will be destroyed.  The purpose of this parameter is to optimise the usage of resources by avoiding wastage that results from having idle threads.	5
snd_qlen	The length of the queue that is used to hold runnable tasks to be executed by the request processing worker pool.  If the number of live threads is equal to the core pool size (snd_t_core) and all threads are busy, new requests will be pushed into the queue so that they can be processed when a thread becomes available.  If you need new threads to be created beyond the core pool size (snd_t_core), ensure you change the default value of this parameter so that it becomes a bounded queue.  If the queue is full, all threads are busy and the number of live threads is equal to the max pool size (snd_t_max), new requests will be rejected by the request processing worker.	-1 (Unbounded queue)
lst_io_threads	Sender-side IO workers.	2

### Other parameters

Parameter Name	Description	Default Value
http.block_service_list	If this parameter is set to true, all the services deployed to the WSO2 ESB cannot be viewed from the http(s):<esb>:8280/services/ URL.	false

### Example

The following example shows excerpts from a nhttp.properties file.

```

#
http://www.apache.org/licenses/LICENSE-2.0
#
This file contains the configuration parameters used by the Non-blocking HTTP
transport
#http.socket.timeout=60000
#nhttp_buffer_size=8192
#http.tcp.nodelay=1
#http.connection.stalecheck=0
Uncomment the following property for an AIX based deployment
#http.nio.interest-ops-queueing=true

HTTP Listener thread pool parameters
#snd_t_core=20
#snd_t_max=100
#snd_alive_sec=5
#snd_qlen=-1
#snd_io_threads=2

HTTP Sender thread pool parameters
#lst_t_core=20
#lst_t_max=100
#lst_alive_sec=5
#lst_qlen=-1
#lst_io_threads=2
nhttp.rest.dispatcher.service=__MultitenantDispatcherService
URI configurations that determine if it requires custom rest dispatcher
rest_uri_api_regex=\w+://.+:\d+/t/.*/\w+://.+/\w+/t/.*/^(/t/).*
rest_uri_proxy_regex=\w+://.+:\d+/services/t/.*/\w+://.+/\w+/services/t/.*/^(/services/t/).*

```

### Configuring synapse.properties

The <ESB\_Home>/repository/conf/synapse.properties file contains tuning parameters relating to the mediation engine. These parameters are mainly used when mediators such as `Iterate` and `Clone` which leverage on internal thread pools are used.

The following table explains the parameters included in the `synapse.properties` file.

Parameter Name	Description	Default Value
----------------	-------------	---------------

<code>synapse.threads.core</code>	The initial number of synapse threads in the pool. This parameter is applicable only if the Iterate or the Clone mediator is used to handle a higher load. The number of threads specified for this parameter should be increased as required to balance an increased load.	20
<code>synapse.threads.max</code>	The maximum number of synapse threads in the pool. This parameter is applicable only if the Iterate or the Clone mediator is used to handle a higher load. The number of threads specified for this parameter should be increased as required to balance an increased load.	100
<code>synapse.threads.keepalive</code>	The keep-alive time for extra threads defined in milliseconds. This parameter is applicable only if the Iterate or the Clone mediator is used to handle a higher load.	5
<code>synapse.threads.qlen</code>	The length of the queue that is used to hold the runnable tasks to be executed by the pool. This parameter is applicable only if the Iterate or the Clone mediator is used to handle a higher load.	10

<code>synapse.threads.group</code>	The name of the thread group.	<code>synapse-thread-group</code>
<code>synapse.threads.idprefix</code>	The prefix of each thread name.	<code>SynapseWorker</code>
<code>synapse.sal.endpoints.session.timeout.default</code>	The session time-out time for the session aware load balance endpoint given in milliseconds.	600000
<code>synapse.global_timeout_interval</code>	The maximum number of milliseconds within which a response for the request should be received. A response which arrives after the specified number of seconds cannot be correlated with the request. Hence, a warning will be logged and the request will be dropped. This parameter is also referred to as the time-out handler	120000
<code>statistics.clean.enable</code>	If this parameter is set to <code>true</code> , all the existing statistics would be cleared before processing a request. This is recommended if you want to increase the processing speed.	<code>false</code>
<code>synapse.xpath.dom.failover.enabled</code>	If this parameter is set to <code>true</code> , it will be possible for ESB to switch to xpath 2.0. The default value for this parameter is <code>false</code> since xpath 2.0 evaluations can cause performance degradation.	<code>false</code>

<code>synapse.timeout_handler_interval</code>	The back end service to which a request has been sent are repeatedly called back for responses at time intervals specified for this parameter. Any endpoints have timed out are identified during these time intervals, and they are no longer called back. Note that specifying a lower value for this parameter results in a higher overhead on the system.	15 (seconds)
<code>synapse.script.mediator.pool.size</code>	When using externally referenced scripts, this parameter is used to specify the size of the script engine pool to be used per script mediator. The script engines from this pool are used for externally referenced script execution where updates to external scripts on an engine currently in use may otherwise not be thread safe. It is recommended to keep this value at a reasonable size since there will be a pool per externally referenced script.	15
<code>synapse.connection.read_timeout</code>	The connection time-out in milliseconds when ESB is accessing a WSDL URI.	100000

The parameters of this file may also need to be changed depending on the NHTTP properties as shown in the example below.

If the NHTTP properties are as follows:

```

snd_t_core=100
#snd_t_max=250
snd_io_threads=5
lst_t_core=100
#lst_t_max=250
lst_io_threads=5
http.socket.timeout=60000
#http.connection.disable.keepalive=1

```

A reasonable setting for synapse properties would be as follows:

```

synapse.threads.core=100
synapse.threads.max=250
synapse.threads.keepalive=5
synapse.threads.qlen=1000
synapse.threads.group=synapse-thread-group
synapse.threads.idprefix=SynapseWorker

```

## Calling Admin Services from Apps

WSO2 products are managed internally using SOAP Web services known as **admin services**. WSO2 products come with a management console UI, which communicates with these admin services to facilitate administration capabilities through the UI.

A service in WSO2 products is defined by the following components:

- Service component: provides the actual service
- UI component: provides the Web user interface to the service
- Service stub: provides the interface to invoke the service generated from the service WSDL

There can be instances where you want to call back-end Web services directly. For example, in test automation, to minimize the overhead of having to change automation scripts whenever a UI change happens, developers prefer to call the underlying services in scripts. The topics below explain how to discover and invoke these services from your applications.

### ***Discovering the admin services***

By default, the WSDLs of admin services are hidden from consumers. Given below is how to discover them.

1. Set the <HideAdminServiceWSDLs> element to false in the <PRODUCT\_HOME>/repository/conf/carbon.xml file.
2. Restart the server.
3. Start the WSO2 product with the -DosgiConsole option, such as sh <PRODUCT\_HOME>/bin/wso2server.sh -DosgiConsole in Linux.
4. When the server is started, hit the enter/return key several times to get the OSGI shell in the console.
5. In the OSGI shell, type: osgi> listAdminServices
6. The list of admin services of your product are listed. For example:

```
osgi> listAdminServices
Admin services deployed on this server:
1. ProvisioningAdminService, ProvisioningAdminService, https://192.168.219.1:8243/services/ProvisioningAdminService
2. SynapseApplicationAdmin, SynapseApplicationAdmin, https://192.168.219.1:8243/services/SynapseApplicationAdmin
3. CarbonAppUploader, CarbonAppUploader, https://192.168.219.1:8243/services/CarbonAppUploader
4. OperationAdmin, OperationAdmin, https://192.168.219.1:8243/services/OperationAdmin
5. SequenceAdminService, SequenceAdminService, https://192.168.219.1:8243/services/SequenceAdminService
6. MediationLibraryAdminService, MediationLibraryAdminService, https://192.168.219.1:8243/services/MediationLibraryAdminService
7. StatisticsAdmin, StatisticsAdmin, https://192.168.219.1:8243/services/StatisticsAdmin
8. LoggedUserInfoAdmin, LoggedUserInfoAdmin, https://192.168.219.1:8243/services/LoggedUserInfoAdmin
9. MediationStatisticsAdmin, MediationStatisticsAdmin, https://192.168.219.1:8243/services/MediationStatisticsAdmin
10. TopicManagerAdminService, TopicManagerAdminService, https://192.168.219.1:8243/services/TopicManagerAdminService
11. MessageProcessorAdminService, MessageProcessorAdminService, https://192.168.219.1:8243/services/MessageProcessorAdminService
12. ApplicationAdmin, ApplicationAdmin, https://192.168.219.1:8243/services/ApplicationAdmin
13. NDataSourceAdmin, NDataSourceAdmin, https://192.168.219.1:8243/services/NDataSourceAdmin
14. ServiceGroupAdmin, ServiceGroupAdmin, https://192.168.219.1:8243/services/ServiceGroupAdmin
15. ClassMediatorAdmin, ClassMediatorAdmin, https://192.168.219.1:8243/services/ClassMediatorAdmin
```

7. To see the service contract of an admin service, select the admin service's URL and then paste it in your browser with **?wsdl** at the end. For example:

<https://localhost:9443/services/UserAdmin?wsdl>

In products like WSO2 ESB and WSO2 API Manager, the port is 8243 (assuming 0 port offset). However, you should be accessing the Admin Services via the management console port, which is 9443 when there is no port offset.

8. Note that the admin service's URL appears as follows in the list you discovered in step 6:

AuthenticationAdmin, AuthenticationAdmin, https://<host IP>:8243/services/AuthenticationAdmin

### **Invoking an admin service**

Admin services are secured using common types of security protocols such as HTTP basic authentication, WS-Security username token, and session based authentication to prevent anonymous invocations. For example, the UserAdmin Web service is secured with the HTTP basic authentication. To invoke a service, you do the following:

1. Authenticate yourself and get the session cookie.
2. Generate the client stubs to access the back-end Web services.

To generate the stubs, you can write your own client program using the Axis2 client API or use an existing tool like [SoapUI](#) (4.5.1 or later) or wsdl2java.

The wsdl2java tool, which comes with WSO2 products by default hides all the complexity and presents you with a proxy to the back-end service. The stub generation happens during the project build process within the Maven POM files. It uses the Maven ant run plug-in to execute the wsdl2java tool.

You can also use the Java client program given [here](#) to invoke admin services. All dependency JAR files that you need to run this client are found in the /lib directory.

### **Authenticate the user**

The example code below authenticates the user and gets the session cookie:

```

import org.apache.axis2.AxisFault;
import org.apache.axis2.transport.http.HTTPConstants;
import org.wso2.carbon.authenticator.stub.AuthenticationAdminStub;
import org.wso2.carbon.authenticator.stub.LoginAuthenticationExceptionException;
import org.wso2.carbon.authenticator.stub.LogoutAuthenticationExceptionException;
import org.apache.axis2.context.ServiceContext;
import java.rmi.RemoteException;

public class LoginAdminServiceClient {
 private final String serviceName = "AuthenticationAdmin";
 private AuthenticationAdminStub authenticationAdminStub;
 private String endPoint;

 public LoginAdminServiceClient(String backEndUrl) throws AxisFault {
 this.endPoint = backEndUrl + "/services/" + serviceName;
 authenticationAdminStub = new AuthenticationAdminStub(endPoint);
 }

 public String authenticate(String userName, String password) throws
RemoteException,
 LoginAuthenticationExceptionException {

 String sessionCookie = null;

 if (authenticationAdminStub.login(userName, password, "localhost")) {
 System.out.println("Login Successful");

 ServiceContext serviceContext = authenticationAdminStub.
 _getServiceClient().getLastOperationContext().getServiceContext();
 sessionCookie = (String)
serviceContext.getProperty(HTTPConstants.COOKIE_STRING);
 System.out.println(sessionCookie);
 }

 return sessionCookie;
 }

 public void logOut() throws RemoteException,
LogoutAuthenticationExceptionException {
 authenticationAdminStub.logout();
 }
}

```

To resolve dependency issues, if any, add the following dependency JARs location to the class path: <PRODUCT\_HOME>/repository/components/plugins.

The the AuthenticationAdminStub class requires org.apache.axis2.context.ConfigurationContext as a parameter. You can give a null value there.

## Generate the client stubs

After authenticating the user, give the retrieved admin cookie with the service endpoint URL as shown in the sample below. The service management service name is ServiceAdmin. You can find its URL (e.g., <https://localhost:9443/services/ServiceAdmin>) in the service.xml file in the META-INF folder in the respective bundle that you find in <PRODUCT\_HOME>/repository/components/plugins.

```

import org.apache.axis2.AxisFault;
import org.apache.axis2.client.Options;
import org.apache.axis2.client.ServiceClient;
import org.wso2.carbon.service.mgt.stub.ServiceAdminStub;
import org.wso2.carbon.service.mgt.stub.types.carbon.ServiceMetaDataWrapper;
import java.rmi.RemoteException;

public class ServiceAdminClient {
 private final String serviceName = "ServiceAdmin";
 private ServiceAdminStub serviceAdminStub;
 private String endPoint;

 public ServiceAdminClient(String backEndUrl, String sessionCookie) throws AxisFault
 {
 this.endPoint = backEndUrl + "/services/" + serviceName;
 serviceAdminStub = new ServiceAdminStub(endPoint);
 //Authenticate Your stub from sessionCookie
 ServiceClient serviceClient;
 Options option;

 serviceClient = serviceAdminStub._getServiceClient();
 option = serviceClient.getOptions();
 option.setManageSession(true);
 option.setProperty(org.apache.axis2.transport.http.HTTPConstants.COOKIE_STRING,
sessionCookie);
 }

 public void deleteService(String[] serviceGroup) throws RemoteException {
 serviceAdminStub.deleteServiceGroups(serviceGroup);
 }

 public ServiceMetaDataWrapper listServices() throws RemoteException {
 return serviceAdminStub.listServices("ALL", "*", 0);
 }
}

```

The following sample code lists the back-end Web services:

```

import org.wso2.carbon.authenticator.stub.LoginAuthenticationExceptionException;
import org.wso2.carbon.authenticator.stub.LogoutAuthenticationExceptionException;
import org.wso2.carbon.service.mgt.stub.types.carbon.ServiceMetaData;
import org.wso2.carbon.service.mgt.stub.types.carbon.ServiceMetaDataWrapper;

import java.rmi.RemoteException;

public class ListServices {
 public static void main(String[] args)
 throws RemoteException, LoginAuthenticationExceptionException,
 LogoutAuthenticationExceptionException {
 System.setProperty("javax.net.ssl.trustStore",
 "$ESB_HOME/repository/resources/security/wso2carbon.jks");
 System.setProperty("javax.net.ssl.trustStorePassword", "wso2carbon");
 System.setProperty("javax.net.ssl.trustStoreType", "JKS");
 String backEndUrl = "https://localhost:9443";

 LoginAdminServiceClient login = new LoginAdminServiceClient(backEndUrl);
 String session = login.authenticate("admin", "admin");
 ServiceAdminClient serviceAdminClient = new ServiceAdminClient(backEndUrl,
session);
 ServiceMetaDataWrapper serviceList = serviceAdminClient.listServices();
 System.out.println("Service Names:");
 for (ServiceMetaData serviceData : serviceList.getServices()) {
 System.out.println(serviceData.getName());
 }

 login.logOut();
 }
}

```

## Default Ports of WSO2 Products

This page describes the default ports that are used for each WSO2 product when the port offset is 0.

- [Common ports](#)
- [Product-specific ports](#)

### **Common ports**

The following ports are common to all WSO2 products that provide the given feature. Some features are bundled in the WSO2 Carbon platform itself and therefore are available in all WSO2 products by default.

### **Management console ports**

WSO2 products that provide a management console use the following servlet transport ports:

- 9443 - HTTPS servlet transport (the default URL of the management console is <https://localhost:9443/carbon>)
- 9763 - HTTP servlet transport

### **LDAP server ports**

Provided by default in the WSO2 Carbon platform.

- 10389 - Used in WSO2 products that provide an embedded LDAP server

### **KDC ports**

- 8000 - Used to expose the Kerberos key distribution center server

### JMX monitoring ports

WSO2 Carbon platform uses TCP ports to monitor a running Carbon instance using a JMX client such as JConsole. By default, JMX is enabled in all products. You can disable it using <PRODUCT\_HOME>/repository/conf/etc/jmx.xml file.

- 11111 - RMIRegistry port. Used to monitor Carbon remotely
- 9999 - RMIServer port. Used along with the RMIRegistry port when Carbon is monitored from a JMX client that is behind a firewall

### Clustering ports

To cluster any running Carbon instance, either one of the following ports must be opened.

- 45564 - Opened if the membership scheme is multicast
- 4000 - Opened if the membership scheme is wka

### Random ports

Certain ports are randomly opened during server startup. This is due to specific properties and configurations that become effective when the product is started. Note that the IDs of these random ports will change every time the server is started.

- A random TCP port will open at server startup because of the -Dcom.sun.management.jmxremote property set in the server startup script. This property is used for the JMX monitoring facility in JVM.
- A random UDP port is opened at server startup due to the log4j appender (SyslogAppender), which is configured in the <PRODUCT\_HOME>/repository/conf/log4j.properties file.

### *Product-specific ports*

Some products open additional ports.

[API Manager](#) | [BAM](#) | [BPS](#) | [Data Analytics Server](#) | [Complex Event Processor](#) | [Elastic Load Balancer](#) | [ESB](#) | [Identity Server](#) | [Message Broker](#) | [Machine Learner](#) | [Storage Server](#) | [Enterprise Mobility Manager](#)

### **API Manager**

- 10397 - Thrift client and server ports
- 8280, 8243 - NIO/PT transport ports
- 7711 - Thrift SSL port for secure transport, where the client is authenticated to BAM/CEP: stat pub

If you change the default API Manager ports with a port offset, most of its ports will be changed automatically according to the offset except a few exceptions described in the [APIM Manager documentation](#).

### **BAM**

- 9160 - Cassandra port using which Thrift listens to clients
- 7711 - Thrift SSL port for secure transport, where the client is authenticated to BAM
- 7611 - Thrift TCP port to receive events from clients to BAM
- 21000 - Hive Thrift server starts on this port

### **BPS**

- 2199 - RMI registry port (datasources provider port)

### **Data Analytics Server**

- 9160 - Cassandra port on which Thrift listens to clients
- 7711 - Thrift SSL port for secure transport, where the client is authenticated to DAS
- 7611 - Thrift TCP port to receive events from clients to DAS
- For a list of Apache Spark related ports, see [WSO2 Data Analytics Server Documentation - Spark Configurations](#).

### **Complex Event Processor**

- 9160 - Cassandra port on which Thrift listens to clients
- 7711 - Thrift SSL port for secure transport, where the client is authenticated to CEP
- 7611 - Thrift TCP port to receive events from clients to CEP
- 11224 - Thrift TCP port for HA management of CEP

### **Elastic Load Balancer**

- 8280, 8243 - NIO/PT transport ports

### **ESB**

Non-blocking HTTP/S transport ports: Used to accept message mediation requests. If you want to send a request to an API or a proxy service for example, you must use these ports. `ESB_HOME}/repository/conf/axis2/axis2.xml` file.

- 8243 - Passthrough or NIO HTTPS transport
- 8280 - Passthrough or NIO HTTP transport

### **Identity Server**

- 8000 - KDCServerPort. Port which KDC (Kerberos Key Distribution Center) server runs
- 10500 - ThriftEntitlementReceivePort

### **Message Broker**

Message Broker uses the following JMS ports to communicate with external clients over the JMS transport.

- 5672 - Port for listening for messages on TCP when the AMQP transport is used.
- 8672 - Port for listening for messages on TCP/SSL when the AMQP Transport is used.
- 1883 - Port for listening for messages on TCP when the MQTT transport is used.
- 8833 - Port for listening for messages on TCP/SSL when the MQTT Transport is used.
- 7611 - The port for Apache Thrift Server.

### **Machine Learner**

- 7077 - The default port for Apache Spark.
- 54321 - The default port for H2O.
- 4040 - The default port for Spark UI.

### **Storage Server**

Cassandra:

- 7000 - For Inter node communication within cluster nodes
- 7001 - For inter node communication within cluster nodes via SSL
- 9160 - For Thrift client connections
- 7199 - For JMX

HDFS:

- 54310 - Port used to connect to the default file system.
- 54311 - Port used by the MapRed job tracker
- 50470 - Name node secure HTTP server port

- 50475 - Data node secure HTTP server port
- 50010 - Data node server port for data transferring
- 50075 - Data node HTTP server port
- 50020 - Data node IPC server port

### **Enterprise Mobility Manager**

The following ports need to be opened for Android and iOS devices so that it can connect to Google Cloud Messaging (GCM)/Firebase Cloud Messaging (FCM) and APNS (Apple Push Notification Service) and enroll to WSO2 EMM.

Android:

The ports to open are 5228, 5229 and 5230. GCM/FCM typically only uses 5228, but it sometimes uses 5229 and 5230.

GCM/FCM does not provide specific IPs, so it is recommended to allow the firewall to accept outgoing connections to all IP addresses contained in the IP blocks listed in Google's ASN of 15169.

iOS:

- 5223 - TCP port used by devices to communicate to APNs servers
- 2195 - TCP port used to send notifications to APNs
- 2196 - TCP port used by the APNs feedback service
- 443 - TCP port used as a fallback on Wi-Fi, only when devices are unable to communicate to APNs on port 5223

The APNs servers use load balancing. The devices will not always connect to the same public IP address for notifications. The entire 17.0.0.0/8 address block is assigned to Apple, so it is best to allow this range in the firewall settings.

API Manager:

The following WSO2 API Manager ports are only applicable to WSO2 EMM 1.1.0 onwards.

- 10397 - Thrift client and server ports
- 8280, 8243 - NIO/PT transport ports

## **ESB Tools**

The following tools are available in the WSO2 ESB:

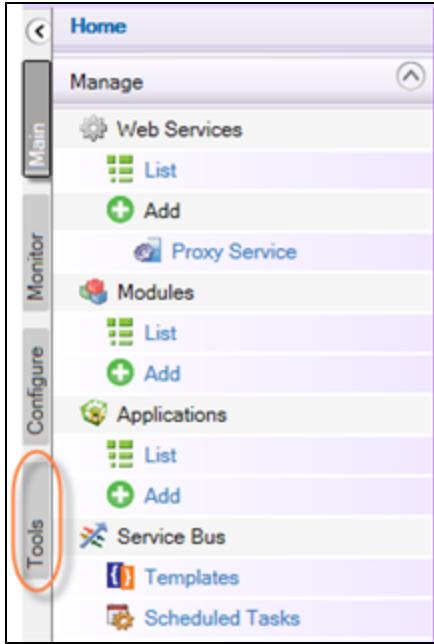
- [WSDL2Java](#) - Generates Java code to provide or consume a Web service for a given WSDL.
- [Java2WSDL](#) - Generates the WSDL for Java code that is already written to provide a Web service.
- [Try It](#) - Allows to try out a custom Web Services Definition Language (WSDL) or any publicly available WSDL of the document/literal style.
- [WSDL Validator](#) - Validates a WSDL document and prints the result in the "Validation Result" section.

### **WSDL2Java**

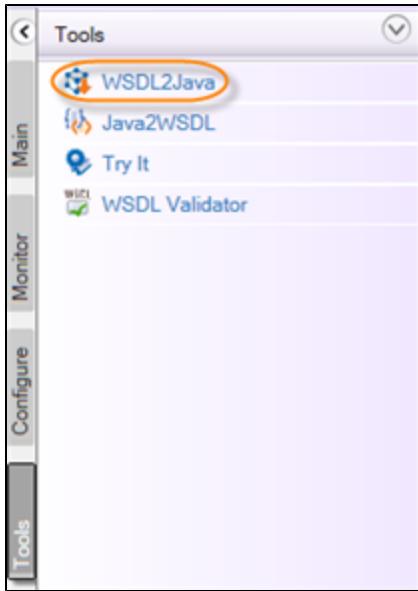
The "**WSDL2Java**" tool is used to generate Java code to provide or consume a Web service for a given WSDL. You can use this tool to develop Web services for contract first development. The tool provides many options to customize your code.

Follow the instructions below to use the "WSDL2Java" tool in WSO2 ESB.

1. Sign in. Enter your user name and password to log on to the ESB Management Console.
2. Click the "Tools" button to access the "Tools" menu.



3. From the "Tools" menu, select "WSDL2Java."



4. The "WSDL2Java" page appears. Fill in the fields as required.

### Note

Each option is described on the "WSDL2Java" page.

### WSDL2Java

**WSDL2Code Options**

Option	Description	Select/Type Value
-uri *	A url or path to a WSDL	<input type="text"/> ...
-a	Generate async style code only (Default: off)	<input type="checkbox"/>
-s	Generate sync style code only (Default: off). Takes precedence over -a	<input type="checkbox"/>
-p	Specify a custom package name for the generated code	<input type="text"/>
-l	Valid languages	java <input type="button" value="▼"/>
-t	Generate a test case for the generated code	<input type="checkbox"/>
-ss	Generate server side code (i.e. skeletons)	<input type="checkbox"/>
-sd	Generate service descriptor (i.e. services.xml). Valid with -ss	<input type="checkbox"/>
-d	Valid databinding(s)	adb <input type="button" value="▼"/>
-g	Generates all the classes. Valid only with -ss	<input type="checkbox"/>
-pn	Port name. Choose a specific port when there are multiple ports in the wsdl	<input type="text"/>
-sn	Service name. Choose a specific service when there are multiple services in the wsdl	<input type="text"/>
-u	Unpacks the databinding classes	<input type="checkbox"/>
-r	Specify a repository against which code is generated	<input type="text"/>
-ns2p	ns1=pkg1.ns2=pkg2 .Specify a custom package name for each namespace specified in the wsdl schema	<input type="text"/>
-ssi	Generate an interface for the service implementation	<input type="checkbox"/>
-wv	WSDL Version	1.1 <input type="button" value="▼"/>
-uw	Switch on un-wrapping	<input type="checkbox"/>
-xsdconfig	Use XMLBeans .xsdconfig file. Valid only with -d xmlbeans	<input type="text"/> ...
-ap	Generate code for all ports	<input type="checkbox"/>
-or	Overwrite the existing classes	<input type="checkbox"/>
-Emp	Package name for ADB. Extension mapper package name	<input type="text"/>
-Eosv	(for ADB) - off strict validation	<input type="checkbox"/>
-Ebindingfile	File path to your JIBX binding definition	<input type="text"/> ...

**Generate >**

5. Enter the URL of the WSDL in the the "uri" field.

### Tip

Entering a value in this field is mandatory.

**WSDL2Code Options**

Option	Description	Select/Type Value
-uri *	A url or path to a WSDL	<input type="text" value="/home/urssl/myWSDL"/> ...

6. Click "Generate."



You can now call this stub object from your client Java code. Note that when you instantiate a stub object generated from WSDL2java without passing any parameters, the underling Axis2 runtime creates a configuration context for the new stub object. Therefore, when initializing multiple stub objects, many configuration context objects are

created, which slows performance of the client. To avoid performance problems, create a Configuration context at the beginning of your client code and pass that same context to all stub objects. For example:

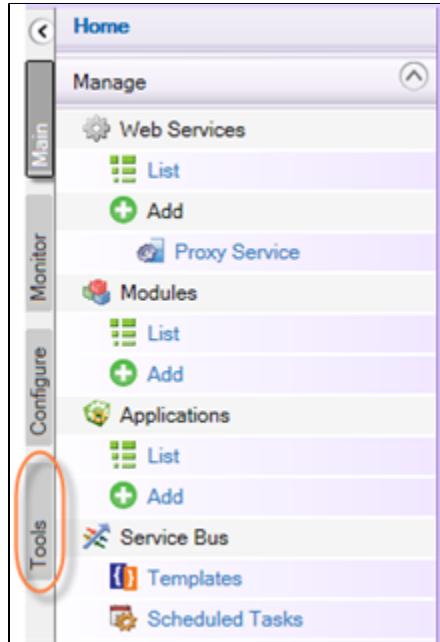
```
ConfigurationContext configurationContext =
ConfigurationContextFactory.createConfigurationContextFromFileSystem("repository",
"axis2.xml");
CustomerOrderServiceStub stub = new CustomerOrderServiceStub(configurationContext, "h
ttp://localhost:8088/services/CustomerOrderService");
```

## Java2WSDL

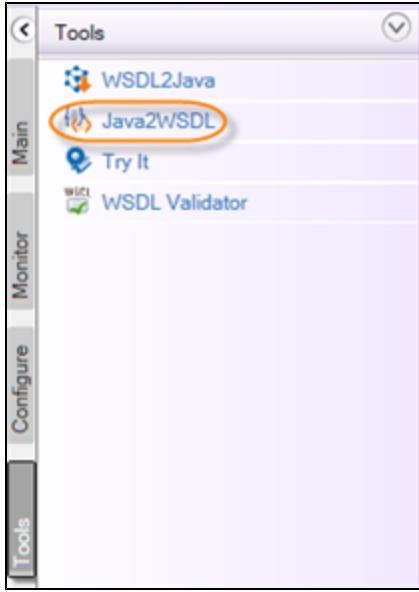
The "**Java2WSDL**" tool allows you to generate the WSDL for Java code that is already written to provide a Web service. This tool helps you to develop Web services in the code first approach, where the development starts with the code and thereafter you derive the WSDL from the source.

Follow the instructions below to use the "Java2WSDL" tool in the WSO2 ESB.

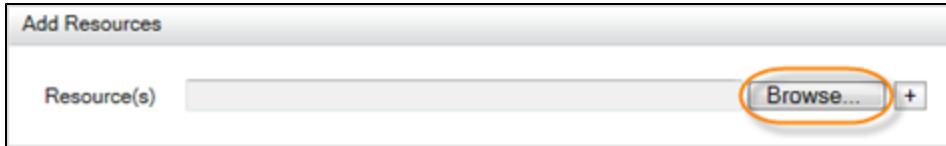
1. Sign in. Enter your user name and password to log on to the ESB Management Console.
2. Click the "Tools" button to access the "Tools" menu.



3. From the "Tools" menu, select "Java2WSDL."



4. Click "Browse" to locate the file you want to add.



### Tip

If you want to add more than one file, click the plus sign button.

5. Enter values as required.

### Note

Each field is described in the user interface.

## Select Options

Option	Description	Select/Type Value
cn*	Fully qualified class name	<input type="text"/>
sn	Service name	<input type="text"/>
	Address of the port for the WSDL	<input type="text"/>
tn	Target namespace for service	<input type="text"/>
tp	Target namespace prefix for service	<input type="text"/>
stn	Target namespace for schema	<input type="text"/>
stp	Target namespace prefix for schema	<input type="text"/>
st	Style for the WSDL	document <input type="button" value="▼"/>
u	Use for the WSDL	literal <input type="button" value="▼"/>
nsg	Fully qualified name of a class that implements NamespaceGenerator	<input type="text"/>
sg	Fully qualified name of a class that implements SchemaGenerator	<input type="text"/>
p2n	[java package,namespace][java package,namespace]...package to namespace mapping	<input type="text"/>
efd	Setting for elementFormDefault (defaults to qualified)	qualified <input type="button" value="▼"/>
afd	Setting for attributeFormDefault (defaults to qualified)	qualified <input type="button" value="▼"/>
xc	class1,class2,class3,...Extra class(es) for which schematype must be generated	<input type="text"/>
wv	WSDL version - defaults to 1.1	1.1 <input type="button" value="▼"/>
dlb	Generate schemas conforming to doc/lit/bare style	<input type="checkbox"/>

6. Click "Generate."



Try It

The "Try It" tool is a tool available with the WSO2 Carbon that allows you to try out your own Web Services Definition Language (WSDL) or any publicly available WSDL of the document/literal style.

"Try It" provides you with a mechanism to test your WSDL by creating endpoints on-the-fly. It helps to test a WSDL before actually coding it, without the need for a third party WSDL validator tool.

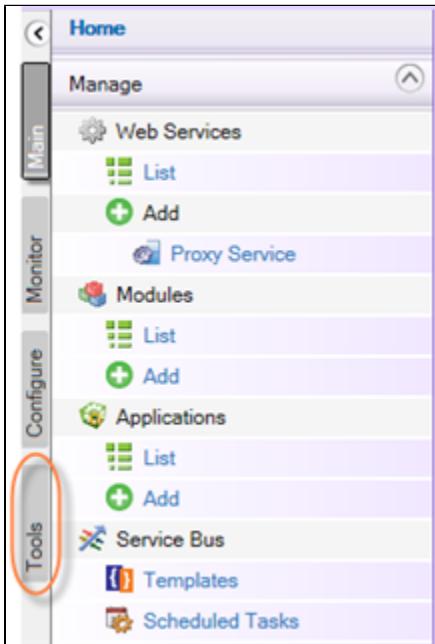
### Note

The "Try It" tool does not support relative schema imports and WSDL imports at the moment. We will provide this feature in a future release.

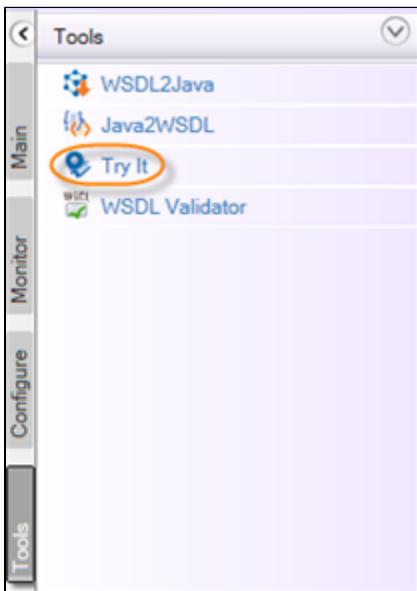
Follow the instructions below to use the "Try It" tool in the WSO2 ESB.

1. Sign in. Enter your user name and password to log on to the ESB Management Console.

2. Click the "Tools" button to access the "Tools" menu.



3. From the "Tools" menu, select "Try It."



4. In the "Enter URL" field, type or paste the location of the WSDL. It may be a local file system path or a web URL.

The screenshot shows the 'Try It' page. At the top, it says 'Try It'. Below that is a text input field labeled 'Enter WSDL 1.1 or 2.0 Document Location'. Underneath is another text input field labeled 'Enter URL' (which is highlighted with a red circle). At the bottom is a blue 'Try It' button.

5. Click the "Try It" button.



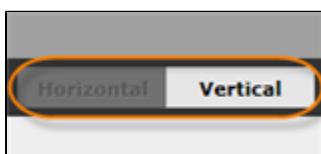
## Note

If your browser has Pop up-Blocking enabled, you will receive a message saying a pop up was blocked. Allow pop ups for the current domain and try again. The available operations in the given WSDL appears. For operations that take arguments you will see primitive argument type fields. The values specified in these fields will be passed to the operations. For no-argument operations you will only see the button that has the same name as the operation.

6. Click the "Send" button to invoke that service operation. The return value immediately appears in the response textarea.

The screenshot shows the WSO2 ESB 'Send' interface. On the left, there's a sidebar with buttons for 'Priority Operations', 'All Operations', and 'mediate'. The 'mediate' button is selected and highlighted. To its right is a tree view with 'PS2' expanded, showing '+ Using Endpoint - PS2HttpSoap12Endpoint'. Below this is a note: '⚠ Private proxy protocol will be attempted as cross-domain browser restrictions might be enforced for this endpoint. [Hide](#)'. Underneath is a link 'Try an alternate http [Hide](#)'. The main area is titled 'mediate' and contains a 'Send' button. Below it are two large text areas: 'Request' and 'Response'. Each text area has a toolbar at the top and a status bar at the bottom showing character counts. At the very bottom is a 'Send' button. Above the textareas are 'Horizontal' and 'Vertical' layout buttons, with 'Horizontal' being the active one.

Two text areas can be seen, each for request and response. You can switch the layout using either "Horizontal" or "Vertical" buttons.



## Choosing Endpoints

You can change the endpoint for the service, if there are multiple endpoints. You can also specify a customized endpoint.

1. Click the "Using Endpoint - ...." link.

The screenshot shows the 'Using Endpoint - echoHttpSoap12Endpoint' configuration screen. At the top, there's a link 'Using Endpoint - echoHttpSoap12Endpoint' which is highlighted with a yellow oval. Below it is a section titled 'Select a different Endpoint'. It contains a dropdown menu labeled 'Select an endpoint : echoHttpSoap12Endpoint' and a text input field 'Change the address for the selected endpoint : http://localhost:8280/services/echo.echoHttpSoap12Endpoint'.

2. Select a new endpoint from the list.

### - Using Endpoint - echoHttpSoap12Endpoint

Select a different Endpoint

Select an endpoint : echoHttpSoap12Endpoint ▾

Change the address for the selected endpoint : http://localhost:8280/services/echo.echoHttpSoap12Endpoint

3. Specify the address of the new endpoint.

### - Using Endpoint - echoHttpSoap12Endpoint

Select a different Endpoint

Select an endpoint : echoHttpSoap12Endpoint ▾

Change the address for the selected endpoint : http://localhost:8280/services/echo.echoHttpSoap12Endpoint

## Viewing the Service Information

1. To view the information about the service click on the "Service Information" link.

The screenshot shows a user interface for managing a service. At the top, there's a header bar with a dropdown menu and some other controls. Below it, a main area has a title 'Service Infomation' which is highlighted with a red oval. Underneath this, there's a section labeled 'echo'.

## Prioritizing Operations

1. Use the small yellow icon with the plus(+) sign to prioritize operation.

The screenshot shows a list of operations under a 'Priority Operations' section. The 'echoOMElement' operation is highlighted with a red oval and has a yellow plus sign icon next to it, indicating it is prioritized. Other operations listed include 'echoInt', 'echoString', 'echoStringArrays', and 'throwAxisFault'. Each operation has a yellow plus sign icon to its right.

2. The chosen operation is added to the "Priority Operations" section.

This screenshot is similar to the previous one, but the 'echoOMElement' operation is now at the top of the list, indicating it has been moved to the 'Priority Operations' section. The other operations ('echoInt', 'echoString', 'echoStringArrays', 'throwAxisFault') are still present below it with their respective priority icons.

This will be useful when you have a lot of operations in the Try It page. Then you can add operations you want into this section and switch among them easily.

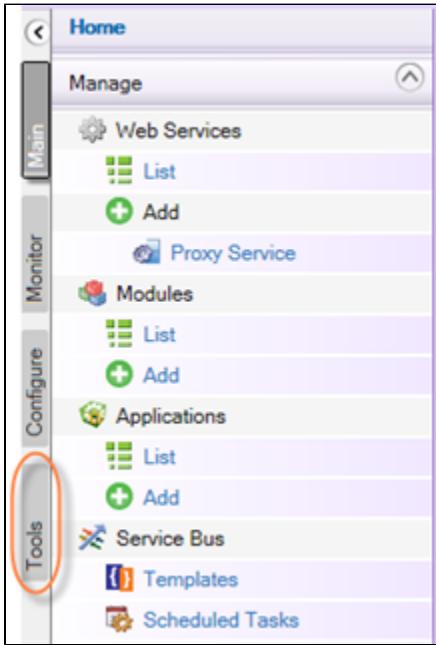
## WSDL Validator

The "**WSDL Validator**" tool can be used to validate a WSDL document. You can upload your WSDL or you can provide a WSDL URL. The tool will validate your WSDL and print the result in the "Validation Result" section.

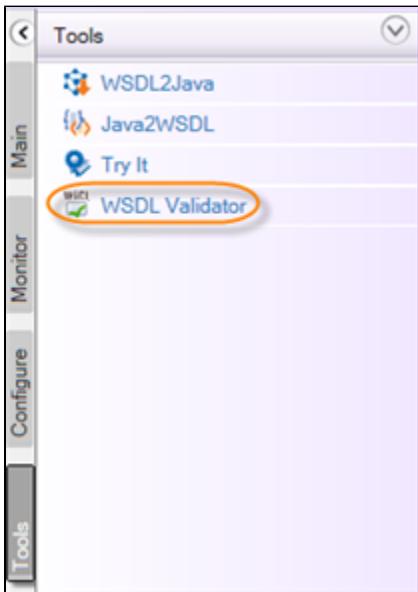
Follow the instructions below to use the "WSDL Validator" tool in the WSO2 ESB.

1. Sign in. Enter your user name and password to log on to the ESB Management Console.

2. Click the "Tools" button to access the "Tools" menu.



3. From the "Tools" menu, select "WSDL Validator."



4. The "WSDL Validator" page appears.

**WSDL Validator**

<b>Select WSDL File</b>	
WSDL File Location	<input type="text"/> <input type="button" value="Browse..."/>
<input type="button" value="Validate From File"/>	
<b>Provide WSDL URL</b>	
WSDL URL	<input type="text"/>
<input type="button" value="Validate From URL"/>	

### Uploading a WSDL

1. You can upload the WSDL document to be validated using the following user interface. Click on the "Browse" button to upload a document.

<b>Select WSDL File</b>	
WSDL File Location	<input type="text"/> <input type="button" value="Browse..."/>
<input type="button" value="Validate From File"/>	

2. Click on the "Validate From File" button.

<input type="button" value="Validate From File"/>
---------------------------------------------------

### Providing a WSDL URL

1. You can provide the URL of your WSDL document to be validated using the following user interface.

<b>Provide WSDL URL</b>	
WSDL URL	<input type="text"/>
<input type="button" value="Validate From URL"/>	

2. Click on the "Validate From URL" button.

<input type="button" value="Validate From URL"/>
--------------------------------------------------

### Validation Result

Validation result will be shown as follows.

**Validation Result**

```
[0][1]The 'HelloServiceHttpsSoap12Endpoint' port has an invalid binding - 'HelloServiceSoap12BindingFoo'. Check that the 'HelloServiceSoap12BindingFoo' binding is defined.*****invalid*****
```

**Properties Reference**

**Properties** provide the means of accessing various types of information regarding a message that passes through the ESB. You can also use properties to control the behavior of the ESB on a given message. Following are the types of properties you can use:

- **Generic Properties**: Allow you to configure messages as they're processed by the ESB, such as marking a message as out-only (no response message will be expected), adding a custom error message or code to the message, and disabling WS-Addressing headers.
- **HTTP Transport Properties**: Allow you to configure how the HTTP transport processes messages, such as forcing a 202 HTTP response to the client so that it stops waiting for a response, setting the HTTP status code, and appending a context to the target URL in RESTful invocations.
- **SOAP Headers**: Provide information about the message, such as the To and From values.
- **Axis2 Properties**: Allow you to configure the web services engine in the ESB, such as specifying how to cache JMS objects, setting the minimum and maximum threads for consuming messages, and forcing outgoing HTTP/S messages to use HTTP 1.0.
- **Synapse Message Context Properties**: Allow you to get information about the message, such as the date/time it was sent, the message format, and the message operation.

For many properties, you can use the [Property mediator](#) to retrieve and set the properties. Additionally, see [Accessing Properties with XPath](#) for information on the XPath extension functions and Synapse XPath variables you can use.

**Generic Properties**

```
[PRESERVE_WS_ADDRESSING][RESPONSE][OUT_ONLY][ERROR_CODE][ERROR_MESSAGE][ERROR_DETAIL][ERROR_EXCEPTION][TRANSPORT_HEADERS][messageType][ContentType][disableAddressingForOutMessages][DISABLE_SMOOKS_RESULT_PAYLOAD][ClientApiNonBlocking][transportNonBlocking][TRANSPORT_IN_NAME][preserveProcessedHeaders][SERVER_IP][FORCE_ERROR_ON_SOAP_FAULT]
```

Generic properties allow you to configure messages as they're processed by the ESB, such as marking a message as out-only (no response message will be expected), adding a custom error message or code to the message, and disabling WS-Addressing headers.

**PRESERVE\_WS\_ADDRESSING**

<b>Name</b>	PRESERVE_WS_ADDRESSING
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	none
<b>Scope</b>	synapse
<b>Description</b>	By default, the ESB adds a new set of WS-Addressing headers to the messages forwarded from the ESB. If this property is set to "true" on a message, the ESB will forward it without altering its existing WS-Addressing headers.

<b>Example</b>	<pre>&lt;property name="PRESERVE_WS_ADDRESSING" value="true" /&gt;</pre>
----------------	--------------------------------------------------------------------------

**RESPONSE**

<b>Name</b>	RESPONSE
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	none
<b>Scope</b>	synapse
<b>Description</b>	Once this property is set to 'true' on a message, the ESB will start treating it as a response message. It is generally used to route a request message back to its source as the response.
<b>Example</b>	<pre>&lt;property name="RESPONSE" value="true" /&gt;</pre>

**OUT\_ONLY**

<b>Name</b>	OUT_ONLY
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	none
<b>Scope</b>	synapse
<b>Description</b>	Set this property to "true" on a message to indicate that no response message is expected for it once it is forwarded from the ESB. In other words, the ESB will do an out-only invocation with such messages. It is very important to set this property on messages that are involved in out-only invocations to prevent the ESB from registering unnecessary callbacks for response handling and eventually running out of memory.
<b>Example</b>	<pre>&lt;property name="OUT_ONLY" value="true" /&gt;</pre>

**ERROR\_CODE**

<b>Name</b>	ERROR_CODE
<b>Possible Values</b>	string
<b>Default Behavior</b>	none

<b>Scope</b>	synapse
<b>Description</b>	Use this property to set a custom error code on a message which can be later processed by a Synapse fault handler. If the Synapse encounters an error during mediation or routing, this property will be automatically populated.
<b>Example</b>	<pre>&lt;property name="ERROR_CODE" value="100100"/&gt;</pre>

### ***ERROR\_MESSAGE***

<b>Name</b>	ERROR_MESSAGE
<b>Possible Values</b>	string
<b>Default Behavior</b>	none
<b>Scope</b>	synapse
<b>Description</b>	Use this property to set a custom error message on a message which can be later processed by a Synapse fault handler. If the Synapse encounters an error during mediation or routing, this property will be automatically populated.
<b>Example</b>	<pre>&lt;log level="custom"&gt;     &lt;property name="Cause" expression="get-property('ERROR_MESSAGE')"/&gt; &lt;/log&gt;</pre>

### ***ERROR\_DETAIL***

<b>Name</b>	ERROR_DETAIL
<b>Possible Values</b>	string
<b>Default Behavior</b>	none
<b>Scope</b>	synapse
<b>Description</b>	Use this property to set the exception stacktrace in case of an error. If the ESB encounters an error during mediation or routing, this property will be automatically populated.
<b>Example</b>	<pre>&lt;log level="custom"&gt;     &lt;property name="Trace" expression="get-property('ERROR_DETAIL')"/&gt; &lt;/log&gt;</pre>

### ***ERROR\_EXCEPTION***

<b>Name</b>	ERROR_EXCEPTION
<b>Possible Values</b>	java.lang.Exception
<b>Default Behavior</b>	none
<b>Scope</b>	synapse
<b>Description</b>	Contains the actual exception thrown in case of a runtime error.

### ***TRANSPORT\_HEADERS***

<b>Name</b>	TRANSPORT_HEADERS
<b>Possible Values</b>	java.util.Map
<b>Default Behavior</b>	Populated with the transport headers of the incoming request.
<b>Scope</b>	axis2
<b>Description</b>	Contains the map of transport headers. Automatically populated. Individual values of this map can be accessed using the property mediator in the transport scope.
<b>Example</b>	<pre>&lt;property name="TRANSPORT_HEADERS" action="remove" scope="axis2"/&gt;</pre>

### ***messageType***

<b>Name</b>	messageType
<b>Possible Values</b>	string
<b>Default Behavior</b>	Content type of incoming request.
<b>Scope</b>	axis2
<b>Description</b>	Message formatter is selected based on this property. This property should have the content type, such as text/xml, application/xml, or application/json.
<b>Example</b>	<pre>&lt;property name="messageType" value="text/xml" scope="axis2"/&gt;</pre>

### ***ContentType***

<b>Name</b>	ContentType
<b>Possible Values</b>	string

<b>Default Behavior</b>	Value of the Content-type header of the incoming request.
<b>Scope</b>	axis2
<b>Description</b>	This property will be in effect only if the messageType property is set. If the messageType is set, the value of Content-Type HTTP header of the outgoing request will be chosen based on this property. Note that this property is required to be set only if the message formatter seeks it in the message formatter implementation.
<b>Example</b>	<property name="ContentType" value="text/xml" scope="axis2"/>

***disableAddressingForOutMessages***

<b>Name</b>	disableAddressingForOutMessages
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	false
<b>Scope</b>	axis2
<b>Description</b>	Set this property to "true" if you do not want the ESB to add WS-Addressing headers to outgoing messages. This property can affect messages sent to backend services as well as the responses routed back to clients.
<b>Example</b>	<property name="disableAddressingForOutMessages" value="true" scope="axis2"/>

***DISABLE\_SMOOKS\_RESULT\_PAYLOAD***

<b>Name</b>	DISABLE_SMOOKS_RESULT_PAYLOAD
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	false
<b>Scope</b>	synapse
<b>Description</b>	If this property is set to true, the result of file content processing carried out by the <a href="#">Smooks Mediator</a> will not be loaded into the message context. This is useful in situations where you want to avoid large memory growth/out of heap space issue that may occur when large files processed by the Smooks mediator are reprocessed. See <a href="#">VFS Transport</a> for a proxy service configuration where this property is used.

<b>Example</b>	<pre>&lt;property name="DISABLE_SMOOKS_RESULT_PAYLOAD"      value="true" scope="default"          type="STRING" /&gt;</pre>
----------------	-----------------------------------------------------------------------------------------------------------------------------

***ClientApiNonBlocking***

<b>Name</b>	ClientApiNonBlocking
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	true
<b>Scope</b>	axis2
<b>Description</b>	<p>By default, Axis2 will spawn a new thread to handle each outgoing message. To change this behavior, remove this property from the message. Removal of this property could be vital when queuing transports like JMS are involved.</p> <p>A VFS proxy that writes to a VFS endpoint should have this property set in order to hold the primary thread until the send happens.</p>
<b>Example</b>	<pre>&lt;property name="ClientApiNonBlocking" action="remove" scope="axis2" /&gt;</pre>

***transportNonBlocking***

<b>Name</b>	transportNonBlocking
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	true
<b>Scope</b>	axis2
<b>Description</b>	<p>This property works the same way as ClientApiNonBlocking. It is recommended to use ClientApiNonBlocking for this purpose instead of transportNonBlocking since the former uses the latest axis2 translations.</p>
<b>Example</b>	<pre>&lt;property name="transportNonBlocking" action="remove" scope="axis2" value="true" /&gt;</pre>

***TRANSPORT\_IN\_NAME***

<b>Name</b>	TRANSPORT_IN_NAME
<b>Scope</b>	synapse

<b>Description</b>	Mediation logic can read incoming transport name using this property (since WSO2 ESB 4.7.0)
<b>Example</b>	<pre>&lt;log level="custom"&gt;     &lt;property name="INCOMING_TRANSPORT" expression="get-property('TRANSPORT_IN_NAME')"/&gt; &lt;/log&gt;</pre>

***preserveProcessedHeaders***

<b>Name</b>	preserveProcessedHeaders
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	Preserving SOAP headers
<b>Scope</b>	synapse(default)
<b>Description</b>	By default, Synapse removes the SOAP headers of incoming requests that have been processed. If we set this property to 'true', Synapse preserves the SOAP headers.
<b>Example</b>	<pre>&lt;property name="preserveProcessedHeaders" value="true" scope="default"/&gt;</pre>

***SERVER\_IP***

<b>Name</b>	SERVER_IP
<b>Possible Values</b>	IP address or hostname of the ESB host
<b>Default Behavior</b>	Set automatically by the mediation engine upon startup
<b>Scope</b>	synapse

***FORCE\_ERROR\_ON\_SOAPFAULT***

<b>Name</b>	FORCE_ERROR_ON_SOAPFAULT
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	true
<b>Scope</b>	synapse(default)
<b>Description</b>	When a SOAP error occurs in a response, the SOAPFault sent from the back end is received by the out sequence as a usual response by default. If this property is set to true, the SOAPFault is redirected to a fault sequence. Note that when this property is true, only properties in the 'operation' scope will be passed to the error handler, and other properties in the axis2 or default scopes will not be passed to the error handler.

**Example**

```
<property name="FORCE_ERROR_ON_SOAP_FAULT" value="true" scope="default"
type="STRING"></property>
```

**HTTP Transport Properties**

[ POST\_TO\_URI ] [ FORCE\_SC\_ACCEPTED ] [ DISABLE\_CHUNKING ] [ NO\_ENTITY\_BODY ] [ FORCE\_HTTP\_1.0 ] [ HTTP\_SC ] [ FAULTS\_AS\_HTTP\_200 ] [ NO\_KEEPALIVE ] [ REST\_URL\_POSTFIX ] [ REQUEST\_HOST\_HEADER ] [ FORCE\_POST\_PUT\_NOBODY ] [ FORCE\_HTTP\_CONTENT\_LENGTH ] [ COPY\_CONTENT\_LENGTH\_FROM\_INCOMING ]

HTTP transport properties allow you to configure how the HTTP transport processes messages, such as forcing a 202 HTTP response to the client so that it stops waiting for a response, setting the HTTP status code, and appending a context to the target URL in RESTful invocations.

**POST\_TO\_URI**

<b>Name</b>	<b>POST_TO_URI</b>
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	false
<b>Scope</b>	axis2
<b>Description</b>	This property makes the request URL that is sent from ESB a complete URL. When set to <code>false</code> only the context path will be included in the request URL that is sent. It is important that this property is set to <code>true</code> when ESB needs to communicate with the back-end service through a proxy server.
<b>Example</b>	<pre>&lt;property name="POST_TO_URI" scope="axis2" value="true"/&gt;</pre>

**FORCE\_SC\_ACCEPTED**

<b>Name</b>	FORCE_SC_ACCEPTED
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	false
<b>Scope</b>	axis2
<b>Description</b>	When set to true, this property forces a 202 HTTP response to the client so that it stops waiting for a response.
<b>Example</b>	<pre>&lt;property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/&gt;</pre>

***DISABLE\_CHUNKING***

<b>Name</b>	DISABLE_CHUNKING
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	false
<b>Scope</b>	axis2
<b>Description</b>	Disables the HTTP chunking for outgoing messaging.
<b>Example</b>	<property name="DISABLE_CHUNKING" value="true" scope="axis2"/>

***NO\_ENTITY\_BODY***

<b>Name</b>	NO_ENTITY_BODY
<b>Possible Values</b>	none
<b>Default Behavior</b>	In case of GET and DELETE requests this property is set to true.
<b>Scope</b>	axis2
<b>Description</b>	This property should be set if a user wants to generate a response from the ESB to a request without an entity body, for example, GET request.  If using the <a href="#">PayloadFactory mediator</a> , this property does not need to be manually set since it is done automatically by the mediator.
<b>Example</b>	<property name="NO_ENTITY_BODY" action="remove" scope="axis2"/>

***FORCE\_HTTP\_1.0***

<b>Name</b>	FORCE_HTTP_1.0
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	false
<b>Scope</b>	axis2
<b>Description</b>	Force HTTP 1.0 for outgoing HTTP messages.
<b>Example</b>	<property name="FORCE_HTTP_1.0" value="true" scope="axis2"/>

**HTTP\_SC**

<b>Name</b>	HTTP_SC
<b>Possible Values</b>	HTTP status code number
<b>Default Behavior</b>	none
<b>Scope</b>	axis2
<b>Description</b>	Set the HTTP status code.
<b>Example</b>	<property name="HTTP_SC" value="500" scope="axis2"/>

**FAULTS\_AS\_HTTP\_200**

<b>Name</b>	FAULTS_AS_HTTP_200
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	false
<b>Scope</b>	axis2
<b>Description</b>	When ESB receives a soap fault as a HTTP 500 message, ESB will forward this fault to client with status code 200.
<b>Example</b>	<property name="FAULTS_AS_HTTP_200" value="true" scope="axis2"/>

**NO\_KEEPALIVE**

<b>Name</b>	NO_KEEPALIVE
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	false
<b>Scope</b>	axis2
<b>Description</b>	Disables HTTP keep alive for outgoing requests.
<b>Example</b>	<property name="NO_KEEPALIVE" value="true" scope="axis2"/>

**REST\_URL\_POSTFIX**

<b>Name</b>	REST_URL_POSTFIX
-------------	------------------

<b>Possible Values</b>	A URL fragment starting with "/"
<b>Default Behavior</b>	In the case of GET requests through an address endpoint, this contains the query string.
<b>Scope</b>	axis2
<b>Description</b>	The value of this property will be appended to the target URL when sending messages out in a RESTful manner through an address endpoint. This is useful when you need to append a context to the target URL in case of RESTful invocations. If you are using an HTTP endpoint instead of an address endpoint, specify variables in the format of "uri.var.*" instead of using this property.
<b>Example</b>	<pre>&lt;property name="REST_URL_POSTFIX" value="/context" scope="axis2"/&gt;</pre>

### ***REQUEST\_HOST\_HEADER***

<b>Name</b>	REQUEST_HOST_HEADER
<b>Possible Values</b>	string
<b>Default Behavior</b>	ESB will set hostname of target endpoint and port as the HTTP host header
<b>Scope</b>	axis2
<b>Description</b>	The value of this property will be set as the HTTP host header of outgoing request
<b>Example</b>	<pre>&lt;property name="REQUEST_HOST_HEADER" value="www.wso2.org" scope="axis2"/&gt;</pre>

### ***FORCE\_POST\_PUT\_NOBODY***

<b>Name</b>	FORCE_POST_PUT_NOBODY
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	false
<b>Scope</b>	axis2
<b>Description</b>	This property allows to send a request without a body for POST and PUT HTTP methods. Applicable only for HTTP Passthrough transport.

**Example**

```
<property name="FORCE_POST_PUT_NOBODY" value="true" scope="axis2"
type="BOOLEAN" />
```

**FORCE\_HTTP\_CONTENT\_LENGTH**

<b>Name</b>	FORCE_HTTP_CONTENT_LENGTH
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	false
<b>Scope</b>	axis2
<b>Description</b>	<p>This property allows the content length to be sent when the ESB sends a request to a back end server. When HTTP 1.1 is used, this property disables chunking and sends the content length. When HTTP 1.0 is used, the property only sends the content length.</p> <p>This property should be set in scenarios where the backend server is not able to accept chunked content. For example, in a scenario where a pass-through proxy is defined and the backend does not accept chunked content, this property should be used together with the <a href="#">COPY_CONTENT_LENGTH_FROM_INCOMING</a> property, to simply add the content length without chunking.</p> <div style="border: 1px solid red; padding: 5px; margin-top: 10px;"> <p>This property can cause performance degradation. It should only be used with message relay.</p> </div>
<b>Example</b>	<pre>&lt;property name="FORCE_HTTP_CONTENT_LENGTH" scope="axis2" value="true"&gt;&lt;/property&gt;</pre>

**COPY\_CONTENT\_LENGTH\_FROM\_INCOMING**

<b>Name</b>	COPY_CONTENT_LENGTH_FROM_INCOMING
<b>Possible Values</b>	"true", "false"
<b>Default Behavior</b>	false
<b>Scope</b>	axis2
<b>Description</b>	This property allows the HTTP content length to be copied from an incoming message. It is only valid when the FORCE_HTTP_CONTENT_LENGTH property is used. The COPY_CONTENT_LENGTH_FROM_INCOMING avoids buffering the message in memory for calculating the content length, thus reducing the risk of performance degradation.

**Example**

```
<property name="COPY_CONTENT_LENGTH_FROM_INCOMING" value="true"
scope="axis2" />
```

**SOAP Headers**

[ [To](#) ] [ [From](#) ] [ [Action](#) ] [ [ReplyTo](#) ] [ [MessageID](#) ] [ [RelatesTo](#) ] [ [FaultTo](#) ]

SOAP headers provide information about the message, such as the To and From values. You can use the `get-property()` function in the [Property mediator](#) to retrieve these headers. You can also add [Custom SOAP Headers](#) using the [PayloadFactory mediator](#).

**To**

<b>Header Name</b>	To
<b>Possible Values</b>	Any URI
<b>Description</b>	The To header of the message.
<b>Example</b>	<code>get-property("To")</code>

**From**

<b>Header Name</b>	From
<b>Possible Values</b>	Any URI
<b>Description</b>	The From header of the message.
<b>Example</b>	<code>get-property("From")</code>

**Action**

<b>Header Name</b>	Action
<b>Possible Values</b>	Any URI
<b>Description</b>	The SOAPAction header of the message.
<b>Example</b>	<code>get-property("Action")</code>

**ReplyTo**

<b>Header Name</b>	ReplyTo
<b>Possible Values</b>	Any URI
<b>Description</b>	The ReplyTo header of the message.
<b>Example</b>	<code>&lt;header name="ReplyTo" action="remove"/&gt;</code>

**MessageID**

<b>Header Name</b>	MessageID
<b>Possible Values</b>	UUID
<b>Description</b>	The unique message ID of the message. It is not recommended to make alterations to this property of a message.
<b>Example</b>	get-property("MessageID")

#### RelatesTo

<b>Header Name</b>	RelatesTo
<b>Possible Values</b>	UUID
<b>Description</b>	The unique ID of the request to which the current message is related. It is not recommended to make changes.
<b>Example</b>	get-property("RelatesTo")

#### FaultTo

<b>Header Name</b>	FaultTo
<b>Possible Values</b>	Any URI
<b>Description</b>	The FaultTo header of the message.
<b>Example</b>	<header name="FaultTo" value="http://localhost:9000"/>

#### Custom SOAP Headers

Custom SOAP headers can be added to a request by using the [PayloadFactory Mediator](#) in the proxy service as shown in the example below.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
<proxy name="StockQuoteProxy"
transports="https http"
startOnLoad="true"
trace="disable">
<description/>
<target>
<endpoint>
<address uri="http://localhost:9001/services/SimpleStockQuoteService"/>
</endpoint>
<inSequence>
<log level="full"/>
<payloadFactory media-type="xml">
<format>
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
```

```
xmlns:xsd="http://services.samples/xsd"
xmlns:ser="http://services.samples">
<soapenv:Header>
<ser:authenticationRequest>
<userName xmlns="">$1</userName>
<password xmlns="">$2</password>
</ser:authenticationRequest>
</soapenv:Header>
<soapenv:Body>
<ser:getQuote>
<ser:request>
<xsd:symbol>$3</xsd:symbol>
</ser:request>
</ser:getQuote>
</soapenv:Body>
</soapenv:Envelope>
</format>
<args>
<arg value="punnadi"/>
<arg value="password"/>
<arg value="hello"/>
</args>
</payloadFactory>
</inSequence>
<outSequence>
<send/>
</outSequence>
</target>
<publishWSDL uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
</proxy>
<sequence name="fault">
<log level="full">
<property name="MESSAGE" value="Executing default "fault" sequence"/>
<property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
<property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
</log>
<drop/>
</sequence>
<sequence name="main">
<log/>
```

```
<drop/>
</sequence>
</definitions>
```

## Axis2 Properties

[ CacheLevel ] [ ConcurrentConsumers ] [ HTTP\_ETAG ] [ JMS\_COORELATION\_ID ] [ MaxConcurrentConsumers ] [ MercurySequenceKey ] [ MercuryLastMessage ] [ FORCE\_HTTP\_1.0 ]

Axis2 properties allow you to configure the web services engine in the ESB, such as specifying how to cache JMS objects, setting the minimum and maximum threads for consuming messages, and forcing outgoing HTTP/S messages to use HTTP 1.0. You can access some of these properties through the [Property mediator](#) with the scope set to axis2 or axis2-client as shown below.

### ***CacheLevel***

<b>Name</b>	CacheLevel
<b>Possible Values</b>	none, connection, session, consumer, producer, auto
<b>Default Behavior</b>	
<b>Scope</b>	
<b>Description</b>	<p>This property determines which JMS objects should be cached. JMS objects are cached so that they can be reused in the subsequent invocations. Each caching level can be described as follows:</p> <ul style="list-style-type: none"> <li>none: No JMS object will be cached.</li> <li>connection: JMS connection objects will be cached.</li> <li>session: JMS connection and session objects will be cached.</li> <li>consumer: JMS connection, session and consumer objects will be cached.</li> <li>producer: JMS connection, session and producer objects will be cached.</li> <li>auto: An appropriate caching level will be used depending on the transaction strategy.</li> </ul>
<b>Example</b>	<pre>&lt;parameter name="transport.jms.CacheLevel"&gt;consumer&lt;/parameter&gt;</pre>

### ***ConcurrentConsumers***

<b>Name</b>	ConcurrentConsumers
<b>Possible Values</b>	integer
<b>Default Behavior</b>	
<b>Scope</b>	

<b>Description</b>	The minimum number of threads for message consuming. The value specified for this property is the initial number of threads started. As the number of messages to be consumed increases, number of threads are also increased to match the load until the total number of threads equals the value specified for the <code>transport.jms.MaxConcurrentConsumers</code> property.
<b>Example</b>	<pre>&lt;parameter name="transport.jms.ConcurrentConsumers" locked="false"&gt;50&lt;/parameter&gt;</pre>

### HTTP\_ETAG

<b>Name</b>	HTTP_ETAG
<b>Possible Values</b>	true/false
<b>Default Behaviour</b>	
<b>Scope</b>	axis2
<b>Description</b>	<p>This property determines whether the HTTP Etag should be enabled for the request or not.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <p>HTTP Etag is a mechanism provided by HTTP for Web cache validation.</p> </div>
<b>Example</b>	<pre>&lt;property name="HTTP_ETAG" scope="axis2" type="BOOLEAN" value="true"/&gt;</pre>

### JMS\_COORELATION\_ID

<b>Name</b>	JMS_COORELATION_ID
<b>Possible Values</b>	String
<b>Default Behavior</b>	
<b>Scope</b>	axis2
<b>Description</b>	<p>The JMS coorelation ID is used to match responses with specific requests. This property can be used to set the JMS coorrelation ID as a dynamic or a hard coded value in a request. As a result, responses with the matching JMS correlation IDs will be matched with the request.</p>
<b>Example</b>	<pre>&lt;property name="JMS_COORELATION_ID" action="set" scope="axis2" expression="\$header/wsa:MessageID" xmlns:sam="http://sample.esb.org/&gt;</pre>

### MaxConcurrentConsumers

<b>Name</b>	MaxConcurrentConsumers
<b>Possible Values</b>	integer
<b>Default Behavior</b>	
<b>Scope</b>	
<b>Description</b>	The maximum number of threads that can be added for message consuming. See <a href="#">ConcurrentConsumers</a> .
<b>Example</b>	<pre>&lt;parameter     name="transport.jms.MaxConcurrentConsumers" locked="false"&gt;50&lt;/parameter&gt;</pre>

### *MercurySequenceKey*

<b>Name</b>	MercurySequenceKey
<b>Possible Values</b>	integer
<b>Default Behavior</b>	
<b>Scope</b>	
<b>Description</b>	Can be an identifier specifying a Mercury internal sequence key.
<b>Example</b>	

### *MercuryLastMessage*

<b>Name</b>	MercuryLastMessage
<b>Possible Values</b>	true/false
<b>Default Behavior</b>	
<b>Scope</b>	
<b>Description</b>	When set to "true", it will make this the last message and terminate the sequence.
<b>Example</b>	

### *FORCE\_HTTP\_1.0*

<b>Name</b>	FORCE_HTTP_1.0
<b>Possible Values</b>	true/false
<b>Default Behavior</b>	
<b>Scope</b>	axis2-client

<b>Description</b>	Forces outgoing http/s messages to use HTTP 1.0 (instead of the default 1.1).
<b>Example</b>	

## Synapse Message Context Properties

[ SYSTEM\_DATE ] [ SYSTEM\_TIME ] [ To, From, Action, FaultTo, ReplyTo, MessageID ] [ MESSAGE\_FORMAT ] [ OperationName ]

The Synapse message context properties allow you to get information about the message, such as the date/time it was sent, the message format, and the message operation. You can use the `get-property()` function in the [Property mediator](#) with the scope set to `Synapse` to retrieve these properties.

### **SYSTEM\_DATE**

<b>Name</b>	SYSTEM_DATE
<b>Possible Values</b>	
<b>Default Behavior</b>	
<b>Scope</b>	
<b>Description</b>	Returns the current date as a String. Optionally, a date format as per the standard date format may be supplied. e.g. <code>synapse:get-property("SYSTEM_DATE", "yyyy.MM.dd G 'at' HH:mm:ss z")</code> or <code>get-property('SYSTEM_DATE')</code> .
<b>Example</b>	

### **SYSTEM\_TIME**

<b>Name</b>	SYSTEM_TIME
<b>Possible Values</b>	
<b>Default Behavior</b>	
<b>Scope</b>	
<b>Description</b>	Returns the current time in milliseconds, i.e. the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.
<b>Example</b>	

### **To, From, Action, FaultTo, ReplyTo, MessageID**

<b>Names</b>	To, From, Action, FaultTo, ReplyTo, MessageID
<b>Possible Values</b>	
<b>Default Behavior</b>	
<b>Scope</b>	

<b>Description</b>	The message To, Action and WS-Addressing properties.
<b>Example</b>	

### ***MESSAGE\_FORMAT***

<b>Names</b>	MESSAGE_FORMAT
<b>Possible Values</b>	
<b>Default Behavior</b>	
<b>Scope</b>	
<b>Description</b>	Returns the message format, i.e. returns pox, get, soap11 or soap12.
<b>Example</b>	

### ***OperationName***

<b>Names</b>	OperationName
<b>Possible Values</b>	
<b>Default Behavior</b>	
<b>Scope</b>	
<b>Description</b>	Returns the operation name for the message.
<b>Example</b>	

## Accessing Properties with XPath

WSO2 ESB supports standard XPath functions and variables through its underlying XPath engine. The ESB also provides custom XPath functions and variables for accessing message properties.

- [XPath Extension Functions](#)
- [Synapse XPath Variables](#)

### XPath Extension Functions

In addition to standard XPath functions, the ESB supports the following custom functions for working with XPath expressions:

- `base64Encode()` function
- `base64Decode()` function
- `get-property()` function
- `url-encode()` function

`base64Encode()` function

The `base64Encode` function returns the base64-encoded value of the specified string.

Syntax:

- `base64Encode(string value)`

- `base64Encode(string value, string charset)`
- `base64Decode()` function

The `base64Decode` function returns the original value of the specified base64-encoded value.

Syntax:

- `base64Decode(string encodedValue)`
  - `base64Decode(string encodedValue, string charset)`
- `get-property()` function

The `get-property()` function allows any XPath expression used in a configuration to look up information from the current message context. Using the [Property mediator](#), you can retrieve properties from the message context and header.

The syntax of the function takes the following format.

- `get-property(String propertyName)`
- `get-property(String scope, String propertyName)`

The function accepts scope as an optional parameter. It retrieves a message property at the given scope, which can be one of the following.

- [Synapse scope](#)
- [axis2 scope](#)
- [axis2-client](#)
- [transport scope](#)
- [registry scope](#)
- [system scope](#)
- [operation scope](#)

If you provide only the property name without the scope, the default `synapse` scope will be used.

When the result of an XPath evaluation results in a single XML node, the evaluator will return the text content of this node by default (equivalent of doing `/root/body/node/text()`). If you want to retrieve the node itself, you can configure the [Enrich mediator](#) as shown in the following example.

```

<inSequence>
<log level="custom">
<property name="WHERE" value="before doing stuff"/>
</log>
<enrich>
<source type="body" clone="true"/>
<target type="property" property="ENRICH_PROPERTY" />
</enrich>
<property name="PROPERTY_PROPERTY"
expression="$body/child::node()"
scope="default" />
<log level="custom">
<property name="WHERE" value="before doing stuff"/>
<property name="ENRICH_PROPERTY" expression="get-property('ENRICH_PROPERTY')"/>
<property name="PROPERTY_PROPERTY"
expression="get-property('PROPERTY_PROPERTY')"/>
</log>
<enrich>
<source type="property" clone="true" property="ENRICH_PROPERTY" />
<target type="body" action="sibling" />
</enrich>
<log level="full"/>
</inSequence>

```

#### Synapse scope

When the scope of a property mediator is `synapse`, its value is available throughout both the in sequence and the out sequence. In addition to the user-defined properties, you can retrieve the following special properties from the `synapse` scope.

Name	Return Value
To	Incoming URL as a String, or empty string («») if a To address is not defined.
From	From address as a String, or empty string («») if a From address is not defined.
Action	SOAP Addressing Action header value as a String, or empty string («») if an Action is not defined.
FaultTo	SOAP FaultTo header value as a String, or empty string («») if a FaultTo address is not defined.
ReplyTo	ReplyTo header value as a String, or empty string («») if a ReplyTo address is not defined.
MessageID	A unique identifier (UUID) for the message as a String, or empty string («») if a MessageID is not defined. This ID is guaranteed to be unique.
FAULT	TRUE if the message has a fault, or empty string if the message does not have a fault.
MESSAGE_FORMAT	Returns pox, get, soap11, or soap12 depending on the message. If a message type is unknown this returns soap12

OperationName	Operation name corresponding to the message. A proxy service with a WSDL can have different operations. If the WSDL is not defined, ESB defines fixed operations.
---------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------

To access a property with the `synapses` scope inside the `mediate()` method of a mediator, you can include the following configuration in a custom mediator created using the [Class mediator](#):

```
public boolean mediate(org.apache.synapse.MessageContext mc) {
 // Available in both in-sequence and out-sequenc
 String propValue = (String) mc.getProperty("PropName");
 System.out.println("SCOPE_SYNAPSE : " + propValue);
 return true;
}
```

axis2 scope

When the scope of a property mediator is `axis2`, its value is available only throughout the sequence for which the property is defined (e.g., if you add the property to an in sequence, its value will be available only throughout the in sequence). You can retrieve message context properties within the `axis2` scope using the following syntax.

Syntax:

```
get-property('axis2', String propertyName)
```

To access a property with the `axis2` scope inside the `mediate()` method of a mediator, you can include the following configuration in a custom mediator created using the [Class mediator](#):

```
public boolean mediate(org.apache.synapse.MessageContext mc) {
 org.apache.axis2.context.MessageContext axis2MsgContext;
 axis2MsgContext = ((Axis2MessageContext) mc).getAxis2MessageContext();

 // Available only in the sequence the property is defined.
 String propValue = (String) axis2MsgContext.getProperty("PropName");
 System.out.println("SCOPE_AXIS2 : " + propValue);
 return true;
}
```

axis2-client

This is similar to the `synapse` scope. The difference is that it can be accessed inside the `mediate()` method of a mediator by including one of the following configurations in a custom mediator, created using the [Class mediator](#):

```
public boolean mediate(org.apache.synapse.MessageContext mc) {
 org.apache.axis2.context.MessageContext axis2MsgContext;
 axis2MsgContext = ((Axis2MessageContext) mc).getAxis2MessageContext();
 String propValue = (String) axis2MsgContext.getProperty("PropName");
 System.out.println("SCOPE_AXIS2_CLIENT - 1 : " + propValue);
```

or

```

propValue = (String) axis2MsgContext.getOptions().getProperty("PropName");
System.out.println("SCOPE_AXIS2_CLIENT - 2: " + propValue);
return true;
}

```

#### transport scope

When the scope of a property mediator is `transport`, it will be added to the transport header of the outgoing message from the ESB. You can retrieve message context properties within the `transport` scope using the following syntax.

#### Syntax:

```
get-property('transport', String propertyName)
registry scope
```

You can retrieve properties within the registry using the following syntax.

#### Syntax:

```
get-property('registry', String registryPath@propertyName)
get-property('registry', String registryPath)
system scope
```

You can retrieve Java System properties using the following syntax.

#### Syntax:

```
get-property('system', String propertyName)
operation scope
```

You can retrieve a property in the operation context level from the `operation` scope. The properties within iterated/cloned message with the `operation` scope are preserved in the in sequence even if you have configured your API resources to be sent through the fault sequence when faults exist. A given property with the `operation` scope only exists in a single request and can be accessed by a single resource. The properties in this scope are passed to the error handler when the `FORCE_ERROR_ON_SOAPFAULT` property is set to `true`.

See `FORCE_ERROR_ON_SOAPFAULT` section in [Generic Properties](#) for more information.

#### Syntax:

```
get-property('operation', String propertyName)
url-encode() function
```

The `url-encode` function returns the URL-encoded value of the specified string.

#### Syntax:

- `url-encode(string value)`
- `url-encode(string value, string charset)`

Next, see [Synapse XPath Variables](#).

### Synapse XPath Variables

There is a set of predefined XPath variables that you can directly use to write XPaths in the Synapse configuration, instead of using the `synapse:get-property()` function. These XPath variables get properties of various scopes as follows:

- `$body`
- `$header`
- `$axis2`
- `$ctx`
- `$trp`

- \$url
- \$func
- \$env

## \$body

The SOAP 1.1 or 1.2 body element. For example, the expression **\$body/getQuote** refers to the first **getQuote** element in a SOAP body, regardless of whether the message is SOAP-11 or SOAP-12. We have discussed an example below.

### Example of \$body usage:

1. Deploy the following proxy service using instructions in [Adding a Proxy Service](#).

Note the property, `<property xmlns:m0=" http://services.samples " name="stockprop" expression="$body/m0:getQuote"/>` in the configuration. It is used to log the first `<m0:getQuote>` element of the request SOAP body.

```

<proxy xmlns="http://ws.apache.org/ns/synapse" name="StockQuoteProxy"
transports="https,http" statistics="disable" trace="disable" startOnLoad="true">
 <target>
 <inSequence>
 <log
 <property xmlns:m0="http://services.samples" name="stockprop"
expression="$body/m0:getQuote" />
 </log>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <description></description>
</proxy>

```

2. Send the following StockQuote request using the sample StockQuote client. For information on the sample client, refer to the **Sample Clients** sub heading in [Setting Up the ESB Samples](#).

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
```

3. Note the following message in the ESB log.

```
[2013-03-18 14:04:41,019] INFO - LogMediator To: /services/StockQuoteProxy,
WSAction: urn:getQuote, SOAPAction: urn:getQuote, ReplyTo:
http://www.w3.org/2005/08/addressing/anonymous, MessageID:
urn:uuid:930f68f5-199a-4eff-90d2-ea679c2362ab, Direction: request, stockprop =
<m0:getQuotexmlns:m0="http://services.samples"><m0:request><m0:symbol>IBM</m0:sym
bol></m0:request></m0:getQuote>
```

## \$header

The SOAP 1.1 or 1.2 header element. For example, the expression **\$header/wsa:To** refers to the addressing **To** header regardless of whether this message is SOAP-11 or SOAP-12. We have discussed an example below.

#### Example of \$header usage:

1. Deploy the following proxy service using instructions in [Adding a Proxy Service](#).

Note the property, `<property xmlns:wsa="http://www.w3.org/2005/08/addressing" name="stockprop" expression="$header/wsa:To"/>` in the configuration. It is used to log the value of **wsa:To** header of the SOAP request.

```

<proxy xmlns="http://ws.apache.org/ns/synapse" name="StockQuoteProxy"
transports="https,http" statistics="disable" trace="disable" startOnLoad="true">
 <target>
 <inSequence>
 <log>
 <property xmlns:wsa="http://www.w3.org/2005/08/addressing"
name="stockprop" expression="$header/wsa:To" />
 </log>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <description></description>
</proxy>

```

2. Send the following StockQuote request using the sample StockQuote client. For information on the sample client, refer to the **Sample Clients** sub heading in [Setting Up the ESB Samples](#).

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
```

3. Note the following message in the ESB log.

```

[2013-03-18 14:14:16,356] INFO - LogMediator To:
http://localhost:9000/services/SimpleStockQuoteService, WSAction: urn:getQuote,
SOAPAction: urn:getQuote, ReplyTo:
http://www.w3.org/2005/08/addressing/anonymous, MessageID:
urn:uuid:8a64c9cb-b82f-4d6f-a45d-bef37f8b664a, Direction: request,
stockprop = http://localhost:9000/services/SimpleStockQuoteService

```

#### \$axis2

Prefix for Axis2 MessageContext properties. This is used to get the property value at the axis2 scope. For example, to get the value of Axis2 message context property with name REST\_URL\_POSTFIX, use the XPath expression **\$axis2:REST\_URL\_POSTFIX**. We have discussed an example below.

#### Example of \$axis2 usage:

1. Deploy the following proxy service using instructions in [Adding a Proxy Service](#).

Note the property, `<property name="stockprop" expression="$axis2:REST_URL_POSTFIX"/>` in the configuration which is used to log the REST\_URL\_POSTFIX value of the request message.

```

<proxy xmlns="http://ws.apache.org/ns/synapse" name="StockQuoteProxy"
transports="https,http" statistics="disable" trace="disable" startOnLoad="true">
 <target>
 <inSequence>
 <log>
 <property name="stockprop" expression="$axis2:REST_URL_POSTFIX" />
 </log>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <description></description>
</proxy>

```

2. Send the following StockQuote request using the sample StockQuote client. For information on the sample client, refer to the **Sample Clients** sub heading in [Setting Up the ESB Samples](#).

```

ant stockquote
-Daddurl=http://localhost:8280/services/StockQuoteProxy/test/prefix

```

3. Note the following message in the ESB log.

```

INFO - LogMediator To:
http://localhost:8280/services/StockQuoteProxy/test/prefix, WSAction:
urn:getQuote, SOAPAction: urn:getQuote, ReplyTo:
http://www.w3.org/2005/08/addressing/anonymous, MessageID:
urn:uuid:ecd228c5-106a-4448-9c83-3b1e957e2fe5, Direction: request, stockprop =
/test/prefix

```

In this example, the property definition, `<property name="stockprop" expression="$axis2:REST_URL_POSTFIX"/>` is equivalent to `<property name="stockprop" expression="get-property('axis2','REST_URL_POSTFIX')"/>`

Similarly, you can use \$axis2 prefix with [HTTP Transport Properties](#).

### \$ctx

Prefix for Synapse MessageContext properties and gets a property at the default scope. For example, to get the value of Synapse message context property with name ERROR\_MESSAGE, use the XPath expression `$ctx:ERROR_MESSAGE`. We have discussed an example below.

#### **Example of \$ctx usage:**

This example sends a request to a sample proxy service, and sets the target endpoint to a non-existent endpoint

reference key. It causes a mediation fault, which triggers the fault sequence.

1. Deploy the following proxy service using instructions in [Adding a Proxy Service](#).

Note the property, `<property name="stockerrorprop" expression="$ctx:ERROR_MESSAGE"/>` in the fault sequence configuration. It is used to log the error message that occurs due to a mediation fault.

```

<proxy xmlns="http://ws.apache.org/ns/synapse" name="StockQuoteProxy"
transports="https,http" statistics="disable" trace="disable" startOnLoad="true">
 <target>
 <inSequence>
 <send>
 <endpoint key="ep2" />
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 <faultSequence>
 <log>
 <property name="stockerrorprop" expression="$ctx:ERROR_MESSAGE" />
 <property name="Cause" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 </faultSequence>
 </target>
 <description></description>
</proxy>

```

2. Send the following StockQuote request using the sample StockQuote client. For information on the sample client, refer to the **Sample Clients** sub heading in [Setting Up the ESB Samples](#).

```
ant stockquote -Dtrpurl=http://localhost:8280/services/StockQuoteProxy
```

3. Note the following message in the ESB log.

```

INFO - LogMediator To: /services/StockQuoteProxy, WSAction: urn:getQuote,
SOAPAction: urn:getQuote, ReplyTo:
http://www.w3.org/2005/08/addressing/anonymous, MessageID:
urn:uuid:54205f7d-359b-4e82-9099-0f8e3bf9d014, Direction: request, stockerrorprop
= Couldn't find the endpoint with the key : ep2

```

In this example, the property definition, `<property name="stockerrorprop" expression="$ctx:ERROR_MESSAGE"/>` is equivalent to `<property name="stockerrorprop" expression="get-property('ERROR_MESSAGE')"/>`.

Similarly, you can use \$ctx prefix with [Generic Properties](#).

### \$trp

Prefix used to get the transport headers. For example, to get the transport header named Content-Type of the current message, use the XPath expression **\$trp:Content-Type**. HTTP transport headers are not case sensitive. Therefore, \$trp:Content-Type and \$trp:CONTENT-TYPE are regarded as the same. We have discussed an example below.

#### **Example of \$trp usage:**

1. Deploy the following proxy service using instructions given in section [Adding a Proxy Service](#).

Note the property, `<property name="stockprop" expression="$trp:Content-Type"/>` in the configuration, which is used to log the Content-Type HTTP header of the request message.

```

<proxy xmlns="http://ws.apache.org/ns/synapse" name="StockQuoteProxy"
transports="https,http" statistics="disable" trace="disable" startOnLoad="true">
 <target>
 <inSequence>
 <log>
 <property name="stockprop" expression="$trp:Content-Type" />
 </log>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <description></description>
</proxy>

```

2. Send the following StockQuote request using the sample StockQuote client. For information on the sample client, refer to the **Sample Clients** sub heading in [Setting Up the ESB Samples](#).

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
```

3. Note the following message in the ESB log.

```
[2013-03-18 12:23:14,101] INFO - LogMediator To:
http://localhost:8280/services/StockQuoteProxy, WSAction: urn:getQuote,
SOAPAction: urn:getQuote, ReplyTo:
http://www.w3.org/2005/08/addressing/anonymous, MessageID:
urn:uuid:25a3143a-5b18-4cbb-b8e4-27d4dd1895d2, Direction: request, stockprop =
text/xml; charset=UTF-8
```

In this example, the property definition, `<property name="stockprop" expression="$trp:Content-Type"/>` is equivalent to `<property name="stockprop" expression="get-property('transport','Content-Type')"/>`. Similarly, you can use \$trp prefix with [HTTP Transport Properties](#).

### \$url

The prefix used to get the URI element of a request URL.

#### **Example of \$url usage:**

1. Create a REST API with the following configuration using instructions given in page [Working with APIs](#).

```

<api xmlns="http://ws.apache.org/ns/synapse" name="Editing" context="/editing">
 <resource methods="GET" uri-template="/edit?a={symbol}&b={value}">
 <inSequence>
 <log level="full">
 <property name="SYMBOL" expression="$url:a"/>
 <property name="VALUE" expression="$url:b"/>
 </log>
 <respond/>
 </inSequence>
 </resource>
</api>

```

- Send a request to the REST API you created using a browser as follows:

```
http://10.100.5.73:8280/editing/edit?a=wso2&b=2.4
```

You will see the following in the ESB log:

```

LogMediator To: /editing/edit?a=wso2&b=2.4, MessageID:
urn:uuid:36cb5ad7-f150-490d-897a-ee7b86a9307d, Direction: request, SYMBOL = wso2,
VALUE = 2.4, Envelope: <?xml version="1.0" encoding="utf-8"?><soapenv:Envelope
xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"><soapenv:Body></soapenv:Body></soapenv:Envelope>
```

## \$func

The prefix used to refer to a particular parameter value passed externally by an invoker such as the [Call Template Mediator](#).

### Example of \$func usage:

- Add a sequence template with the following configuration. See [Adding a New Sequence Template](#) for detailed instructions.

```

<template xmlns="http://ws.apache.org/ns/synapse" name="HelloWordLogger">
 <sequence>
 <log level="full">
 <property xmlns:ns2="http://org.apache.synapse/xsd"
 xmlns:ns="http://org.apache.synapse/xsd" name="message"
 expression="$func:message"/>
 </log>
 </sequence>
</template>

```

- Deploy the following proxy service using instructions given in section [Adding a Proxy Service](#).

```

<proxy xmlns="http://ws.apache.org/ns/synapse"
 name="StockQuoteProxy"
 transports="https,http"
 statistics="disable"
 trace="disable"
 startOnLoad="true">
 <target>
 <inSequence>
 <call-template target="HelloWorldLogger">
 <with-param name="message" value="HelloWorld"/>
 </call-template>
 <log/>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </target>
 <description/>
 </proxy>

```

- Send the following StockQuote request using the sample StockQuote client. For information on the sample client, refer to the **Sample Clients** sub heading in [Setting Up the ESB Samples](#).

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
```

- Note the following message in the ESB log.

```

LogMediator To: http://localhost:8280/services/StockQuoteProxy, WSAction:
urn:getQuote, SOAPAction: urn:getQuote, ReplyTo:
http://www.w3.org/2005/08/addressing/anonymous, MessageID:
urn:uuid:8d90e21b-b5cc-4a02-98e2-24b324fa704c, Direction: request, message =
HelloWorld

```

## \$env

Prefix used to get a SOAP 1.1 or 1.2 envelope level element. For example, to get the body element from the SOAP envelope, use the expression **\$env/\*[local-name()='Body']** .

### Example of \$env usage:

- Create an API with the following configuration. For information on how to create an API, see [Working with APIs](#).

```

<api context="/soapEnvelopeTest" name="SoapEnvelopeTest">
 <resource url-mapping="/">
 <inSequence>
 <loopback/>
 </inSequence>
 <outSequence>
 <property name="messageType" scope="axis2"
value="application/xml"/>
 <payloadFactory media-type="xml">
 <format>
 <theData xmlns="http://some.namespace">
 <item>$1</item>
 </theData>
 </format>
 <args>
 <arg evaluator="xml"
expression="$env/*[local-name()='Body']/*[local-name()='jsonObject']/*"/>
 </payloadFactory>
 <property name="messageType" scope="axis2"
value="application/json"/>
 <send/>
 </outSequence>
 </resource>
 </api>

```

2. Send a post request to the API you created (i.e.,<http://localhost:8280/soapEnvelopeTest>), with the following json payload using a rest client.

```
{ "content":{ "paramA": "ValueA", "paramB": "valueB" } }
```

You will receive the following response:

```
{"theData":{ "item":{ "content":{ "paramA": "ValueA", "paramB": "valueB" } } }}
```

## Synapse Configuration Reference

WSO2 ESB is developed based on the Apache Synapse mediation engine. For a full guide of Apache Synapse configuration, refer to <http://synapse.apache.org/userguide/config.html>.

### Setting Up Host Names and Ports

This sections describes the configurations required in order to bind address values and HTTP/HTTPS ports used by the ESB.

#### axis2.xml file

Define the following parameters under the HTTP and HTTPS transport receiver configurations in the <ESB\_HOME>/repository/conf/axis2/axis2.xml file.

1. To define a binding address, add the following parameter.

```
<parameter name="bind-address" locked="false">hostname or IP address</parameter>
```

2. When a bind address is defined, it is recommended to also define a WSDL prefix for the HTTP and HTTPS transports. The WSDL prefix value is added to all the HTTP/HTTPS endpoints in the auto-generated, user-published WSDLs. To setup a WSDL prefix, add the following parameter.

```
<parameter name="WSDLEPRPrefix"
locked="false">https://apachehost:port/somepath</parameter>
```

3. To configure the HTTP and HTTPS ports used by the ESB HTTP-NIO transport, add the following parameter.

```
<parameter name="port" locked="false">8280</parameter>
```

The following sample HTTP configuration shows how to listen on HTTP port 8280 of the hostname **my.test.server.com**.

```
<transportReceiver name="http"
class="org.apache.synapse.transport.nhttp.HttpCoreNIOListener">
<parameter name="port" locked="false">8280</parameter>
<parameter name="non-blocking" locked="false">true</parameter>
<parameter name="bind-address" locked="false">my.test.server.com</parameter>
</transportReceiver>
```

4. WSO2 ESB also allows you to set a HTTP proxy port to deploy the ESB behind a proxy server using Apache mod\_proxy. Add the following parameter to specify the HTTP proxy port.

```
<parameter name="proxyPort">80</parameter>
```

Refer to [WSO2 Carbon Transports Catalog](#) for more information on setting up HTTP and HTTPS NIO transports, and the servlet HTTPS transport for various deployments.

#### catalina-server.xml file

1. To change the ports used by the ESB management console, modify the <ESB\_HOME>/repository/conf/tomcat/catalina-server.xml file. By default the management console accepts HTTPS requests on port 9443. Change the **port** parameter to set the HTTPS port used by the console. For example,

```
<parameter name="port">9443</parameter>
```

# Samples

WSO2 ESB includes working examples that demonstrate its features and capabilities. You can use the sample configurations as a starting point when building your own configurations. To try a sample, [follow the setup instructions](#) to set up the samples and start the back-end services, and then use the provided [sample clients](#) to run them. For details on running a specific sample, browse the list of samples below and click the link for the sample you want to try.

The ESB configurations included with each sample are the raw source XML serialization of the sample configuration. You can view the configuration graphically in the management console.

- [Setting Up the ESB Samples](#)
- [Using the Sample Clients](#)
- [Message Mediation Samples](#)
  - [Sample 0: Introduction to ESB](#)
  - [Sample 1: Simple Content-Based Routing \(CBR\) of Messages](#)
  - [Sample 2: CBR with the Switch-Case Mediator Using Message Properties](#)
  - [Sample 3: Local Registry Entry Definitions, Reusable Endpoints and Sequences](#)
  - [Sample 4: Specifying a Fault Sequence with a Regular Mediation Sequence](#)
  - [Sample 5: Creating SOAP Fault Messages and Changing the Direction of a Message](#)
  - [Sample 6: Manipulating SOAP Headers and Filtering Incoming and Outgoing Messages](#)
  - [Sample 7: Using Schema Validation and the Usage of Local Registry for Storing Configuration Metadata](#)
  - [Sample 8: Introduction to Static and Dynamic Registry Resources and Using XSLT Transformations](#)
  - [Sample 9: Introduction to Dynamic Sequences with the Registry](#)
  - [Sample 10: Introduction to Dynamic Endpoints with the Registry](#)
  - [Sample 11: Using a Full Registry-Based Configuration and Sharing a Configuration Between Multiple Instances](#)
  - [Sample 12: One-Way Messaging in a Fire-and-Forget Mode through ESB](#)
  - [Sample 13: Dual Channel Invocation Through Synapse](#)
  - [Sample 14: Using Sequences and Endpoints as Local Registry Items](#)
  - [Sample 15: Using the Enrich Mediator for Message Copying and Content Enrichment](#)
  - [Sample 16: Introduction to Dynamic and Static Registry Keys](#)
  - [Sample 17: Transforming / Replacing Message Content with PayloadFactory Mediator](#)
  - [Sample 18: Transforming a Message Using ForEach Mediator](#)
- [Advanced Mediation with Endpoints](#)
  - [Sample 50: POX to SOAP conversion](#)
  - [Sample 51: MTOM and SwA Optimizations and Request/Response Correlation](#)
  - [Sample 52: Using Load Balancing Endpoints to Handle Peak Loads](#)
  - [Sample 53: Using Failover Endpoints to Handle Peak Loads](#)
  - [Sample 54: Session Affinity Load Balancing between Three Endpoints](#)
  - [Sample 55: Session Affinity Load Balancing between Failover Endpoints](#)
  - [Sample 56: Using a WSDL Endpoint as the Target Endpoint](#)
  - [Sample 57: Dynamic Load Balancing between Three Nodes](#)
  - [Sample 58: Static Load Balancing between Three Nodes](#)
  - [Sample 59: Weighted load balancing between 3 endpoints](#)
  - [Sample 60: Routing a Message to a Static List of Recipients](#)
  - [Sample 61: Routing a Message to a Dynamic List of Recipients](#)
  - [Sample 62: Routing a Message to a Dynamic List of Recipients and Aggregating Responses](#)
- [Quality of Service Samples in Message Mediation](#)
  - [Sample 100: Using WS-Security for Outgoing Messages](#)
- [Proxy Service Samples](#)
  - [Sample 150: Introduction to Proxy Services](#)
  - [Sample 151: Custom Sequences and Endpoints with Proxy Services](#)
  - [Sample 152: Switching Transports and Message Format from SOAP to REST POX](#)
  - [Sample 153: Routing Messages that Arrive to a Proxy Service without Processing Security Headers](#)

- Sample 154: Load Balancing with Proxy Services
- Sample 155: Dual Channel Invocation on Both Client Side and Server Side of Synapse with Proxy Services
- Sample 156: Service Integration with specifying the receiving sequence
- Sample 157: Conditional Router for Routing Messages based on HTTP URL, HTTP Headers and Query Parameters
- Quality of Service Samples in Service Mediation
- Transports Samples and Switching Transports
  - Sample 250: Introduction to Switching Transports
  - Sample 251: Switching from HTTP(S) to JMS
  - Sample 252: Pure Text (Binary) and POX Message Support with JMS
  - Sample 253: Bridging from JMS to HTTP and Replying with a 202 Accepted Response
  - Sample 254: Using the File System as Transport Medium (VFS)
  - Sample 255: Switching from FTP Transport Listener to Mail Transport Sender
  - Sample 256: Proxy Services with the MailTo Transport
  - Sample 257: Proxy Services with the FIX Transport
  - Sample 258: Switching from HTTP to FIX
  - Sample 259: Switch from FIX to HTTP
  - Sample 260: Switch from FIX to AMQP
  - Sample 261: Switching between FIX Versions
  - Sample 262: CBR of FIX Messages
  - Sample 263: Transport Switching - JMS to http/s Using JBoss Messaging(JBM)
  - Sample 264: Sending Two-Way Messages Using JMS transport
  - Sample 265: Accessing a Windows Share Using the VFS Transport
  - Sample 266: Switching from TCP to HTTPS/S
  - Sample 267: Switching from UDP to HTTPS/S
  - Sample 268: Proxy Services with the Local Transport
  - Sample 270: Transport switching from HTTP to MSMQ and MSMQ to HTTP
  - Sample 271: File Processing
  - Sample 272: Publishing and Subscribing using WSO2 ESB's MQ Telemetry Transport
- Introduction to ESB Tasks
  - Sample 300: Introduction to Tasks with a Simple Trigger
- Advanced Mediations with Advanced Mediators
  - Using Scripts in Mediation (Script Mediator)
  - Database Interactions in Mediation (DBLookup / DBReport)
  - Throttling Messages (Throttle Mediator)
  - Extending the Mediation in Java (Class Mediator)
  - Evaluating XQuery for Mediation (XQuery Mediator)
  - Splitting Messages into Parts and Processing in Parallel (Iterate/Aggregate)
  - Caching Responses Over Requests (Cache Mediator)
  - Mediating JSON Messages
  - Rewriting the URL (URL Rewrite Mediator)
  - Eventing (Event Mediator)
  - Mediating with Spring
- Invoking Web Services
  - Sample 430: Callout Mediator for Synchronous Service Invocation
  - Sample 500: Call Mediator for Non-Blocking Service Invocation
- Introduction to Rule Mediator
  - Sample 600 : Simple Message Transformation - Rule Mediator for Message Transformation
  - Sample 601 : Advance Rule Based Routing - Switching Routing Decision According to the Rules - Rule Mediator as Switch mediator
- Miscellaneous Samples
  - Sample 650: File Hierarchy-Based Configuration Builder
  - Sample 651: Using Synapse Observers
  - Sample 652: Priority Based Message Mediation
  - Sample 653: NHTTP Transport Priority Based Dispatching
  - Sample 654: Smooks Mediator

- Sample 655: Message Relay - Basics Sample
- Sample 656: Message Relay - Builder Mediator
- Sample 657: Distributed Transaction Management
- Sample 658: Huge XML Message Processing with Smooks Mediator
- Sample 659: Huge EDI Message Processing with Smooks Mediator
- Store and Forward Messaging Patterns with Message Stores and Processors
  - Sample 700: Introduction to Message Store
  - Sample 701: Introduction to Message Sampling Processor
  - Sample 702: Introduction to Message Forwarding Processor
  - Sample 703: Adding Security to Message Forwarding Processor
  - Sample 704: RESTful Invocations with Message Forwarding Processor
  - Sample 705: Load Balancing with Message Forwarding Processor
- Template Samples
  - Sample 750: Stereotyping XSLT Transformations with Templates
  - Sample 751: Message Split Aggregate Using Templates
  - Sample 752: Load Balancing Between 3 Endpoints With Endpoint Templates
- REST API Management
  - Sample 800: Introduction to REST API
- Inbound Endpoint Samples
  - Sample 900: Inbound Endpoint File Protocol Sample (VFS)
  - Sample 901: Inbound Endpoint JMS Protocol Sample
  - Sample 902: HTTP Inbound Endpoint Sample
  - Sample 903: HTTPS Inbound Endpoint Sample
  - Sample 904: Inbound Endpoint Kafka Protocol Sample
  - Sample 905: Inbound HL7 with Automatic Acknowledgement
  - Sample 906: Inbound Endpoint MQTT Protocol Sample
  - Sample 907: Inbound Endpoint RabbitMQ Protocol Sample

## Setting Up the ESB Samples

This section describes the prerequisites and instructions on how to start the ESB with the sample configurations, how to start the Axis2 server and deploy the sample back-end services to the Axis2 server as well as how to set up any listeners and transports that are required by the samples.

Once you have set up the samples, you can run them using the sample clients.

---

[Prerequisites](#) | [Understanding the samples](#) | [Starting the ESB with a sample configuration](#) | [Deploying sample back-end services](#) | [Starting the Axis2 server](#) | [Configuring WSO2 ESB to use the JMS transport](#) | [Configuring WSO2 ESB to use the mail transport](#) | [Configuring WSO2 ESB to use the NHTTP transport](#) | [Configuring WSO2 ESB to use the FIX transport](#) | [Configuring WSO2 ESB to use the VFS transport](#) | [Configuring WSO2 ESB to use the AMQP transport](#) | [Configuring WSO2 ESB to use the TCP transport](#) | [Configuring WSO2 ESB to use the UDP transport](#) | [Configuring the ESB for script mediator support](#) | [Setting up the databases](#) | [Setting up Synapse DataSources](#)

---

### Prerequisites

1. Install all the required software as described in the [Installation Prerequisites](#) section.
  - To run the samples, you must have the correct version of the Java Development Kit/JRE as well as Apache Ant installed .
  - To run the JMS samples, you must have ActiveMQ or another JMS provider installed.
2. Open a command prompt (or a shell in Linux) and go to the <ESB\_HOME>\bin directory. Then run the ant c ommand as shown below to build the build.xml file.

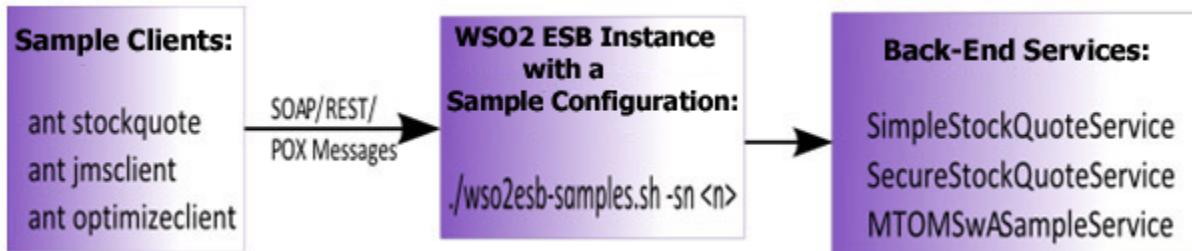
```
C:\wso2\wso2esb\bin>ant
```

3. Run the ESB in the DEBUG mode. To switch from the default INFO log messages to DEBUG log messages, edit the <ESB\_HOME>/repository/conf/log4j.properties file and change the line log4j.categor

y.org.apache.synapse=INFO to log4j.category.org.apache.synapse=DEBUG.

## Understanding the samples

The ESB samples use several sample clients, ESB configurations and sample back-end services in order to explain different use cases. The diagram below depicts the interaction between these sample clients, the ESB and the services at a high level. The clients are able to send SOAP/REST or POX messages over transports such as HTTP/HTTPS or JMS with WS-Addressing, or WS-Security. They can send binary optimized content using MTOM, SwA, binary or plain text JMS messages. After mediation through the ESB, the requests are passed over to sample services.



## Starting the ESB with a sample configuration

### To start the WSO2 ESB with a selected sample configuration

1. Open a command prompt (or a shell in Linux) and go to the <ESB\_HOME>\bin directory.
2. Execute one of the following commands, where <n> denotes the number assigned to the sample.
  - On Windows: wso2esb-samples.bat -sn <n>
  - On Linux/Solaris: ./wso2esb-samples.sh -sn <n>

For example, to start the ESB with the Sample 0 configuration on Linux/Solaris, run the following command:

```
./wso2esb-samples.sh -sn 0
```

The <ESB\_HOME>/repository/samples directory contains all the configuration files of the ESB samples as `synapse_sample_<n>.xml` files. Here again <n> denotes the sample number of the sample you are running.

For example, the configuration file for Sample 0 is `synapse_sample_0.xml`.

## Deploying sample back-end services

The sample back-end services come with a pre-configured Axis2 server. These sample services demonstrate in-only and in-out SOAP/REST or POX messaging over HTTP/HTTPS and JMS transports using WS-Addressing, and WS-Security. The samples handle binary content using MTOM and SwA.

Each back-end sample service can be found in a separate folder in the <ESB\_HOME>/samples/axis2Server/src directory. You need to compile, build and deploy each back-end service to the Axis2 server.

### To build and deploy a back-end service

1. Open a command prompt (or a shell in Linux) and go to the required sample folder in the <ESB\_HOME>/samples/axis2Server/src directory.
2. Run ant from the selected sample directory.

For example, to build and deploy **SimpleStockQuoteService**, run the ant command from the <ESB\_HOME>

/samples/axis2Server/src/SimpleStockQuoteService directory, as follows:

```
user@host:/tmp/wso2esb-2.0/samples/axis2Server/src/SimpleStockQuoteService$ ant
Buildfile: build.xml
...
build-service:
 ...
 [jar] Building jar:
/tmp/wso2esb-2.0/samples/axis2Server/repository/services/SimpleStockQuoteService.
aar

BUILD SUCCESSFUL
Total time: 3 seconds
```

### **Sample back-end services explained**

#### **1. SimpleStockQuoteService**

This service has four operations:

- `getQuote` (in-out) - Generates a sample stock quote for a given symbol.
- `getFullQuote` (in-out) - Generates a history of stock quotes for a symbol for a number of days.
- `getMarketActivity` (in-out) - Returns stock quotes for a list of given symbols.
- `placeOrder` (in-only) - Accepts a one way message for an order.

#### **2. SecureStockQuoteService**

This service is a clone of the `SimpleStockQuoteService`, but has WS-Security enabled as well as an attached security policy for signing and encrypting messages.

#### **3. MTOMSwASampleService**

This service has three operations:

- `uploadFileUsingMTOM` (in-out) - Accepts a binary image from the SOAP request as MTOM, and returns this image back again as the response.
- `uploadFileUsingSwA` (in-out) - Accepts a binary image from the SOAP request as SwA, and returns this image back again as the response.
- `oneWayUploadUsingMTOM` (in-only) - Saves the request message to the disk.

This service also demonstrates the use of MTOM and SwA.

### **Starting the Axis2 server**

For the ESB samples, a standalone Apache Axis2 Web services engine is used as the back-end server, which is bundled with the WSO2 ESB distribution by default.

Once each back-end service is deployed to the Axis2 server, you need to start the Axis2 server before executing the sample client.

### **To start the Axis2 server**

1. Open a command prompt (or a shell in Linux) and go to the <ESB\_HOME>/samples/axis2Server directory.
2. Execute one of the following commands
  - On Windows: `axis2server.bat`
  - On Linux/Solaris: `./axis2server.sh`

This starts the Axis2 server with the HTTP transport listener on port 9000 and HTTPS on port 9002 respectively.

If you want to start **multiple instances** of the Axis2 server on different ports, run the following commands. Give unique names to each server.

```
./axis2server.sh -http 9001 -https 9005 -name MyServer1
./axis2server.sh -http 9002 -https 9006 -name MyServer2
./axis2server.sh -http 9003 -https 9007 -name MyServer3
```

Running the commands as specified above will start three instances of the Axis2 server on HTTP ports 9001, 9002 and 9003 respectively.

#### Configuring WSO2 ESB to use the JMS transport

To run some of the ESB samples, you need to configure WSO2 ESB's JMS transport with ActiveMQ 5.5.0 or higher. For instructions on configuring WSO2 ESB with ActiveMQ, see [Configure with ActiveMQ](#).

#### Configuring WSO2 ESB to use the mail transport

##### To enable the mail transport sender for the ESB samples

- Uncomment the mail transport sender configuration in the <ESB\_HOME>/repository/conf/axis2/axis2.xml file and make sure it points to a valid SMTP configuration for an actual scenario.

```
<transportSender name="mailto"
class="org.apache.synapse.transport.mail.MailTransportSender">
<parameter name="mail.smtp.host">smtp.gmail.com</parameter>
<parameter name="mail.smtp.port">587</parameter>
<parameter name="mail.smtp.starttls.enable">true</parameter>
<parameter name="mail.smtp.auth">true</parameter>
<parameter name="mail.smtp.user">synapse.demo.0</parameter>
<parameter name="mail.smtp.password">mailpassword</parameter>
<parameter name="mail.smtp.from">synapse.demo.0@gmail.com</parameter>
</transportSender>
```

##### To enable the mail transport receiver for the ESB samples

- Uncomment the mail transport receiver configuration in the <ESB\_HOME>/repository/conf/axis2/axis2.xml file.

```
<transportReceiver name="mailto"
class="org.apache.axis2.transport.mail.MailTransportListener">
</transportReceiver>
```

You need to provide correct parameters for a valid mail account at the service level.

#### Configuring WSO2 ESB to use the NHTTP transport

WSO2 ESB uses the HTTP PassThrough Transport (PTT) as the default HTTP transport. Therefore, the default axis2.xml file in <ESB\_HOME>/repository/conf/axis2 is PassThrough Transport enabled.

To run some of the ESB samples, you need to configure the ESB to use the NHTTP transport as the default transport.

### To enable the NHTTP transport

1. Rename the axis2.xml file in the <ESB\_HOME>/repository/conf/axis2 directory to default\_axis2.xml.
  2. Rename the axis2\_nhttp.xml file in the <ESB\_HOME>/repository/conf/axis2 directory to axis2.xml.
  3. Restart the ESB server.
- 

### Configuring WSO2 ESB to use the FIX transport

To run the FIX samples, you need to have a local Quickfix/J installation. To download the latest version of Quickfix/J, go to <http://www.quickfixj.org/downloads>

### To enable the FIX transport sender and receiver

- Uncomment the FIX transport sender and FIX transport receiver configurations in the <ESB\_HOME>/repository/conf/axis2/axis2.xml file.

### Configuring the ESB for FIX samples

In order to configure WSO2 ESB to run the FIX samples, you need to create the FIX configuration files.

You can find the config files in <ESB\_HOME>/repository/samples/resources/fix folder.

### To create the FIX configuration files

1. Put the following entries in a file named fix-synapse.cfg

```
[default]
FileStorePath=repository/logs/fix/store
FileLogPath=repository/logs/fix/log
ConnectionType=acceptor
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ValidOrderTypes=1,2,F
SenderCompID=SYNAPSE
TargetCompID=BANZAI
UseDataDictionary=Y
DefaultMarketPrice=12.30

[session]
BeginString=FIX.4.0
SocketAcceptPort=9876
```

2. Put the following entries in a file named synapse-sender.cfg

```
[default]
FileStorePath=repository/logs/fix/data
FileLogPath=repository/logs/fix/log
SocketConnectHost=localhost
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ReconnectInterval=5
SenderCompID=SYNAPSE
TargetCompID=EXEC
ConnectionType=initiator

[session]
BeginString=FIX.4.0
SocketConnectPort=19876
```

The `FileStorePath` property in the `fix-synapse.cfg` file and `synapse-sender.cfg` file should point to two different directories in your local file system. Once the samples are executed, Synapse will create FIX message stores in these two directories.

#### ***Configuring sample FIX applications***

If you use a binary distribution of Quickfix/J, the two samples and their configuration files are all packed to a single JAR file called `quickfixj-examples.jar`.

#### **To configure sample FIX applications**

1. Either modify the `quickfixj-examples.jar` file or pass the new configuration file as a command line parameter.

##### **To modify the `quickfixj-examples.jar` file:**

- Extract the JAR file, modify the configuration files and pack them to a JAR file with the same name again.

##### **To pass the new configuration file as a command line parameter:**

- Copy the config files from the `<ESB_HOME>/repository/samples/resources/fix` directory to the `<QFJ_HOME>/etc` directory. Then execute the sample apps from `<QFJ_HOME>/bin`,  
`./banzai.sh/bat ..../etc/banzai.cfg` `executor.sh/bat ..../etc/executor.cfg`.

2. Locate and edit the FIX configuration file of Executor to be as follows. This file is usually named `executor.cfg`

```
[default]
FileStorePath=examples/target/data/executor
ConnectionType=acceptor
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ValidOrderTypes=1,2,F
SenderCompID=EXEC
TargetCompID=SYNAPSE
UseDataDictionary=Y
DefaultMarketPrice=12.30

[session]
BeginString=FIX.4.0
SocketAcceptPort=19876
```

3. Locate and edit the FIX configuration file of Banzai to be as follows. This file is usually named `banzai.cfg`

```
[default]
FileStorePath=examples/target/data/banbai
ConnectionType=initiator
SenderCompID=BANZAI
TargetCompID=SYNAPSE
SocketConnectHost=localhost
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ReconnectInterval=5

[session]
BeginString=FIX.4.0
SocketConnectPort=9876
```

The `FileStorePath` property in the two files above should point to two different directories in your local file system. The launcher scripts for the sample application can be found in the `bin` directory of the Quickfix/J distribution.

For more information regarding the FIX sample applications, see <http://www.quickfixj.org/quickfixj/usermanual/1.5.0/usage/examples.html>

For more information on configuring Quickfix/J applications, see <http://www.quickfixj.org/quickfixj/usermanual/1.5.0/usage/configuration.html>

#### Configuring WSO2 ESB to use the VFS transport

To run some of the ESB samples, you need to configure the ESB to enable the VFS listener and the VFS sender. For instructions on configuring the ESB to use the VFS transport, see [Enable the VFS transport](#).

#### Configuring WSO2 ESB to use the AMQP transport

To configure WSO2 ESB to use the AMQP transport, you need to have QPid version 1.0-M2 or higher installed and started. You also need to copy the following client JAR files into the `<ESB_HOME>/repository/components/lib`

b directory to support AMQP. These files can be found in the lib directory of the QPid installation.

- qpid-client-1.0-incubating-M2.jar
- qpid-common-1.0-incubating-M2.jar
- geronimo-jms\_1.1\_spec-1.0.jar
- slf4j-api-1.4.0.jar \*\*
- slf4j-log4j12-1.4.0.jar \*\*

To configure FIX (Quickfix/J 1.3) with AMQP (QPid-1.0-M2), copy the s14j-\* libraries that come with QPid and ignore the s14j-\* libraries that come with Quickfix/J.

## To enable the AMQP transport over the JMS transport

1. Uncomment the JMS transport listener configuration.

### To enable AMQP over JMS for ESB:

- Update the <ESB\_HOME>/repository/conf/axis2/axis2.xml file by uncommenting the JMS transport listener configuration.

### To enable JMS support for the sample Axis2 server:

- Update the <ESB\_HOME>/samples/axis2Server/repository/conf/axis2.xml file by uncommenting the JMS transport listener configuration.

```
<\!--Uncomment this and configure as appropriate for JMS transport support, after
setting up your JMS environment \-->
<transportReceiver name="jms">
</transportReceiver>

<transportSender name="jms">
</transportReceiver>
```

2. Locate and edit the AMQP connection settings file for the message consumer. This file is named direct.properties and can be found in the <ESB\_HOME>/repository/samples/resources/fix directory.

```
java.naming.factory.initial =
org.apache.qpid.jndi.PropertiesFileInitialContextFactory
register some connection factories
connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.qpidConnectionfactory =
amqp://guest:guest@clientid/test?brokerlist='tcp://localhost:5672'
Register an AMQP destination in JNDI
destination.[jniName] = [BindingURL]
destination.directQueue =
direct://amq.direct//QpidStockQuoteService?routingkey='QpidStockQuoteService'
destination.replyQueue = direct://amq.direct//replyQueue?routingkey='replyQueue'
```

3. Locate and edit the AMQP connection settings file for WSO2 ESB. This file is named conn.properties and can be found in the <ESB\_HOME>/repository/samples/resources/fix directory.

```
#initial context factory
\#java.naming.factory.initial
=org.apache.qpid.jndi.PropertiesFileInitialContextFactory
register some connection factories
connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.qpidConnectionfactory=amqp://guest:guest@clientid/test?brokerlist='tcp://localhost:5672'
Register an AMQP destination in JNDI
destination.[jndiName] = [BindingURL]
destination.directQueue=direct://amq.direct//QpidStockQuoteService
```

## Configuring WSO2 ESB to use the TCP transport

### To enable the **TCP transport** for samples

- Open the <ESB\_HOME>/repository/conf/axis2/axis2.xml file in a text editor and add the following transport receiver configuration and sender configuration.

```
<transportReceiver name="tcp"
class="org.apache.axis2.transport.tcp.TCPTTransportListener">
<parameter name="transport.tcp.port">6060</parameter>
</transportReceiver>

<transportSender name="tcp"
class="org.apache.axis2.transport.tcp.TCPTTransportSender" />
```

If you want to use the sample Axis2 client to send TCP messages, you have to uncomment the TCP transport sender configuration in the <ESB\_HOME>/samples/axis2Client/client\_repo/conf/axis2.xml file.

## Configuring WSO2 ESB to use the UDP transport

### To enable the **UDP transport** for samples

- Open the <ESB\_HOME>/repository/conf/axis2/axis2.xml file in a text editor and add the following transport configurations.

```
<transportReceiver name="udp" class="org.apache.axis2.transport.udp.UDPListener"/>
<transportSender name="udp" class="org.apache.axis2.transport.udp.UDPSender" />
```

If you want to use the sample Axis2 client to send UDP messages, add the UDP transport sender configuration in the <ESB\_HOME>/samples/axis2Client/client\_repo/conf/axis2.xml file.

## Configuring the ESB for script mediator support

The **Script mediator** is a Synapse extension, and thus all pre-requisites for all BSF supported scripting languages may not be bundled by default with the ESB distribution. Before you use some script mediators, you need to manually add the required JAR files to the <ESB\_HOME>/repository/components/lib directory, and optionally perform other installation tasks as required by the individual scripting language. The following section describes this in detail.

### **JavaScript support**

The JavaScript/E4X support is enabled by default and comes ready-to-use with the WSO2 ESB distribution.

### **Ruby support**

For Ruby support you need to install **WSO2 Carbon - JRuby Engine for Mediation**. This is available in the WSO2 P2 repository. Use the WSO2 Carbon Component Manager UI in the management console to connect to the WSO2 P2 repository and install **WSO2 Carbon - JRuby Engine for Mediation**. This will install a JRuby 1.3 engine in the ESB runtime.

Alternatively, you can also download and install the JRuby engine manually. Download the `jruby-complete-1.3.0.wso2v1.jar` file from the WSO2 P2 repository and copy it into the `<ESB_HOME>/repository/components/dropins` directory.

---

### **Setting up the databases**

To run the samples that involve database integration, you need to setup the databases, sample tables and reusable data sources as required by the samples. Most samples require a remote Derby database, whereas a few samples require a MySQL database.

#### **Setting up with Apache Derby**

For instructions on installing and configuring the remote Derby database, see [Setting up the remote Derby database](#).

When the database is installed and configured, you can create the sample tables and insert sample data into the tables.

#### **To create a new table in the database**

- Execute the following statement.

```
CREATE table company(name varchar(10) primary key, id varchar(10), price double);
```

#### **To insert sample data into the table**

- Execute the following statement.

```
INSERT into company values ('IBM','c1',0.0);
INSERT into company values ('SUN','c2',0.0);
INSERT into company values ('MSFT','c3',0.0);
```

#### **Setting up with MySQL**

For instructions on installing and configuring the MySQL database, see [Setting up the MySQL database](#).

When the database is installed and configured, you can create the sample tables, insert sample data into the tables and create the two stored procedures.

#### **To create a new table in the database**

- Execute the following statement.

```
CREATE table company(name varchar(10) primary key, id varchar(10), price double);
```

### To insert sample data into the table

- Execute the following statements.

```
INSERT into company values ('IBM','c1',0.0);
INSERT into company values ('SUN','c2',0.0);
INSERT into company values ('MSFT','c3',0.0);
```

### To create the stored procedures

- Execute the following statements.

```
CREATE PROCEDURE getCompany(compName VARCHAR(10)) SELECT name, id, price FROM
company WHERE name = compName;
CREATE PROCEDURE updateCompany(compPrice DOUBLE,compName VARCHAR(10)) UPDATE
company SET price = compPrice WHERE name = compName;
```

---

### Setting up Synapse DataSources

Definition of a reusable database connection pool or datasources can be done using the `datasources.properties` file. It is possible to configure any number of datasources. Currently only two types of datasources are supported and those are based on the Apache DBCP datasources. The two types are `BasicDataSource` and `PerUserPoolDataSource` (based on Apache DBCP). The following configuration includes both definitions.

`datasources.properties` configuration

```

\#####
DataSources Configuration
\#####
synapse.datasources=lookupds,reportds
synapse.datasources.icFactory=com.sun.jndi.rmi.registry.RegistryContextFactory
synapse.datasources.providerPort=2199
If following property is present , then assumes that there is an external JNDI
provider and will not start a RMI registry
\#synapse.datasources.providerUrl=rmi://localhost:2199

synapse.datasources.lookupds.registry=Memory
synapse.datasources.lookupds.type=BasicDataSource
synapse.datasources.lookupds.driverClassName=org.apache.derby.jdbc.ClientDriver
synapse.datasources.lookupds.url=jdbc:derby://localhost:1527/lookupdb;create=false
Optionally you can specify a specific password provider implementation which
overrides any globally configured provider
synapse.datasources.lookupds.secretProvider=org.apache.synapse.commons.security.secret
.handler.SharedSecretCallbackHandler
synapse.datasources.lookupds.username=esb
Depending on the password provider used, you may have to use an encrypted password
here\!
synapse.datasources.lookupds.password=esb
synapse.datasources.lookupds.dsName=lookupdb
synapse.datasources.lookupds.maxActive=100
synapse.datasources.lookupds.maxIdle=20
synapse.datasources.lookupds.maxWait=10000

synapse.datasources.reportds.registry=JNDI
synapse.datasources.reportds.type=PerUserPoolDataSource
synapse.datasources.reportds.cpdsadapter.factory=org.apache.commons.dbcp.cpdsadapter.D
riverAdapterCPDS
synapse.datasources.reportds.cpdsadapter.className=org.apache.commons.dbcp.cpdsadapter
.DriverAdapterCPDS
synapse.datasources.reportds.cpdsadapter.name=cpds
synapse.datasources.reportds.dsName=reportdb
synapse.datasources.reportds.driverClassName=org.apache.derby.jdbc.ClientDriver
synapse.datasources.reportds.url=jdbc:derby://localhost:1527/reportdb;create=false
Optionally you can specify a specific password provider implementation which
overrides any globally configured provider
synapse.datasources.reportds.secretProvider=org.apache.synapse.commons.security.secret
.handler.SharedSecretCallbackHandler
synapse.datasources.reportds.username=esb
Depending on the password provider used, you may have to use an encrypted password
here\!
synapse.datasources.reportds.password=esb
synapse.datasources.reportds.maxActive=100
synapse.datasources.reportds.maxIdle=20
synapse.datasources.reportds.maxWait=10000

```

The configuration is similar to the log4j appender configuration.

It requires two databases, follow the steps in the section above to create the two databases `jdbc:derby://loca
lhost:1527/lookupdb` and `jdbc:derby://localhost:1527/reportdb` using the user name and
password as esb. Fill in the data for the two databases as described in the section above.

To secure data sources password, you should use the mechanism for securing secret Information. If that
mechanism is used, the passwords that have been specified are considered as aliases and those are used for

picking actual passwords. To get password securely, you should set the password provider for each data source. The password provider should be an implementation of the following:

- org.apache.synapse.commons.security.secret.SecretCallbackHandler.

## To run this sample

- Uncomment `secret-conf.properties`, `cipher-text.properties` and `datasources.properties`. These files can be found in the `<ESB_HOME>/repository/conf` directory.

## Using the Sample Clients

After [setting up the ESB samples](#), you can run them using the sample clients. The sample clients can be executed from the `<ESB_HOME>/samples/axis2Client` directory by specifying the relevant ant command.

You can execute ant from the `<ESB_HOME>/samples/axis2Client` directory, in order to view the available sample clients and some of the sample options used to configure them.

This section describes each of the following sample clients in detail.

- Stock Quote client
  - Smart client mode
  - Gateway / dumb client mode
  - Proxy client mode
- Generic JMS client
- Optimize client
- FIX client
- JSON client

### Stock Quote client

This is a simple SOAP client that can create and send stock quote requests, as well as receive and display the last sale price for a stock symbol.

The required stock symbol and operation can be specified as follows:

```
ant stockquote [-Dsymbol=IBM|MSFT|SUN|...]
[-Dmode=quote | customquote | fullquote | placeorder | marketactivity]
[-Daddurl=http://localhost:9000/soap/SimpleStockQuoteService]
[-Dtrpurl=http://localhost:8280]
[-Dprxurl=http://localhost:8280]
[-Drest=true]
[-Dwsrm=true]
[-Dpolicy=../../repository/samples/resources/policy/policy_1.xml]
```

The stock quote client is able to operate in the **smart client mode**, **gateway/dumb client mode** as well as **proxy client mode**, and can send the payloads listed below as SOAP messages:

- **quote** - Sends a quote request for a single stock as follows. The response contains the last sale price for the stock which would be displayed.

```
<m:getQuote xmlns:m="http://services.samples/xsd">
 <m:request>
 <m:symbol>IBM</m:symbol>
 </m:request>
</m:getQuote>
```

- **customquote** - Sends a quote request in a custom format. The ESB would transform this custom request into the standard stock quote request format and send it to the service. Upon receipt of the response, it would be transformed again to a custom response format and would be returned to the client, which will then display the last sales price.

```
<m0:checkPriceRequest xmlns:m0="http://services.samples/xsd">
 <m0:Code>symbol</m0:Code>
</m0:checkPriceRequest>
```

- **fullquote** - Gets quote reports for the stock over a number of days (for example, for the last 100 days of the year).

```
<m:getFullQuote xmlns:m="http://services.samples/xsd">
 <m:request>
 <m:symbol>IBM</m:symbol>
 </m:request>
</m:getFullQuote>
```

- **placeorder** - Places an order for stocks using a one way request.

```
<m:placeOrder xmlns:m="http://services.samples/xsd">
 <m:order>
 <m:price>3.141593E0</m:price>
 <m:quantity>4</m:quantity>
 <m:symbol>IBM</m:symbol>
 </m:order>
</m:placeOrder>
```

- **marketactivity** - Gets a market activity report for the day (for example, quotes for multiple symbols).

```
<m:getMarketActivity xmlns:m="http://services.samples/xsd">
 <m:request>
 <m:symbol>IBM</m:symbol>
 ...
 <m:symbol>MSFT</m:symbol>
 </m:request>
</m:getMarketActivity>
```

## Note

You can apply a WS-Policy to the request using the `policy` property in order to enforce QoS aspects such as WS-Security on the request. The policy specifies details such as timestamps, signatures, and encryption.

### **Smart client mode**

The `addurl` property sets the WS-Addressing EPR, and the `trpurl` sets a transport URL for a message. Therefore, by specifying both properties, the client can operate in the `smart client mode`, where the addressing EPR can specify the ultimate receiver, while the transport URL set to the ESB ensures that any mediation required takes place before the message is delivered to the ultimate receiver. For example:

```
ant stockquote -Daddurl=<addressingEPR> -Dtrpurl=<esb>
```

### **Gateway / dumb client mode**

By specifying only a transport URL, the client operates in the `dumb client mode`, where it sends the message to the ESB and depends on the ESB rules for proper mediation and routing of the message to the ultimate destination.

```
ant stockquote -Dtrpurl=<esb>
```

### **Proxy client mode**

The client uses the `prxurl` as an HTTP proxy to send the request. Therefore, by setting the `prxurl` to the ESB, the client can ensure that the message would reach the ESB for mediation. The client can optionally set a WS-Addressing EPR if required.

```
ant stockquote -Dprxurl=<esb> [-Daddurl=<addressingEPR>]
```

### **Generic JMS client**

This is a client that can send plain text, plain binary content, or POX content by directly publishing a JMS message to a specified destination. The JMS destination name should be specified with the `jms_dest` property. The `jms_type` property can be `text`, `binary` or `pox` to specify the type of message payload.

The plain text payload for a text message can be specified through the `payload` property. For binary messages, the `payload` property will contain the path to the binary file. For POX messages, the `payload` property will hold a stock symbol name to be used within the POX request for stock order placement request.

```
ant jmsclient [-Djms_type=text]
[-Djms_dest=dynamicQueues/JMSTextProxy][-Djms_payload="24.34 100 IBM"]
ant jmsclient [-Djms_type=pox] [-Djms_dest=dynamicQueues/JMSPoxProxy]
[-Djms_payload=MSFT]
ant jmsclient [-Djms_type=binary] [-Djms_dest=dynamicQueues/JMSFileUploadProxy]
[-Djms_payload=../../../../repository/samples/resources/mtom/asf-logo.gif]
```

### **Note**

The JMS client assumes the existence of a default ActiveMQ (5.2.0) installation on your local machine in order to run some of the samples.

## Optimize client

This is a client that can send a binary image file as a MTOM or SwA optimized message, and receive the same file again through the response and save it as a temporary file. The `opt_mode` can be `mtom` or `swa` respectively for the optimization. Optionally, the path to a custom file can be specified through the `opt_file` property, and the destination address can be changed through the `opt_url` property if required.

```
ant optimizeclient [-Dopt_mode=mtom | swa]
[-Dopt_url=http://localhost:8280/soap/MTOMSwASampleService]
[-Dopt_file=../../../../repository/samples/resources/mtom/asf-logo.gif]
```

## FIX client

This is a client that can post a FIX message of type **Order-Single** embedded into a SOAP message.

```
ant fixclient -Dsymbol=IBM -Dqty=5 -Dmode=buy
-Daddurl=http://localhost:8280/soap/FIXProxy
```

## JSON client

This is a client that can send get-quote requests using the JSON content interchange format over HTTP.

```
ant jsonclient
[-Daddurl=http://localhost:8280/services/JSONProxy]
[-Dsymbol=DELL]
```

## Message Mediation Samples

The following Message Mediation samples are available with WSO2 Enterprise Service Bus (ESB) :

- Sample 0: Introduction to ESB
- Sample 1: Simple Content-Based Routing (CBR) of Messages
- Sample 2: CBR with the Switch-Case Mediator Using Message Properties
- Sample 3: Local Registry Entry Definitions, Reusable Endpoints and Sequences
- Sample 4: Specifying a Fault Sequence with a Regular Mediation Sequence
- Sample 5: Creating SOAP Fault Messages and Changing the Direction of a Message
- Sample 6: Manipulating SOAP Headers and Filtering Incoming and Outgoing Messages
- Sample 7: Using Schema Validation and the Usage of Local Registry for Storing Configuration Metadata
- Sample 8: Introduction to Static and Dynamic Registry Resources and Using XSLT Transformations
- Sample 9: Introduction to Dynamic Sequences with the Registry
- Sample 10: Introduction to Dynamic Endpoints with the Registry
- Sample 11: Using a Full Registry-Based Configuration and Sharing a Configuration Between Multiple Instances
- Sample 12: One-Way Messaging in a Fire-and-Forget Mode through ESB
- Sample 13: Dual Channel Invocation Through Synapse
- Sample 14: Using Sequences and Endpoints as Local Registry Items
- Sample 15: Using the Enrich Mediator for Message Copying and Content Enrichment
- Sample 16: Introduction to Dynamic and Static Registry Keys
- Sample 17: Transforming / Replacing Message Content with PayloadFactory Mediator
- Sample 18: Transforming a Message Using ForEach Mediator

### Sample 0: Introduction to ESB

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

### ***Introduction***

This sample demonstrates how a message can be passed through the WSO2 ESB and logged before it is delivered to its receiver. This is a basic, introductory usecase of WSO2 ESB. In this sample, you will deploy a sample client and a sample back-end service, and then transfer a message to the back-end service from the client through the WSO2 ESB.

### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <!-- log all attributes of messages passing through -->
 <log level="full"/>
 <!-- Send the message to implicit destination -->
 <send/>
 </in>
 <out>
 <!-- log all attributes of messages passing through -->
 <log level="full"/>
 <!-- send the message back to the client -->
 <send/>
 </out>
 </sequence>
</definitions>
```

This configuration file `synapse_sample_0.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### **To build the sample**

1. Start the ESB with the sample 0 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "WSO2 Carbon started in n seconds."

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

### ***Executing the sample***

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

## To execute the sample client

- Run the following commands from the <ESB\_HOME>/samples/axis2Client directory, to execute the **Stock Quote Client** in its various modes.

To execute the **Stock Quote Client** in the **Smart Client Mode**, run the following command:

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/
```

To execute the **Stock Quote Client** in the **Proxy Client Mode**, run the following command:

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dprxurl=http://localhost:8280/
```

To execute the **Stock Quote Client** in the **Gateway/ Dumb Client Mode**:

See [Sample 1: Simple Content-Based Routing \(CBR\) of Messages](#).

Running each command specified above triggers a sample message to the back-end service. If the message is mediated successfully, an output is displayed on the start-up console of the Axis2 server.

### Analyzing the output

Analyze the output debug messages to understand the actions that are performed.

### Note

You can change the line `log4j.category.org.apache.synapse=INFO` to `log4j.category.org.apache.synapse=DEBUG` in the <ESB\_HOME>/repository/conf/log4j.properties file, to run the ESB in the DEBUG mode, in order to analyze the output debug messages.

When you analyze the output debug messages for the actions in the **Smart Client Mode**, you will see the client request arriving at the ESB with a WS-Addressing To header set to EPR <http://localhost:9000/services/SimpleStockQuoteService>. The ESB engine logs the message in the full log level (i.e, all the message headers and the body) and then sends the message to its implicit To address which is <http://localhost:9000/services/SimpleStockQuoteService>.

You will see a message as follows on the Axis2 server console confirming that the message is routed to the sample server, and that the sample service hosted at the sample server generates a stock quote for the requested symbol.

```
Sat Nov 18 21:01:23 IST 2006 SimpleStockQuoteService :: Generating quote for : IBM
```

The response message generated by the service is again received by the ESB and flows through the same mediation rules, which logs the response message and then sends it back to the client. On the client console you will see an output as follows, based on the message received by the client.

```
Standard :: Stock price = $95.26454380258552
```

When you analyze the output debug messages for the actions in the **Proxy Client Mode**, You will see exactly the same behavior as in the previous scenario. However, this time the difference is at the client, as it sends the

message to the WS-Addressing To address `http://localhost:9000/services/SimpleStockQuoteService`, but the transport specifies ESB as the HTTP Proxy.

### Sample 1: Simple Content-Based Routing (CBR) of Messages

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

#### **Introduction**

This sample demonstrates the simple content-based routing of messages where a message is passed through the ESB in the **Dumb Client Mode**. The ESB acts as a gateway to accept all messages, and then performs mediation and routing based on message properties or content.

#### **Prerequisites**

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

#### **Building the sample**

The XML configuration for this sample is as follows:

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <!-- filtering of messages with XPath and regex matches -->
 <filter source="get-property('To')" regex=".*/StockQuote.*">
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </filter>
 </in>
 <out>
 <send/>
 </out>
 </sequence>
</definitions>
```

This configuration file `synapse_sample_1.xml` is available in the `<ESB_HOME>/repository/samples` directory.

#### **To build the sample**

1. Start the ESB with the sample 1 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a

message to the back-end service through the ESB using a sample client.

#### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

#### **To execute the sample client**

- Run the following command from the <ESB\_HOME>/samples/axis2Client directory, to execute the **Stock Quote Client** in the **Dumb Client Mode**.

```
ant stockquote -Dtrpurl=http://localhost:8280/services/StockQuote
```

#### **Analyzing the output**

Analyze the output debug messages for the actions in the **Dumb Client Mode**.

You will see the ESB receiving a message for which the ESB is set as the ultimate receiver. ESB performs a match to the path /StockQuote based on the To EPR in the following location: <http://localhost:8280/services/StockQuote>

As the request matches the XPath expression of the [Filter Mediator](#), the Filter Mediator's child mediators execute. It sends the message to a different endpoint as specified by the endpoint definition. During response processing, the filter condition fails. Therefore, the implicit Send Mediator forwards the response back to the client.

### **Sample 2: CBR with the Switch-Case Mediator Using Message Properties**

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

#### **Introduction**

This sample demonstrates the functionality of a Switch-Case Mediator. A message is passed through the ESB in the **Smart Client Mode**. The ESB acts as a gateway to accept all messages, writes and reads local properties on a message instance and then performs mediation based on the message properties or content.

#### **Prerequisites**

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

#### **Building the sample**

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <switch source="//m0:getQuote/m0:request/m0:symbol"
xmlns:m0="http://services.samples">
 <case regex="IBM">
 <!-- the property mediator sets a local property on the *current*
message -->
 <property name="symbol" value="Great stock - IBM"/>
 </case>
 <case regex="MSFT">
 <property name="symbol" value="Are you sure? - MSFT"/>
 </case>
 <default>
 <!-- it is possible to assign the result of an XPath expression as
well -->
 <property name="symbol"
 expression="fn:concat('Normal Stock - ',
//m0:getQuote/m0:request/m0:symbol)"
 xmlns:m0="http://services.samples"/>
 </default>
 </switch>
 <log level="custom">
 <!-- the get-property() XPath extension function allows the lookup of
local message properties
as well as properties from the Axis2 or Transport contexts (i.e.
transport headers) -->
 <property name="symbol" expression="get-property('symbol')"/>
 <!-- the get-property() function supports the implicit message headers
To/From/Action/FaultTo/ReplyTo -->
 <property name="epr" expression="get-property('To')"/>
 </log>
 <!-- Send the messages where they are destined to (i.e. the 'To' EPR of
the message) -->
 <send/>
 </in>
 <out>
 <send/>
 </out>
 </sequence>
 </definitions>

```

This configuration file `synapse_sample_2.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 2 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "WSO2 Carbon started in n seconds."

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a

message to the back-end service through the ESB using a sample client.

#### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

#### **To execute the sample client**

- Run each of the following commands from the <ESB\_HOME>/samples/axis2Client directory, specifying IBM, MSFT and SUN as the stock symbols.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=IBM

ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=MSFT

ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=SUN
```

#### **Analyzing the output**

Analyze the mediation log on the ESB start-up console.

When the symbol **IBM** is requested, you will see that the case statements first case which is IBM in the `synapse_sample_2.xml` file is executed and a local property named **symbol** is set to **Great stock - IBM**. Subsequently, this local property value is looked up by the [Log Mediator](#) and logged using the `get-property()` XPath extension function.

The mediation log on the ESB start-up console will be as follows:

```
INFO LogMediator - symbol = Great stock - IBM, epr =
http://localhost:9000/axis2/services/SimpleStockQuoteService
```

When the symbol **MSFT** is requested, the mediation log on the ESB start-up console will be as follows:

```
INFO LogMediator - symbol = Are you sure? - MSFT, epr =
http://localhost:9000/axis2/services/SimpleStockQuoteService
```

When the symbol **SUN** is requested, the mediation log on the ESB start-up console will be as follows:

```
INFO LogMediator - symbol = Normal Stock - SUN, epr =
http://localhost:9000/axis2/services/SimpleStockQuoteService
```

### **Sample 3: Local Registry Entry Definitions, Reusable Endpoints and Sequences**

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

## Introduction

This sample demonstrates the functionality of local registry entry definitions, reusable endpoints and sequences.

## Prerequisites

For a list of prerequisites, see [Prerequisites to start the ESB samples](#).

## Building the sample

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <!-- define a string resource entry to the local registry -->
 <localEntry key="version">0.1</localEntry>
 <!-- define a reusable endpoint definition -->
 <endpoint name="simple">
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 <!-- define a reusable sequence -->
 <sequence name="stockquote">
 <!-- log the message using the custom log level. illustrates custom properties
for log -->
 <log level="custom">
 <property name="Text" value="Sending quote request"/>
 <property name="version" expression="get-property('version')"/>
 <property name="direction" expression="get-property('direction')"/>
 </log>
 <!-- send message to real endpoint referenced by key "simple" endpoint
definition -->
 <send>
 <endpoint key="simple"/>
 </send>
 </sequence>
 <sequence name="main">
 <in>
 <property name="direction" value="incoming"/>
 <sequence key="stockquote"/>
 </in>
 <out>
 <send/>
 </out>
 </sequence>
</definitions>
```

This configuration file `synapse_sample_3.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 3 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

### **Executing the sample**

This example uses a sequence named *main* that specifies the main mediation rules to be executed. This is equivalent to directly specifying the mediators of the main sequence within the `<definitions>` tag. This sample scenario is a recommended approach for non-trivial configurations.

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### **To execute the sample client**

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory, to trigger a sample message to the back-end service.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/
```

### **Analyzing the output**

Analyze the mediation log on the ESB start-up console.

You will see that a sequence named *main* is executed. Then, for the incoming message flow, the In Mediator executes and it calls the sequence named *stockquote*.

```
DEBUG SequenceMediator - Sequence mediator <main> :: mediate()
DEBUG InMediator - In mediator mediate()
DEBUG SequenceMediator - Sequence mediator <stockquote> :: mediate()
```

You will also see that the *stockquote* sequence is executed. and that the log mediator dumps a simple *text/string* property which is the result of an XPath evaluation, that picks up the key named *version*, and a second result of an XPath evaluation that picks up a local message property set previously by the property mediator. The `get-property()` XPath extension function is able to read message properties local to the current message, local or remote registry entries, Axis2 message context properties as well as transport headers. The local entry definition for *version* defines a simple *text/string* registry entry, which is visible to all messages that pass through ESB.

```
[HttpServerWorker-1] INFO LogMediator - Text = Sending quote request, version = 0.1,
direction = incoming
[HttpServerWorker-1] DEBUG SendMediator - Send mediator :: mediate()
[HttpServerWorker-1] DEBUG AddressEndpoint - Sending To:
http://localhost:9000/services/SimpleStockQuoteService
```

## **Sample 4: Specifying a Fault Sequence with a Regular Mediation Sequence**

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

### **Introduction**

This sample demonstrates how to specify a *fault* sequence with a regular mediation sequence. Here a message is

sent from the sample client to the back-end service through the ESB via 3 mediation options which are a success scenario, a failure scenario without error handling and a failure scenario with proper error handling.

#### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

#### ***Building the sample***

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <!-- the default fault handling sequence used by Synapse - named 'fault' -->
 <sequence name="fault">
 <log level="custom">
 <property name="text" value="An unexpected error occurred"/>
 <property name="message" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
 </sequence>
 <sequence name="sunErrorHandler">
 <log level="custom">
 <property name="text" value="An unexpected error occurred for stock SUN"/>
 <property name="message" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
 </sequence>
 <!-- default message handling sequence used by Synapse - named 'main' -->
 <sequence name="main">
 <in>
 <switch source="//m0:getQuote/m0:request/m0:symbol"
xmlns:m0="http://services.samples">
 <case regex="IBM">
 <send>
 <endpoint><address
uri="http://localhost:9000/services/SimpleStockQuoteService"/></endpoint>
 </send>
 </case>
 <case regex="MSFT">
 <send>
 <endpoint key="bogus"/>
 </send>
 </case>
 <case regex="SUN">
 <sequence key="sunSequence"/>
 </case>
 </switch>
 <drop/>
 </in>
 <out>
 <send/>
 </out>
 </sequence>
 <sequence name="sunSequence" onError="sunErrorHandler">
 <send>
 <endpoint key="sunPort"/>
 </send>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_4.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 4 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

- The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "WSO2 Carbon started in n seconds."
2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
  3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### **To execute the sample client**

- Run each of the following commands from the <ESB\_HOME>/samples/axis2Client directory to trigger sample messages to the back-end service.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=IBM

ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=MSFT

ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=SUN
```

### **Analyzing the output**

When the **IBM** stock quote is requested, the configuration routes it to the defined inline endpoint , which then routes the message to the **SimpleStockQuoteService** on the local Axis2 instance. Therefore, a valid response message is shown at the client.

When the **MSFT** stock quote is requested, the ESB is instructed to route the message to the endpoint defined as the *bogus* endpoint, which does not exist. This triggers a fault scenario.

#### **synapse\_sample\_4.xml - MSFT Quote**

```
<case regex="MSFT">
 <send>
 <endpoint key="bogus" />
 </send>
</case>
```

The ESB executes the specified error handler sequence closest to the point where the error was encountered. In this case, the currently-executing sequence is *main* and it does not specify an *onError* attribute. Whenever ESB cannot find an error handler, it looks for a sequence named *fault*. As a result, the *fault* sequence starts executing and it writes a generic error message to the log.

```
[HttpServerWorker-1] DEBUG SendMediator - Send mediator :: mediate()
[HttpServerWorker-1] ERROR IndirectEndpoint - Reference to non-existent endpoint for
key : bogus
[HttpServerWorker-1] DEBUG MediatorFaultHandler - MediatorFaultHandler :: handleFault
[HttpServerWorker-1] DEBUG SequenceMediator - Sequence mediator <fault> :: mediate()
[HttpServerWorker-1] DEBUG LogMediator - Log mediator :: mediate()
[HttpServerWorker-1] INFO LogMediator - text = An unexpected error occurred, message =
Reference to non-existent endpoint for key : bogus
```

When the **SUN** stock quote is requested, it invokes the `sunSequence` custom sequence which specifies `sunErrorHandler` as the error handler as you can see in the `synapse_sample_4.xml` file.

```
<case regex="SUN">
<sequence key="sunSequence" />
</case>
```

When the send fails, you will see a proper error handler invocation and the custom error message that is printed as follows.

```
[HttpServerWorker-1] DEBUG SequenceMediator - Sequence mediator <sunSequence> :: mediate()
[HttpServerWorker-1] DEBUG SequenceMediator - Setting the onError handler for the sequence
[HttpServerWorker-1] DEBUG AbstractListMediator - Implicit Sequence <SequenceMediator> :: mediate()
[HttpServerWorker-1] DEBUG SendMediator - Send mediator :: mediate()
[HttpServerWorker-1] ERROR IndirectEndpoint - Reference to non-existent endpoint for key : sunPort
[HttpServerWorker-1] DEBUG MediatorFaultHandler - MediatorFaultHandler :: handleFault
[HttpServerWorker-1] DEBUG SequenceMediator - Sequence mediator <sunErrorHandler> :: mediate()
[HttpServerWorker-1] DEBUG AbstractListMediator - Implicit Sequence <SequenceMediator> :: mediate()
[HttpServerWorker-1] DEBUG LogMediator - Log mediator :: mediate()
[HttpServerWorker-1] INFO LogMediator - text = An unexpected error occurred for stock SUN, message = Reference to non-existent endpoint for key : sunPort
```

## Sample 5: Creating SOAP Fault Messages and Changing the Direction of a Message

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### ***Introduction***

This sample demonstrates the functionality of the **Fault mediator** (also called the **Makefault mediator**). Here a message is sent from the sample client to the back-end service through the ESB via two faulty mediation options, and then appropriate SOAP error responses are sent back to the client using the **Send mediator**.

### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### **Building the sample**

The XML configuration for this sample is as follows:

```
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <sequence name="myFaultHandler">
 <makefault>
 <code value="tns:Receiver"
xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason expression="get-property('ERROR_MESSAGE')"/>
 </makefault>

 <property name="RESPONSE" value="true"/>
 <header name="To" expression="get-property('ReplyTo')"/>
 <send/>
 </sequence>

 <sequence name="main" onError="myFaultHandler">
 <in>
 <switch source="//m0:getQuote/m0:request/m0:symbol"
xmlns:m0="http://services.samples">
 <case regex="MSFT">
 <send>
 <endpoint><address
uri="http://bogus:9000/services/NonExistentStockQuoteService"/></endpoint>
 </send>
 </case>
 <case regex="SUN">
 <send>
 <endpoint><address
uri="http://localhost:9009/services/NonExistentStockQuoteService"/></endpoint>
 </send>
 </case>
 </switch>
 <drop/>
 </in>

 <out>
 <send/>
 </out>
 </sequence>
</definitions>
```

This configuration file `synapse_sample_5.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### **To build the sample**

1. Start the ESB with the sample 5 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).

- Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### **To execute the sample client**

- Run the following command from the <ESB\_HOME>/samples/axis2Client directory to trigger a **MSFT** stock quote request to the back-end service.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=MSFT
```

- Run the following command from the <ESB\_HOME>/samples/axis2Client directory to trigger a **SUN** stock quote request to the back-end service.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=SUN
```

### **Analyzing the output**

When the **MSFT** stock quote is requested, an unknown host exception is generated according to the XML configuration in `synapse_sample_5.xml`.

```
<case regex="MSFT">
 <send>
 <endpoint>
 <address uri="http://bogus:9000/services/NonExistentStockQuoteService"/>
 </endpoint>
 </send>
</case>
```

This error message is captured and returned to the original client as a SOAP fault. You will see the following response on the console.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
 <soapenv:Body>
 <soapenv:Fault>
 <faultcode
 xmlns:tns="http://www.w3.org/2003/05/soap-envelope">tns:Receiver</faultcode>
 <faultstring>Error connecting to the back end</faultstring>
 </soapenv:Fault>
 </soapenv:Body>
</soapenv:Envelope>
```

When the **SUN** stock quote is requested, a connection refused exception is generated according to the XML configuration in `synapse_sample_5.xml`.

This error message is captured and returned to the original client as a SOAP fault. You will see the following response on the console.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
 <soapenv:Body>
 <soapenv:Fault>
 <faultcode
 xmlns:tns="http://www.w3.org/2003/05/soap-envelope">tns:Receiver</faultcode>
 <faultstring>Error connecting to the back end</faultstring>
 </soapenv:Fault>
 </soapenv:Body>
</soapenv:Envelope>
```

## Sample 6: Manipulating SOAP Headers and Filtering Incoming and Outgoing Messages

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### ***Introduction***

This sample demonstrates how the header mediator can be used to manipulate SOAP headers and how the in/out mediators can be used to filter requests and responses.

### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <header name="To"
 value="http://localhost:9000/services/SimpleStockQuoteService"/>
 </in>
 <send/>
 </sequence>
</definitions>
```

This configuration file `synapse_sample_6.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### **To build the sample**

1. Start the ESB with the sample 6 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds*".

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

#### ***Executing the sample***

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

#### **To execute the sample client**

- Run the following commands from the <ESB\_HOME>/samples/axis2Client directory, to trigger the client in the **Dumb Client Mode**.

```
ant stockquote -Dtrpurl=http://localhost:8280/
```

#### ***Analyzing the output***

According to the `synapse_sample_6.xml` file the `To` EPR of the message is set to the ESB.

You will see that the [In Mediator](#) processes the incoming messages, and manipulates the `To` header to refer to the stock quote service on the sample Axis2 server. As a result, a request for a stock quote is sent.

## **Sample 7: Using Schema Validation and the Usage of Local Registry for Storing Configuration Metadata**

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

#### ***Introduction***

This sample demonstrates how to use the [Validate Mediator](#) for XML schema validation and the usage of the local registry (local entries) for storing configuration metadata. Here a message is sent from the sample client to the back-end service through the ESB and shows how a static XML fragment could be made available to the ESB's local registry. It is assumed that the resources defined in the local registry are static (never changes over the lifetime of the configuration) and may be specified as a source URL, inline text or inline XML.

#### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

#### ***Building the sample***

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <localEntry key="validate_schema">
 <xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema"
 xmlns="http://services.samples"
 elementFormDefault="qualified" attributeFormDefault="unqualified"
 targetNamespace="http://services.samples">
 <xss:element name="getQuote">
 <xss:complexType>
 <xss:sequence>
 <xss:element name="request">
 <xss:complexType>
 <xss:sequence>
 <xss:element name="stocksymbol" type="xs:string"/>
 </xss:sequence>
 </xss:complexType>
 </xss:element>
 </xss:sequence>
 </xss:complexType>
 </xss:element>
 </xss:schema>
 </localEntry>
 <sequence name="main">
 <in>
 <validate>
 <schema key="validate_schema"/>
 <on-fail>
 <!-- if the request does not validate against schema throw a fault
-->
 <makefault>
 <code value="tns:Receiver"
 xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason value="Invalid custom quote request"/>
 </makefault>
 </on-fail>
 </validate>
 </in>
 <send/>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_7.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 7 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

According to the configuration file `synapse_sample_7.xml`, the schema is made available under the `validate_schema` key. The [Validate Mediator](#) by default operates on the first child element of the SOAP body. You can specify an XPath expression using the `source` attribute to override this behaviour. Here, the Validate Mediator uses the `validate_schema` resource to validate the incoming message and if the message validation fails, it invokes the *on-fail* sequence of mediators.

### **To execute the sample client**

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/
```

### **Analyzing the output**

When you send the stock quote request, the schema validation fails and a fault is generated back with the message *Invalid custom quote request*. This is because the schema used in the example expects a slightly different message than what is created by the stock quote client. The schema used in the example expects a `stocksymbol` element instead of `symbol` to specify the stock symbol.

## **Sample 8: Introduction to Static and Dynamic Registry Resources and Using XSLT Transformations**

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### **Introduction**

This sample demonstrates the functionality of static and dynamic registry resources and the [XSLT Mediator](#). Here a message is sent from the sample client to the back-end service through the ESB using the XSLT Mediator to perform the transformations. The XSLT transformations are specified as registry resources.

### **Prerequisites**

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### **Building the sample**

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <!-- the SimpleURLRegistry allows access to a URL based registry (e.g. file:/// or
http://) -->
 <registry provider="org.wso2.carbon.mediation.registry.ESBRegistry">
 <!-- the root property of the simple URL registry helps resolve a resource URL
as root + key -->
 <parameter name="root">file:repository/samples/resources/</parameter>
 <!-- all resources loaded from the URL registry would be cached for this
number of milli seconds -->
 <parameter name="cachableDuration">15000</parameter>
 </registry>
 <!-- define the request processing XSLT resource as a static URL source -->
 <localEntry key="xslt-key-req"
src="file:repository/samples/resources/transform/transform.xslt"/>
 <sequence name="main">
 <in>
 <!-- transform the custom quote request into a standard quote request
expected by the service -->
 <xslt key="xslt-key-req"/>
 <send/>
 </in>
 <out>
 <!-- transform the standard response back into the custom format the
client expects -->
 <!-- the key is looked up in the remote registry and loaded as a 'dynamic'
registry resource -->
 <xslt key="transform/transform_back.xslt"/>
 <send/>
 </out>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_8.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 8 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

## Executing the sample

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

According to the configuration file `synapse_sample_8.xml`, the first resource `xslt-key-req` is specified as a local registry entry. Local entries do not place the resource on the registry, but simply make it available to the local configuration. If a local entry is defined with a key that already exists in the remote registry, the local entry will get a

higher preference and will override the remote resource.

In this example you will notice the new *registry* definition.

ESB comes with a simple URL-based registry implementation `SimpleURLRegistry`. During initialization of the registry, the `SimpleURLRegistry` expects to find a property named `root`, which specifies a prefix for the registry keys used later. When the `SimpleURLRegistry` is used, this root is prefixed to the entry keys to form the complete URL for the resource being looked up. The registry caches a resource once requested, and caches it internally for a specified duration. Once this period expires, if necessary, it will reload the meta information about the resource and reload its cached copy, the next time the resource is requested.

Therefore, the second XSLT resource key `transform/transform_back.xslt` concatenated with the `root` of the `SimpleURLRegistry` `file:repository/samples/resources/` forms the complete URL of the resource as `file:repository/samples/resources/transform/transform_back.xslt` and caches its value for a period of 15000 ms.

### To execute the sample client

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dmode=customquote
```

- Run the client again immediately using the above command (within 15 seconds of the first request).
- Leave the system idle for more than 15 seconds and run the client again using the command specified in step 1.
- Now edit the `<ESB_HOME>/repository/samples/resources/transform/transform_back.xslt` file and add a blank line at the end
- Run the client again using the command specified in step 1.

### Analyzing the output

When you analyze the debug log output on the ESB console, you will see that the incoming message is transformed into a standard stock quote request as expected by the `SimpleStockQuoteService` deployed on the local Axis2 instance by the XSLT Mediator. The XSLT Mediator uses Xalan-J to perform the transformations. It is possible to configure the underlying transformation engine using properties where necessary.

You will also see that the response from the `SimpleStockQuoteService` is converted back into the custom format as expected by the client during the out message processing.

During the response processing, the `SimpleURLRegistry` fetches the resource.

When the client is run for the second time, you will not see the resource being reloaded by the registry as the cached value would be still valid.

When the client is run after leaving the system idle for more than 15 seconds, you will see that the registry detects the expiry of the cached resource and that it checks the meta information about the resource to determine if the resource itself has changed and requires a fresh fetch from the source URL. If the meta data / version number indicates that a reload of the cached resource is not necessary (unless the resource itself actually changed), the updated meta information is used and the cache lease is extended as appropriate.

```
[HttpClientWorker-1] DEBUG AbstractRegistry - Cached object has expired for key :
transform/transform_back.xslt
[HttpClientWorker-1] DEBUG SimpleURLRegistry - Perform RegistryEntry lookup for key :
transform/transform_back.xslt
[HttpClientWorker-1] DEBUG AbstractRegistry - Expired version number is same as
current version in registry
[HttpClientWorker-1] DEBUG AbstractRegistry - Renew cache lease for another 15s
```

Once the `transform_back.xslt` file is edited and the client is run again, If the cache is expired, you will see the following debug log messages which shows that the resource is re-fetched from its URL by the registry.

```
[HttpClientWorker-1] DEBUG AbstractRegistry - Cached object has expired for key : transform/transform_back.xslt
[HttpClientWorker-1] DEBUG SimpleURLRegistry - Perform RegistryEntry lookup for key : transform/transform_back.xslt
```

The `SimpleURLRegistry` allows the resource to be cached and detects updates so that the changes can be reloaded without restarting the ESB instance.

### **Sample 9: Introduction to Dynamic Sequences with the Registry**

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

#### ***Introduction***

This sample demonstrates how you can achieve dynamic behavior of the WSO2 ESB by the use of a registry.

The ESB supports dynamic definitions for sequences, endpoints and configuration resources. Here a Synapse configuration is defined which references a sequence definition specified as a registry key. The registry key resolves to the actual content of the sequence which is loaded dynamically by the ESB at runtime and cached appropriately as per its definition in the registry. Once the cache expires, ESB rechecks the meta information for the definition, reloads the sequence definition if necessary, and caches it again.

#### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

#### ***Building the sample***

The XML configuration for this sample is as follows:

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <registry provider="org.wso2.carbon.mediation.registry.ESBRegistry">
 <parameter name="root">file:./repository/samples/resources/</parameter>
 <parameter name="cachableDuration">15000</parameter>
 </registry>
 <sequence name="main">
 <sequence key="sequence/dynamic_seq_1.xml" />
 </sequence>
</definitions>
```

This configuration file `synapse_sample_9.xml` is available in the `<ESB_HOME>/repository/samples` directory.

#### **To build the sample**

1. Start the ESB with the sample 9 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

#### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

#### **To execute the sample client**

1. Run the following command from the <ESB\_HOME>/samples/axis2Client directory.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/
```

2. Execute the client immediately again (within 15 seconds of the last execution) using the above command to make sure that the sequence is not reloaded.
3. Edit the sequence definition in <ESB\_HOME>/repository/samples/resources/sequence/dynamic\_seq\_1.xml, change the log message to *Test Message 2* and execute the client again.
4. Wait for more than 15 seconds since the original caching of the sequence and execute the client again.

#### **Analyzing the output**

When you run the client for the first time, ESB fetches the definition of the sequence from the registry and executes its rules. Analyze the debug log output on the ESB console, you will see the following:

```
[HttpServerWorker-1] DEBUG SequenceMediator - Sequence mediator <dynamic_sequence> :: mediate()
...
[HttpServerWorker-1] INFO LogMediator - message = *** Test Message 1 ***
```

When you execute the client immediately again (within 15 seconds of the last execution), you will notice that the sequence is not reloaded.

Once you edit the sequence definition and execute the client again, you will see that the new message is not displayed if you executed the client within 15 seconds of loading the resource for the first time. However, after the elapse of 15 seconds since the original caching of the sequence, you will notice that the new sequence is loaded and executed by Synapse from the following log message.

However, when you wait for 15 seconds since the original caching of the sequence and then execute the client, you will see that the new sequence is loaded and executed by the ESB. This can be seen by analyzing the following debug log output on the ESB console.

```
[HttpServerWorker-1] DEBUG SequenceMediator - Sequence mediator <dynamic_sequence> :: mediate()
...
[HttpServerWorker-1] INFO LogMediator - message = *** Test Message 2 ***
```

The cache timeout could be tuned appropriately by configuring the URL registry to suit the environment and the needs.

#### **Sample 10: Introduction to Dynamic Endpoints with the Registry**

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

### **Introduction**

This sample demonstrates the functionality of dynamic endpoints. Here you store your endpoints in the registry and refer to them.

### **Prerequisites**

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### **Building the sample**

The XML configuration for this sample is as follows:

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <registry provider="org.wso2.carbon.mediation.registry.ESBRegistry">
 <parameter name="root">file:repository/samples/resources/</parameter>
 <parameter name="cachableDuration">15000</parameter>
 </registry>
 <sequence name="main">
 <in>
 <send>
 <endpoint key="endpoint/dynamic_endpt_1.xml"/>
 </send>
 </in>
 <out>
 <send/>
 </out>
 </sequence>
</definitions>
```

This configuration file `synapse_sample_10.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### **To build the sample**

1. Start the ESB with the sample 10 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).
4. Run the following command to start a second Axis2 server on HTTP port 9001 and HTTPS port 9003.

```
./axis2server.sh -http 9001 -https 9003
```

### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

## To execute the sample client

- Run the following command from the <ESB\_HOME>/samples/axis2Client directory.

```
ant stockquote -Dtrpurl=http://localhost:8280/
```

- Execute the client immediately again (within 15 seconds of the last execution) using the above command.
- Edit the endpoint definition in <ESB\_HOME>/repository/samples/resources/endpoint/dynamic\_endpoint\_1.xml by changing the endpoint address to http://localhost:9001/services/SimpleStockQuoteService.

### **Analyzing the output**

When you execute the client for the first time, the message is routed to the SimpleStockQuoteService on the default Axis2 instance on HTTP port 9000.

When you execute the client immediately again, you will see that the endpoint is cached and reused by the ESB. A similar scenario is explained in [Sample 9: Introduction to Dynamic Sequences with the Registry](#).

When you edit the endpoint definition and execute the client again, you will see that the registry loads the new definition of the endpoint once the cache expires.

Once the registry loads the new definition of the endpoint, you will see that the messages are routed to the second sample Axis2 server on HTTP port 9001.

## **Sample 11: Using a Full Registry-Based Configuration and Sharing a Configuration Between Multiple Instances**

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

### **Introduction**

This sample demonstrates the use of a full registry-based ESB configuration, to start a remote configuration from multiple ESB instances in a clustered environment. Here the Synapse configuration of a given node hosting the ESB simply points to the registry and looks up the actual configuration by requesting the key synapse.xml.

Full registry-based configuration is not dynamic at the moment; it is not reloading itself.

### **Prerequisites**

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### **Building the sample**

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <registry provider="org.wso2.carbon.mediation.registry.ESBRegistry">
 <parameter name="root">file:./repository/samples/resources/</parameter>
 <parameter name="cachableDuration">15000</parameter>
 </registry>
</definitions>

```

This configuration file `synapse_sample_11.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### To build the sample

1. Start the ESB with the sample 11 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

### Executing the sample

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### To execute the sample client

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```

ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/

```

### Analyzing the output

When you analyze the debug log output on the ESB console once the client is executed, you will see the following:

```
[HttpServerWorker-1] INFO LogMediator - message = This is a dynamic ESB configuration
```

The actual `synapse.xml` that is loaded when the sample is run is:

```
<!-- a registry based Synapse configuration -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <log level="custom">
 <property name="message" value="This is a dynamic ESB configuration"/>
 </log>
 <send/>
 </sequence>
</definitions>
```

## Sample 12: One-Way Messaging in a Fire-and-Forget Mode through ESB

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### ***Introduction***

This sample demonstrates how one-way messaging can be done in a fire-and-forget mode through the ESB.

### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to start the ESB samples](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2" />
 <property name="OUT_ONLY" value="true"/>
 <send>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </in>
 </sequence>
</definitions>
```

This configuration file `synapse_sample_12.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### **To build the sample**

1. Start the ESB with the sample 12 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).

3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

#### ***Executing the sample***

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

This sample invokes the one-way `placeOrder` operation on the `SimpleStockQuoteService` using the custom client which uses the `Axis2 ServiceClient.fireAndForget()` API.

#### **To execute the sample client**

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dmode=placeorder
```

#### ***Analyzing the output***

Analyze the Axis2 server log.

You will see that the one-way message flows through the ESB into the sample Axis2 server instance, which reports the acceptance of the order as follows:

```
SimpleStockQuoteService :: Accepted order for : 7482 stocks of IBM at $
169.27205579038733
```

If you send the client request through TCPmon, you will see that the **SimpleStockQuoteService** replies to the ESB with a HTTP 202 reply and then the ESB in turn replies to the client with a HTTP 202 acknowledgement.

#### **Sample 13: Dual Channel Invocation Through Synapse**

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

#### ***Introduction***

This sample demonstrates dual channel messaging through synapse.

#### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

#### ***Building the sample***

1. Start the ESB with the sample 0 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "WSO2 Carbon started in n seconds."

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end

services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

### **Executing the sample**

This example invokes the `getQuote` operation of the `SimpleStockQuoteService` using the custom client which uses the Axis2 ServiceClient API with `useSeparateListener` set to true, so that the response comes through a different channel than the one that is used to send the request to a callback defined in the client.

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### **To execute the sample client**

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dmode=dualquote
```

### **Analyzing the output**

When you execute the client, you will see the dual channel invocation through Synapse into the sample Axis2 server instance, which reports the response back to the client over a different channel.

```
Response received to the callback
Standard dual channel :: Stock price = $57.16686934968289
```

If you send your client request through TCPmon, you will see that Synapse replies to the client with a HTTP 202 acknowledgment when you send the request. The communication between Synapse and the server happens on a single channel. You will get the response back from Synapse to the client's callback through a different channel, which cannot be observed through TCPmon.

Also note the `wsa:Reply-To` header similar to `http://localhost:8200/axis2/services/anonService2`. This implies that the reply is in a different channel listening on port 8200.

It is required to engage addressing when using the dual channel invocation since the `wsa:Reply-To` header is required.

## **Sample 14: Using Sequences and Endpoints as Local Registry Items**

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

### **Introduction**

This sample demonstrates how sequences and endpoints can be fetched from a local registry so that it is possible to have the sequences and endpoints as local registry entries including file entries.

### **Prerequisites**

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### **Building the sample**

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ws.apache.org/ns/synapse
http://synapse.apache.org/ns/2010/04/configuration/synapse_config.xsd">
 <localEntry key="local-enrty-ep-key"
 src="file:repository/samples/resources/endpoint/dynamic_endpt_1.xml"/>
 <localEntry key="local-enrty-sequence-key">
 <sequence name="dynamic_sequence">
 <log level="custom">
 <property name="message" value="*** Test Message 1 ***"/>
 </log>
 </sequence>
 <description/>
 </localEntry>
 <sequence name="main">
 <in>
 <sequence key="local-enrty-sequence-key"/>
 <send>
 <endpoint key="local-enrty-ep-key"/>
 </send>
 </in>
 <out>
 <send/>
 </out>
 </sequence>
</definitions>
```

This configuration file `synapse_sample_14.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### **To build the sample**

1. Start the ESB with the sample 14 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### **To execute the sample client**

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant stockquote -Dtrpurl=http://localhost:8280/
```

### **Analyzing the output**

When you analyze the debug log output, you will see that the log statement for the fetched sequence of the local entry and the endpoint is fetched from the specified file at runtime and is cached in the system.

## **Sample 15: Using the Enrich Mediator for Message Copying and Content Enrichment**

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### ***Introduction***

This sample demonstrates how to copy message content to a property as well as how to enrich a message from a given source.

### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <enrich>
 <source type="custom"
 xpath="//m0:getQuote/m0:request/m0:symbol/text()"
 xmlns:m0="http://services.samples"/>
 <target type="property" property="ORIGINAL_REQ"/>
 </enrich>
 <enrich>
 <source type="body"/>
 <target type="property" property="REQUEST_PAYLOAD"/>
 </enrich>

 <enrich>
 <source type="inline" key="init_req"/>
 <target xmlns:m0="http://services.samples"
 xpath="//m0:getQuote/m0:request/m0:symbol/text()"/>
 </enrich>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 <drop/>
 </in>
 <out>
 <header xmlns:urn="http://synapse.apache.org"
name="urn:lastTradeTimestamp" value="foo"/>
 <enrich>
 <source type="custom"

xpath="//ns:getQuoteResponse/ns:return/ax21:lastTradeTimestamp"
 xmlns:ns="http://services.samples"
 xmlns:ax21="http://services.samples/xsd"/>
 <target xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:urn="http://synapse.apache.org"

xpath="/soapenv:Envelope/soapenv:Header/urn:lastTradeTimestamp"/>
 </enrich>
 <log level="full"/>
 <log>
 <property name="Original Request Symbol"
expression="get-property('ORIGINAL_REQ')"/>
 <property name="Request Payload"
expression="get-property('REQUEST_PAYLOAD')"/>
 </log>
 <send/>
 </out>
 </sequence>
 <localEntry key="init_req">MSFT</localEntry>
 <localEntry key="price_req">
 <m0:symbol xmlns:m0="http://services.samples">MSFT</m0:symbol>
 </localEntry>
</definitions>

```

This configuration file `synapse_sample_15.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 15 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

## ***Executing the sample***

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

## To execute the sample client

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant stockquote -Dtrpurl=http://localhost:8280/services/StockQuote
```

## ***Analyzing the output***

When you analyze the debug log output on the ESB console, you will see that the original IBM request is changed to MSFT using the enrich mediator.

According to the configuration file `synapse_sample_15.xml`, the original payload and the symbol is stored as synapse properties. Once the reply comes to synapse, the `lastTradeTimestamp` value of the response is added as a soap header in order to show the functionality of the enrich mediator. Also the request payload and request symbol values are logged in the out sequence.

Different parts of the message are stored in or copied to properties inside the in-sequence. Just before sending the message to the a back-end service the request value is modified based on the local entry. Then in the out-sequence, the enrich mediator is used to enrich the soap header based on the `lastTradeTimestamp` value of the response.

## **Note**

You can try this sample with different local entries as the source with the correct target xpath values.

## **Sample 16: Introduction to Dynamic and Static Registry Keys**

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### ***Introduction***

This Sample demonstrates the use of dynamic keys with mediators. Here the **XSLT Mediator** is used to demonstrate the difference between the static and dynamic usage of keys.

### Prerequisites

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### Building the sample

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <!-- the SimpleURLRegistry allows access to a URL based registry (e.g. file:/// or
 http://) -->
 <registry provider="org.wso2.carbon.mediation.registry.ESBRegistry">
 <!-- the root property of the simple URL registry helps resolve a resource URL
 as root + key -->
 <parameter name="root">file:repository/samples/resources/</parameter>
 <!-- all resources loaded from the URL registry would be cached for this
 number of milli seconds -->
 <parameter name="cachableDuration">15000</parameter>
 </registry>
 <sequence name="main">
 <in>
 <!-- define the request processing XSLT resource as a property value -->
 <property name="symbol" value="transform/transform.xslt"/>
 <!-- {} denotes that this key is a dynamic one and it is not a static key
 -->
 <!-- use Xpath expression "get-property()" to evaluate real key from
 property -->
 <xslt key="{get-property('symbol')}" />
 </in>
 <out>
 <!-- transform the standard response back into the custom format the
 client expects -->
 <!-- the key is looked up in the remote registry using a static key -->
 <xslt key="transform/transform_back.xslt"/>
 </out>
 <send/>
 </sequence>
</definitions>
```

This configuration file `synapse_sample_16.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### To build the sample

1. Start the ESB with the sample 16 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

According to the configuration file `synapse_sample_16.xml`, the first registry resource `transform/transform.xslt` is set as a property value.

Inside the XSLT mediator, the local property value is looked up using the Xpath expression `get-property()`. Similarly, any XPath expression can be enclosed within curly braces to denote that it is a dynamic key. Then the mediator evaluates the real value for that expression.

The second XSLT resource `transform/transform_back.xslt` is simply used as a static key. It is not included within curly braces since the mediator directly uses the static value as the key.

### **To execute the sample client**

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dmode=customquote
```

### **Analyzing the output**

When you analyze the debug log output on the ESB console, you will see an output similar to that of [Sample 8: Introduction to Static and Dynamic Registry Resources and Using XSLT Transformations](#).

### **Note**

You can try this sample with different local entries as the source with the correct target XPath values.

## **Sample 17: Transforming / Replacing Message Content with PayloadFactory Mediator**

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### **Introduction**

This Sample demonstrates how the [PayloadFactory Mediator](#) can be used to perform transformations as an alternative to the XSLT mediator, which is demonstrated in [Sample 8: Introduction to Static and Dynamic Registry Resources and Using XSLT Transformations](#). In this sample, the ESB implements the message translator enterprise integration pattern, and acts as a translator between the client and the back-end server when mediating a message to the sample back-end server from a sample client.

In this scenario, there is a web service endpoint that has the `getQuote` operation that expects a `symbol` parameter. However, the client that invokes the service sends the `getQuote` operation with the `Code` parameter. Therefore, the request message will be sent in the following format:

```
<p:getquote xmlns:p="http://services.samples">
 <p:request>
 <p:code>IBM</p:code>
 </p:request>
</p:getquote>
```

The service expects the message in the following format:

```
<p:getquote xmlns:p="http://services.samples">
 <p:request>
 <p:symbol>IBM</p:symbol>
 </p:request>
</p:getquote>
```

Similarly, the service will send the response in the following format:

```
<m:checkpriceresponse xmlns:m="http://services.samples/xsd">
 <m:symbol>IBM</m:symbol>
 <m:last>84.76940826343248</m:last>
</m:checkpriceresponse>
```

The client expects the response in the following format:

```
<m:checkpriceresponse xmlns:m="http://services.samples/xsd">
 <m:code>IBM</m:code>
 <m:price>84.76940826343248</m:price>
</m:checkpriceresponse>
```

To resolve this discrepancy, the PayloadFactory mediator is used to transform the message into the request format required by the service and the response format required by the client.

### **Prerequisites**

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### **Building the sample**

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <!-- using payloadFactory mediator to transform the request message -->
 <payloadFactory media-type="xml">
 <format>
 <m:getQuote xmlns:m="http://services.samples">
 <m:request>
 <m:symbol>$1</m:symbol>
 </m:request>
 </m:getQuote>
 </format>
 <args>
 <arg xmlns:m0="http://services.samples" expression="//m0:Code"/>
 </args>
 </payloadFactory>
 </in>
 <out>
 <!-- using payloadFactory mediator to transform the response message -->
 <payloadFactory media-type="xml">
 <format>
 <m:CheckPriceResponse xmlns:m="http://services.samples/xsd">
 <m:Code>$1</m:Code>
 <m:Price>$2</m:Price>
 </m:CheckPriceResponse>
 </format>
 <args>
 <arg xmlns:m0="http://services.samples/xsd"
expression="//m0:symbol"/>
 <arg xmlns:m0="http://services.samples/xsd"
expression="//m0:last"/>
 </args>
 </payloadFactory>
 </out>
 <send/>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_17.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 17 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

## Executing the sample

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### To execute the sample client

- Run the following command from the <ESB\_HOME>/samples/axis2Client directory.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dmode=customquote
```

#### **Analyzing the output**

When you analyze the debug log output on the ESB console, you will see that the incoming message is transformed by the [PayloadFactory Mediator](#) into a standard stock quote request as expected by the **SimpleStockQuoteService** deployed on the Axis2 server.

`printf()` style formatting is used to configure the transformation performed by the mediator. Each argument in the mediator configuration can be a static value or an XPath expression.

When an expression is used, the argument value is fetched at runtime by evaluating the provided XPath expression against the existing SOAP message.

The response from the **SimpleStockQuoteService** is converted back into the custom format as expected by the client during the out message processing, once again using the [PayloadFactory Mediator](#).

### Sample 18: Transforming a Message Using ForEach Mediator

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

#### **Introduction**

This sample demonstrates how the [ForEach mediator](#) can be used to transform a payload.

#### **Prerequisites**

- To invoke the main sequence using `curl`, a request file should be created as shown below. This file is named `stockQuoteReq.xml` in this example.

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
 xmlns:m0="http://services.samples" xmlns:xsd="http://services.samples/xsd">
 <soap:Header/>
 <soap:Body>
 <m0:getQuote>
 <m0:request><m0:symbol>IBM</m0:symbol></m0:request>
 <m0:request><m0:symbol>WSO2</m0:symbol></m0:request>
 <m0:request><m0:symbol>MSFT</m0:symbol></m0:request>
 </m0:getQuote>
 </soap:Body>
</soap:Envelope>
```

- See [Prerequisites to Start the ESB Samples](#) for other prerequisites.

#### **Building the sample**

The XML configuration for this sample is as follows.

```
<definitions>
<sequence xmlns="http://ws.apache.org/ns/synapse" name="main">
 <in>
 <foreach xmlns:ns="http://org.apache.synapse/xsd"
 xmlns:m0="http://services.samples" expression="//m0:getQuote/m0:request">
 <sequence>
 <payloadFactory media-type="xml">
 <format>
 <m0:checkPriceRequest>
 <m0:code>$1</m0:code>
 </m0:checkPriceRequest>
 </format>
 <args>
 <arg expression="//m0:request/m0:symbol" evaluator="xml"/>
 </args>
 </payloadFactory>
 </sequence>
 </foreach>
 <log level="full"/>
 </in>
 <out/>
</sequence>
</definitions>
```

This configuration file `synapse_sample_18.xml` is available in the `<ESB_HOME>/repository/samples` directory.

To build the sample, start the ESB with the sample 18 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

### ***Executing the sample***

Invoke the main sequence using the following command. This command should be issued from the same location in which the `stockQuoteReq.xml` request file you created is saved.

```
curl -d @stockQuoteReq.xml -H "Content-Type: application/soap+xml; charset=UTF-8"
"http://localhost:8280/"
```

### ***Analyzing the output***

The [ForEach mediator](#) splits the message to four different elements based on the evaluation of the `//m0:getQuote/m0:request` expression. The [PayloadFactory mediator](#) produces a result as shown in the following console log.

```

<?xml version='1.0' encoding='utf-8'?>
<soap:Envelope xmlns:m0="http://services.samples"
 xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
 xmlns:xsd="http://services.samples/xsd">
 <soap:Body>
 <m0:getQuote>
 <m0:checkPriceRequest><m0:code>IBM</m0:code></m0:checkPriceRequest>
 <m0:checkPriceRequest><m0:code>WSO2</m0:code></m0:checkPriceRequest>
 <m0:checkPriceRequest><m0:code>MSFT</m0:code></m0:checkPriceRequest>
 </m0:getQuote>
 </soap:Body>
</soap:Envelope>

```

<m0:request><m0:symbol>IBM</m0:symbol></m0:request> in the request has been changed to <m0:checkPriceRequest><m0:code>IBM</m0:code></m0:checkPriceRequest>.

The expression can be changed to perform a transformation or similar selectively. For example if the expression is given as expression = "/>/m0:getQuote/m0:request[1]", only the first element will be transformed and the other elements will be left unchanged.

For more information on XPath expressions, see [XPath Syntax by w3schools.com](#).

## Advanced Mediation with Endpoints

The following advanced mediation samples are available with WSO2 Enterprise Service Bus (ESB) :

- Sample 50: POX to SOAP conversion
- Sample 51: MTOM and SwA Optimizations and Request/Response Correlation
- Sample 52: Using Load Balancing Endpoints to Handle Peak Loads
- Sample 53: Using Failover Endpoints to Handle Peak Loads
- Sample 54: Session Affinity Load Balancing between Three Endpoints
- Sample 55: Session Affinity Load Balancing between Failover Endpoints
- Sample 56: Using a WSDL Endpoint as the Target Endpoint
- Sample 57: Dynamic Load Balancing between Three Nodes
- Sample 58: Static Load Balancing between Three Nodes
- Sample 59: Weighted load balancing between 3 endpoints
- Sample 60: Routing a Message to a Static List of Recipients
- Sample 61: Routing a Message to a Dynamic List of Recipients
- Sample 62: Routing a Message to a Dynamic List of Recipients and Aggregating Responses

### Sample 50: POX to SOAP conversion

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

#### ***Introduction***

This sample demonstrates how you can convert a POX message to a SOAP request.

#### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

#### ***Building the sample***

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <!-- filtering of messages with XPath and regex matches -->
 <filter source="get-property('To')" regex=".*/StockQuote.*">
 <header name="Action" value="urn:getQuote"/>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" format="soap11"/>
 </endpoint>
 </send>
 </filter>
 </in>
 <out>
 <send/>
 </out>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_50.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 50 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

## *Executing the sample*

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

## To execute the sample client

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory, specifying that the request should be a REST request.

```
ant stockquote -Dtrpurl=http://localhost:8280/services/StockQuote -Drest=true
```

## *Analyzing the output*

The request sent by the client is as follows:

```

POST /services/StockQuote HTTP/1.1
Content-Type: application/xml; charset=UTF-8;action="urn:getQuote";
SOAPAction: urn:getQuote
User-Agent: Axis2
Host: 127.0.0.1
Transfer-Encoding: chunked

75
<m0:getQuote xmlns:m0="http://services.samples/xsd">
 <m0:request>
 <m0:symbol>IBM</m0:symbol>
 </m0:request>
</m0:getQuote>0

```

It is a HTTP REST request, which will be transformed into a SOAP request and forwarded to the stock quote service.

### Sample 51: MTOM and SwA Optimizations and Request/Response Correlation

- [Introduction](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

#### *Introduction*

This sample demonstrates how you can use content optimization mechanisms such as Message Transmission Optimization Mechanism (MTOM) and SOAP with Attachments (SwA) with the ESB.

By default ESB serializes binary data as Base64 encoded strings and sends them in the SOAP payload. MTOM and SwA define mechanisms over which files with binary content can be transmitted over SOAP web services.

## Prerequisites

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

#### *Building the sample*

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <filter source="get-property('Action')" regex="urn:uploadFileUsingMTOM">
 <property name="example" value="mtom"/>
 <send>
 <endpoint>
 <address
 uri="http://localhost:9000/services/MTOMSwASampleService" optimize="mtom"/>
 </endpoint>
 </send>
 </filter>
 <filter source="get-property('Action')" regex="urn:uploadFileUsingSwA">
 <property name="example" value="swa"/>
 <send>
 <endpoint>
 <address
 uri="http://localhost:9000/services/MTOMSwASampleService" optimize="swa"/>
 </endpoint>
 </send>
 </filter>
 </in>
 <out>
 <filter source="get-property('example')" regex="mtom">
 <property name="enableMTOM" value="true" scope="axis2"/>
 </filter>
 <filter source="get-property('example')" regex="swa">
 <property name="enableSwA" value="true" scope="axis2"/>
 </filter>
 <send/>
 </out>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_51.xml` is available in the `<ESB_HOME>/repository/samples` directory.

- `<property name="enableMTOM" value="true" scope="axis2"/>`  
When this is enabled, all outgoing messages will be serialized and sent as MTOM optimized MIME messages. You can override this configuration per service in the `services.xml` configuration file.
- `<property name="enableSwA" value="true" scope="axis2"/>`  
When this is enabled, incoming SwA messages are automatically identified by axis2.

The above properties can also be defined in `<ESB_HOME>/repository/conf/axis2/axis2.xml` file.

## To build the sample

1. Start the ESB with the sample 51 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service MTOMSwASampleService. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

### ***Executing the sample***

The sample client used here is the **Optimize Client**. For further details on this sample client, see [Optimize Client](#).

### **To execute the sample client**

1. Run the following command from the <ESB\_HOME>/samples/axis2Client directory, specifying MTOM optimization.

```
ant optimizeclient -Dopt_mode=mtom
```

2. Next, run the following command from the <ESB\_HOME>/samples/axis2Client director, specifying SwA optimization.

```
ant optimizeclient -Dopt_mode=swa
```

### ***Analyzing the output***

The configuration sets a local message context property, and forwards the message to `http://localhost:9000/services/MTOMSwASampleService` optimizing the binary content as MTOM. You can see the actual message sent over the http transport if required by sending this message through TCPMon.

During response processing, by checking the local message property, the ESB can identify the past information about the current message context, and can use this knowledge to transform the response back to the client in the same format as the original request.

When the client executes successfully, it will upload a file containing the ASF logo and will receive its response back again and save it into a temporary file.

When you analyze the log once the client is run specifying MTOM optimization, you will see an output as follows:

```
[java] Sending file : ../../repository/samples/resources/mtom/asf-logo.gif as MTOM
[java] Saved response to file : ../../work/temp/sampleClient/mtom-49258.gif
```

If you use TCPMon and send the message through it, you will see that the requests and responses sent are MTOM optimized or sent as http attachments as follows:

## MTOM

```

POST http://localhost:9000/services/MTOMSwASampleService HTTP/1.1
Host: 127.0.0.1
SOAPAction: urn:uploadFileUsingMTOM
Content-Type: multipart/related;
boundary=MIMEBoundaryurn_uuid_B94996494E1DD5F9B51177413845353;
type="application/xop+xml";
start=<0.urn:uuid:B94996494E1DD5F9B51177413845354@apache.org>;
start-info="text/xml"; charset=UTF-8
Transfer-Encoding: chunked
Connection: Keep-Alive
User-Agent: Synapse-HttpComponents-NIO

--MIMEBoundaryurn_uuid_B94996494E1DD5F9B51177413845353241
Content-Type: application/xop+xml; charset=UTF-8; type="text/xml"
Content-Transfer-Encoding: binary
Content-ID:
<0.urn:uuid:B94996494E1DD5F9B51177413845354@apache.org>221b1
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
 <soapenv:Body>
 <m0:uploadFileUsingMTOM xmlns:m0="http://www.apache-synapse.org/test">
 <m0:request>
 <m0:image>
 <xop:Include
href="cid:1.urn:uuid:78F94BC50B68D76FB41177413845003@apache.org"
xmlns:xop="http://www.w3.org/2004/08/xop/include" />
 </m0:image>
 </m0:request>
 </m0:uploadFileUsingMTOM>
 </soapenv:Body>
 </soapenv:Envelope>
--MIMEBoundaryurn_uuid_B94996494E1DD5F9B51177413845353217
Content-Type: image/gif
Content-Transfer-Encoding: binary
Content-ID:
<1.urn:uuid:78F94BC50B68D76FB41177413845003@apache.org>22800GIF89a... <<
binary content >>

```

When you analyze the log once the client is run specifying SwA optimization, you will see an output as follows:

```

[java] Sending file : ../../repository/samples/resources/mtom/asf-logo.gif as SwA
[java] Saved response to file : ../../work/temp/sampleClient/swa-47549.gif

```

If you use TCPMon and send the message through it, you will see that the requests and responses sent are SwA optimized or sent as http attachments as follows:

## SWA

```

POST http://localhost:9000/services/MTOMSwASampleService HTTP/1.1
Host: 127.0.0.1
SOAPAction: urn:uploadFileUsingSwA
Content-Type: multipart/related;
boundary=MIMEBoundaryurn_uuid_B94996494E1DD5F9B51177414170491; type="text/xml";
start=<0.urn:uuid:B94996494E1DD5F9B51177414170492@apache.org>; charset=UTF-8
Transfer-Encoding: chunked
Connection: Keep-Alive
User-Agent: Synapse-HttpComponents-NIO

--MIMEBoundaryurn_uuid_B94996494E1DD5F9B51177414170491225
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID:
<0.urn:uuid:B94996494E1DD5F9B51177414170492@apache.org>22159
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body>
<m0:uploadFileUsingSwA xmlns:m0="http://www.apache-synapse.org/test">
<m0:request>
<m0:imageId>urn:uuid:15FD2DA2584A32BF7C1177414169826</m0:imageId>
</m0:request>
</m0:uploadFileUsingSwA>
</soapenv:Body>

</soapenv:Envelope>22--34MIMEBoundaryurn_uuid_B94996494E1DD5F9B511774141704912
17
Content-Type: image/gif
Content-Transfer-Encoding: binary
Content-ID:
<urn:uuid:15FD2DA2584A32BF7C1177414169826>22800GIF89a... << binary content >>
```

## Sample 52: Using Load Balancing Endpoints to Handle Peak Loads

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### ***Introduction***

This sample demonstrates how you can handle peak load scenarios using load-balancing endpoints. In the configuration of this sample, three load balancing endpoints are used to handle the peak load by dividing the load equally among each other.

### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main" onError="errorHandler">
 <in>
 <send>
 <endpoint>
 <loadbalance>
 <endpoint>
 <address uri="http://localhost:9001/services/LBService1">
 <enableAddressing/>
 </address>
 </endpoint>
 <endpoint>
 <address uri="http://localhost:9002/services/LBService1">
 <enableAddressing/>
 </address>
 </endpoint>
 <endpoint>
 <address uri="http://localhost:9003/services/LBService1">
 <enableAddressing/>
 </address>
 </endpoint>
 </loadbalance>
 </endpoint>
 </send>
 <drop/>
 </in>
 <out>
 <!-- Send the messages where they have been sent (i.e. implicit To EPR)
-->
 <send/>
 </out>
 </sequence>
 <sequence name="errorHandler">
 <makefault response="true">
 <code value="tns:Receiver"
 xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason value="COULDN'T SEND THE MESSAGE TO THE SERVER."/>
 </makefault>
 <send/>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_52.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 52 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "WSO2 Carbon started in n seconds."

2. Start three instances of the sample Axis2 server on HTTP ports 9001, 9002 and 9003 and give unique names to each server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service `LoadbalanceFailoverService`. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

#### ***Executing the sample***

The sample client used here is the **Load Balance and Failover Client**.

#### **To execute the sample client**

1. Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant loadbalancefailover -Di=100
```

2. Run the client again using the above command without the `-Di=100` parameter.
3. While running the client, stop the server named `MyServer1`.
4. Restart `MyServer1`.

#### ***Analyzing the output***

When the client is run for the first time, the client sends 100 requests to the `LoadbalanceFailoverService` through the ESB. The ESB will distribute the load among the three endpoints specified in the configuration file in a round-robin manner. The `LoadbalanceFailoverService` appends the name of the server to the response, so that the client can determine which server has processed the message.

When you analyze the output on the client console, you will see that the requests are processed by three servers.

The output on the client console will be as follows:

```
[java] Request: 1 ==> Response from server: MyServer1
[java] Request: 2 ==> Response from server: MyServer2
[java] Request: 3 ==> Response from server: MyServer3
[java] Request: 4 ==> Response from server: MyServer1
[java] Request: 5 ==> Response from server: MyServer2
[java] Request: 6 ==> Response from server: MyServer3
[java] Request: 7 ==> Response from server: MyServer1
...

```

When you run the client without the `-Di=100` parameter, the client sends infinite requests. When you stop the server named `MyServer1` while running the client, the requests will be distributed only among `MyServer2` and `MyServer3`.

You can see this by analyzing the log output on the client console, which will be as follows:

```

...
[java] Request: 61 ==> Response from server: MyServer1
[java] Request: 62 ==> Response from server: MyServer2
[java] Request: 63 ==> Response from server: MyServer3
[java] Request: 64 ==> Response from server: MyServer2
[java] Request: 65 ==> Response from server: MyServer3
[java] Request: 66 ==> Response from server: MyServer2
[java] Request: 67 ==> Response from server: MyServer3

```

By analysing the above output, you will understand that MyServer1 was stopped after request 63. You can come to this conclusion because all requests coming after request 63 are distributed only among MyServer2 and MyServer3.

When you restart MyServer1, you will see that the requests are now sent again to all three servers roughly after 60 seconds. This is because <suspendDurationOnFailure> is specified as 60 seconds in the configuration file.

Since <suspendDurationOnFailure> is specified as 60 seconds, the load balance endpoint will suspend any failed child endpoint only for 60 seconds on detecting the failure.

### **Sample 53: Using Failover Endpoints to Handle Peak Loads**

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

#### ***Introduction***

This sample demonstrates how you can handle peak load scenarios using failover endpoints. Here three failover endpoints are used in order to handle the requests that are coming in.

#### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

#### ***Building the sample***

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main" onError="errorHandler">
 <in>
 <send>
 <endpoint>
 <failover>
 <endpoint>
 <address uri="http://localhost:9001/services/LBService1">
 <enableAddressing/>
 </address>
 </endpoint>
 <endpoint>
 <address uri="http://localhost:9002/services/LBService1">
 <enableAddressing/>
 </address>
 </endpoint>
 <endpoint>
 <address uri="http://localhost:9003/services/LBService1">
 <enableAddressing/>
 </address>
 </endpoint>
 </failover>
 </endpoint>
 </send><drop/>
 </in>
 <out>
 <!-- Send the messages where they have been sent (i.e. implicit To EPR)
-->
 <send/>
 </out>
 </sequence>
 <sequence name="errorHandler">
 <makefault>
 <code value="tns:Receiver"
 xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason value="COULDN'T SEND THE MESSAGE TO THE SERVER."/>
 </makefault>

 <header name="To" action="remove"/>
 <property name="RESPONSE" value="true"/>
 <send/>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_53.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 53 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "WSO2 Carbon started in n seconds."

2. Start three instances of the sample Axis2 server on HTTP ports 9001, 9002 and 9003 respectively and give unique names to each server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **LoadbalanceFailoverService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

#### **Executing the sample**

The sample client used here is the **Load Balance and Failover Client**.

#### **To execute the sample client**

1. Run the following command from the <ESB\_HOME>/samples/axis2Client directory, to send infinite requests.

```
ant loadbalancefailover
```

2. While the client is running, stop the server named MyServer1.
3. Stop the server named MyServer2.
4. Stop the server named MyServer3.

#### **Analyzing the output**

According to the configuration file for this sample, messages are sent with the failover behavior. Initially the server at port 9001 is treated as the primary server and the other two are treated as back up servers. Messages will always be directed only to the primary server. If the primary server fails, the next listed server is selected as the primary server. Therefore, messages are sent successfully as long as there is at least one active server. To test this, run the loadbalancefailover client to send infinite requests as follows:

```
ant loadbalancefailover
```

When you run the client, infinite requests will be sent and you will see that all requests are processed by MyServer1. This is because the requests are always directed only to the primary server.

Here, the server at port 9001 is treated as the primary server and the other two are treated as back up servers. Therefore, all requests are processed by the server at port 9001 which is named as MyServer1.

Once you stop MyServer1 and analyze the log output on the client console, you will see that all subsequent requests are processed by MyServer2.

If MyServer1 is stopped after the request 127, the log output will be as follows:

```
...
[java] Request: 125 ==> Response from server: MyServer1
[java] Request: 126 ==> Response from server: MyServer1
[java] Request: 127 ==> Response from server: MyServer1
[java] Request: 128 ==> Response from server: MyServer2
[java] Request: 129 ==> Response from server: MyServer2
[java] Request: 130 ==> Response from server: MyServer2
...
```

When all servers are stopped, the error sequence will be activated and you will see the following fault message on the client console.

```
[java] COULDN'T SEND THE MESSAGE TO THE SERVER.
```

When a server failure is detected, the failed server will be added once again to the active servers list after 60 seconds. This is because <suspendDurationOnFailure> is set to 60 in the configuration file. Therefore, if you have restarted any of the stopped servers and have stopped all other servers, messages will be directed to the server that is restarted.

## Sample 54: Session Affinity Load Balancing between Three Endpoints

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### ***Introduction***

This sample demonstrates how the ESB can handle load balancing with session affinity using simple client sessions. The configuration used here is the same as the configuration in sample 52, except that here the session type is specified as `simpleClientSession`. This is a client initiated session, which means that the client generates the session identifier and sends it with each request. In this sample session type, the client adds a SOAP header named `ClientID` containing the identifier of the client. The ESB binds this ID with a server on the first request and sends all successive requests containing that ID to the same server.

### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main" onError="errorHandler">
 <in>
 <send>
 <endpoint>
 <!-- specify the session as the simple client session provided by
Synapse for
 testing purpose -->
 <session type="simpleClientSession"/>

 <loadbalance>
 <endpoint>
 <address uri="http://localhost:9001/services/LBService1">
 <enableAddressing/>
 </address>
 </endpoint>
 <endpoint>
 <address uri="http://localhost:9002/services/LBService1">
 <enableAddressing/>
 </address>
 </endpoint>
 <endpoint>
 <address uri="http://localhost:9003/services/LBService1">
 <enableAddressing/>
 </address>
 </endpoint>
 </loadbalance>
 </endpoint>
 </send><drop/>
 </in>
 <out>
 <!-- Send the messages where they have been sent (i.e. implicit To EPR)
-->
 <send/>
 </out>
 </sequence>
 <sequence name="errorHandler">
 <makefault>
 <code value="tns:Receiver"
xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason value="COULDN'T SEND THE MESSAGE TO THE SERVER."/>
 </makefault>

 <header name="To" action="remove"/>
 <property name="RESPONSE" value="true"/>
 <send/>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_54.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 54 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

- The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "WSO2 Carbon started in n seconds."
2. Start three instances of the sample Axis2 server on HTTP ports 9001, 9002 and 9003 and give unique names to each server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
  3. Deploy the back-end service `LoadbalanceFailoverService`. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

#### ***Executing the sample***

The sample client used here is the **Load Balance and Failover Client**.

#### **To execute the sample client**

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant loadbalancefailover -Dmode=session
```

#### ***Analyzing the output***

When the client is run in the session mode, the client continuously sends requests with three different session IDs. Out of these three IDs one ID is selected randomly for each request. Then the client prints the session ID with the server that responds for each request.

When you analyze the output on the client console, you will see the client output for the first 10 requests, which will be as follows:

```
[java] Request: 1 Session number: 1 Response from server: MyServer3
[java] Request: 2 Session number: 2 Response from server: MyServer2
[java] Request: 3 Session number: 0 Response from server: MyServer1
[java] Request: 4 Session number: 2 Response from server: MyServer2
[java] Request: 5 Session number: 1 Response from server: MyServer3
[java] Request: 6 Session number: 2 Response from server: MyServer2
[java] Request: 7 Session number: 2 Response from server: MyServer2
[java] Request: 8 Session number: 1 Response from server: MyServer3
[java] Request: 9 Session number: 0 Response from server: MyServer1
[java] Request: 10 Session number: 0 Response from server: MyServer1
...

```

By analysing the above output, you will see that the session number 0 is always directed to the server named MyServer1. This means that the session number 0 is bound to MyServer1. Similarly, session 1 is always directed to MyServer3 and session 2 is always directed to MyServer2. This means that session 1 and 2 are bound to MyServer3 and MyServer2 respectively.

### **Sample 55: Session Affinity Load Balancing between Failover Endpoints**

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

#### ***Introduction***

This sample demonstrates how the ESB can handle session aware load balancing between failover endpoints.

The configuration in this sample also uses the session type `simpleClientSession` to bind session ID values to servers as in sample 54. The difference in this sample is that fail-over endpoints are specified as child endpoints of the load balance endpoint. Therefore, sessions are bound to the fail-over endpoints. The session specific data maintained in endpoint servers are replicated among the failover endpoints using a clustering mechanism. Therefore, if one endpoint bound to a session fails, successive requests for that session will be directed to the next endpoint in that failover group.

## **Prerequisites**

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

## *Building the sample*

The XML configuration for this sample is as follows:

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main" onError="errorHandler">
 <in>
 <send>
 <endpoint>
 <!-- specify the session as the simple client session provided by
Synapse for
 testing purpose -->
 <session type="simpleClientSession"/>
 <loadbalance>
 <endpoint>
 <failover>
 <endpoint>
 <address
uri="http://localhost:9001/services/LBService1">
 <enableAddressing/>
 </address>
 </endpoint>
 <endpoint>
 <address
uri="http://localhost:9002/services/LBService1">
 <enableAddressing/>
 </address>
 </endpoint>
 <failover>
 <endpoint>
 <enableAddressing/>
 </address>
 </endpoint>
 <endpoint>
 <failover>
 <endpoint>
 <address
uri="http://localhost:9003/services/LBService1">
 <enableAddressing/>
 </address>
 </endpoint>
 <endpoint>
 <address
uri="http://localhost:9004/services/LBService1">
 <enableAddressing/>
 </address>
 </endpoint>
 <failover>
 <endpoint>
 <enableAddressing/>
 </address>
 </endpoint>
 </failover>
 </endpoint>
 </loadbalance>
```

```
 </endpoint>
 </send>
 <drop/>
</in>
<out>
 <!-- Send the messages where they have been sent (i.e. implicit To EPR)
-->
 <send/>
</out>
</sequence>
<sequence name="errorHandler">
 <makefault>
 <code value="tns:Receiver"
xmlns:tns="http://www.w3.org/2003/05/soap-envelope" />
 <reason value="COULDN'T SEND THE MESSAGE TO THE SERVER." />
 </makefault>
```

```

 <send/>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_55.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### To build the sample

1. Start the ESB with the sample 55 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start four instances of the sample Axis2 server on HTTP ports 9001, 9002, 9003 and 9004 respectively and give unique names to each server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **LoadbalanceFailoverService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

### **Executing the sample**

The sample client used here is the **Load Balance and Failover Client**.

### To execute the sample client

1. Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant loadbalancefailover -Dmode=session
```

2. While the client is running, stop the server named MyServer1.

### **Analyzing the output**

When the client is run, you will see the following output on the client console:

```

...
[java] Request: 222 Session number: 0 Response from server: MyServer1
[java] Request: 223 Session number: 0 Response from server: MyServer1
[java] Request: 224 Session number: 1 Response from server: MyServer1
[java] Request: 225 Session number: 2 Response from server: MyServer3
[java] Request: 226 Session number: 0 Response from server: MyServer1
[java] Request: 227 Session number: 1 Response from server: MyServer1
[java] Request: 228 Session number: 2 Response from server: MyServer3
[java] Request: 229 Session number: 1 Response from server: MyServer1
[java] Request: 230 Session number: 1 Response from server: MyServer1
[java] Request: 231 Session number: 2 Response from server: MyServer3
...

```

By analysing the above output, you will see that the session 0 and session 1 are always directed to MyServer1 whereas session 2 is always directed to MyServer3. You will notice that none of the requests are directed to MyServer2 and MyServer4. This is because MyServer2 and MyServer4 are kept as backup servers by the failover endpoints.

When MyServer1 is stopped while running the sample, you will see that all successive requests for session 0 are directed to MyServer2, which is the backup server for MyServer1's failover group.

The output on the client console will be as follows:

```
...
[java] Request: 529 Session number: 2 Response from server: MyServer3
[java] Request: 530 Session number: 1 Response from server: MyServer1
[java] Request: 531 Session number: 0 Response from server: MyServer1
[java] Request: 532 Session number: 1 Response from server: MyServer1
[java] Request: 533 Session number: 1 Response from server: MyServer1
[java] Request: 534 Session number: 1 Response from server: MyServer1
[java] Request: 535 Session number: 0 Response from server: MyServer2
[java] Request: 536 Session number: 0 Response from server: MyServer2
[java] Request: 537 Session number: 0 Response from server: MyServer2
[java] Request: 538 Session number: 2 Response from server: MyServer3
[java] Request: 539 Session number: 0 Response from server: MyServer2
...
```

By analysing the above output, you will see that all requests for session 0 are directed to MyServer2 after the request 534. Therefore, you can come to the conclusion that MyServer1 was stopped after request 534.

## Sample 56: Using a WSDL Endpoint as the Target Endpoint

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### ***Introduction***

This sample demonstrates how you can use a WSDL endpoint as the target endpoint. The configuration in this sample uses a WSDL endpoint inside the send mediator. This WSDL endpoint extracts the target endpoint reference from the WSDL document specified in the configuration. In this configuration the WSDL document is specified as a URI.

### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <send>
 <!-- get epr from the given wsdl -->
 <endpoint>
 <wsdl
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"
service="SimpleStockQuoteService"
port="SimpleStockQuoteServiceHttpSoap11Endpoint"/>
 </endpoint>
 </send>
 </in>
 <out>
 <send/>
 </out>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_56.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### To build the sample

1. Start the ESB with the sample 56 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service `SimpleStockQuoteService`. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

### Executing the sample

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### To execute the sample client

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant stockquote -Dsymbol=IBM -Dmode=quote -Daddurl=http://localhost:8280
```

### Analyzing the output

When the client is run, you will see the following output on the client console:

```
Standard :: Stock price = $95.26454380258552
```

According to `synapse_sample_56.xml` the WSDL endpoint inside the send mediator extracts the EPR from the WSDL document. Since WSDL documents can have many services and many ports inside each service, the service and port of the required endpoint has to be specified in the configuration via the `service` and `port` attributes respectively. When it comes to address endpoints, the QoS parameters for the endpoint can be specified in the

configuration. An excerpt taken from `sample_proxy_1.wsdl`, which is the WSDL document used in `synapse_sample_56.xml` is given below.

```

<wsdl:service name="SimpleStockQuoteService">
 <wsdl:port name="SimpleStockQuoteServiceHttpSoap11Endpoint"
binding="ns:SimpleStockQuoteServiceSoap11Binding">
 <soap:address
location="http://localhost:9000/services/SimpleStockQuoteService.SimpleStockQuoteServiceHttpSoap11Endpoint"/>
 </wsdl:port>
 <wsdl:port name="SimpleStockQuoteServiceHttpSoap12Endpoint"
binding="ns:SimpleStockQuoteServiceSoap12Binding">
 <soap12:address
location="http://localhost:9000/services/SimpleStockQuoteService.SimpleStockQuoteServiceHttpSoap12Endpoint"/>
 </wsdl:port>
</wsdl:service>

```

According to the above WSDL, the service and port specified in the configuration refers to the endpoint address `http://localhost:9000/services/SimpleStockQuoteService.SimpleStockQuoteServiceHttpSoap11Endpoint`

### Sample 57: Dynamic Load Balancing between Three Nodes

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

#### *Introduction*

This sample demonstrates how to use the dynamic load balancing mechanism of the ESB to support load balancing between three nodes.

#### *Prerequisites*

- Enable clustering and group management in the `<ESB_HOME>/repository/conf/axis2/axis2.xml` file.
  - . To do this,
    1. Set the `enable` attribute of the `clustering` and `groupManagement` elements to `true`.
    2. Provide the IP address of your machine as the value of the `mcastBindAddress` and `localMemberHost` parameters. For more information on clustering, see [WSO2 Clustering and Deployment Guide](#) .
- Enable clustering in the `<ESB_HOME>/samples/axis2Server/repository/conf/axis2.xml` file. To do this,
  1. Set the `enable` attribute of the `clustering` elements to `true`.
  2. Provide the IP address of your machine as the value of the `mcastBindAddress` and `localMemberHost` parameters.
  3. Make sure that the `applicationDomain` of the `membershipHandler` is the same as the domain name specified in the `axis2.xml` file of the Axis2 servers.
- For a list of general prerequisites, see [Prerequisites to Start the ESB Samples](#).

#### *Building the sample*

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main" onError="errorHandler">
 <in>
 <send>
 <endpoint name="dynamicLB">
 <dynamicLoadbalance failover="true"
algorithm="org.apache.synapse.endpoints.algorithms.RoundRobin">
 <membershipHandler
class="org.apache.synapse.core.axis2.Axis2LoadBalanceMembershipHandler">
 <property name="applicationDomain"
value="apache.axis2.application.domain"/>
 </membershipHandler>
 </dynamicLoadbalance>
 </endpoint>
 </send>
 <drop/>
 </in>
 <out>
 <!-- Send the messages where they have been sent (i.e. implicit To EPR)
-->
 <send/>
 </out>
 </sequence>
 <sequence name="errorHandler">
 <makefault response="true">
 <code value="tns:Receiver"
xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason value="COULDN'T SEND THE MESSAGE TO THE SERVER."/>
 </makefault>
 <send/>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_57.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 57 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start three instances of the sample Axis2 server on HTTP ports 9001, 9002 and 9003 respectively and give unique names to each server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service `LoadbalanceFailoverService`. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

## **Executing the sample**

The sample client used here is the **Load Balance and Failover Client**.

## **To execute the sample client**

1. Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant loadbalancefailover -Di=100
```

2. Run the client again using the above command without the `-Di=100` parameter.
3. While running the client, stop the server named MyServer1.
4. Restart MyServer1.

#### **Analyzing the output**

When the client is run for the first time, the client sends 100 requests to the `LoadbalanceFailoverService` through the ESB. The ESB will distribute the load among the three nodes in a round-robin manner. The `LoadbalanceFailoverService` appends the name of the server to the response, so that the client can determine which server has processed the message.

When you analyze the output on the client console, you will see that the requests are processed by three servers.

The output on the client console will be as follows:

```
[java] Request: 1 ==> Response from server: MyServer1
[java] Request: 2 ==> Response from server: MyServer2
[java] Request: 3 ==> Response from server: MyServer3
[java] Request: 4 ==> Response from server: MyServer1
[java] Request: 5 ==> Response from server: MyServer2
[java] Request: 6 ==> Response from server: MyServer3
[java] Request: 7 ==> Response from server: MyServer1
...
...
```

When you run the client without the `-Di=100` parameter, the client sends infinite requests. When you stop the server named MyServer1 while running the client, you will see that the requests are distributed only among MyServer2 and MyServer3.

You can see this by analyzing the log output on the client console, which will be as follows:

```
...
[java] Request: 61 ==> Response from server: MyServer1
[java] Request: 62 ==> Response from server: MyServer2
[java] Request: 63 ==> Response from server: MyServer3
[java] Request: 64 ==> Response from server: MyServer2
[java] Request: 65 ==> Response from server: MyServer3
[java] Request: 66 ==> Response from server: MyServer2
[java] Request: 67 ==> Response from server: MyServer3
...
...
```

By analysing the above output, you will understand that MyServer1 was stopped after request 63. You can come to this conclusion because all requests coming after request 63 are distributed only among MyServer2 and MyServer3.

When you restart MyServer1, you will see that the requests are now sent again to all three servers.

#### **Sample 58: Static Load Balancing between Three Nodes**

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

## Introduction

This sample demonstrates how to use the ESB to support static load balancing between three nodes when you have a set of statically configured nodes.

## Prerequisites

For a list of general prerequisites, see [Prerequisites to Start the ESB Samples](#)

## Building the sample

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main" onError="errorHandler">
 <in>
 <send>
 <endpoint>
 <loadbalance failover="true">
 <member hostName="127.0.0.1" httpPort="9001"
httpsPort="9005"/>
 <member hostName="127.0.0.1" httpPort="9002"
httpsPort="9006"/>
 <member hostName="127.0.0.1" httpPort="9003"
httpsPort="9007"/>
 </loadbalance>
 </endpoint>
 </send>
 <drop/>
 </in>
 <out>
 <!-- Send the messages where they have been sent (i.e. implicit To EPR)
-->
 <send/>
 </out>
 </sequence>
 <sequence name="errorHandler">
 <makefault response="true">
 <code value="tns:Receiver"
xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason value="COULDN'T SEND THE MESSAGE TO THE SERVER."/>
 </makefault>
 <send/>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_58.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 58 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start three instances of the sample Axis2 server on HTTP ports 9001, 9002 and 9003 respectively and give unique names to each server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).

3. Deploy the back-end service `LoadbalanceFailoverService`. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

#### **Executing the sample**

The sample client used here is the **Load Balance and Failover Client**.

#### **To execute the sample client**

1. Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant loadbalancefailover -Di=100
```

2. Run the client again using the above command without the `-Di=100` parameter.
3. While running the client, stop the server named `MyServer1`.
4. Restart `MyServer1`.

#### **Analyzing the output**

When the client is run for the first time, the client sends 100 requests to the `LoadbalanceFailoverService` through the ESB. The ESB will distribute the load among the three nodes in a round-robin manner. The `LoadbalanceFailoverService` appends the name of the server to the response, so that the client can determine which server has processed the message.

When you analyze the output on the client console, you will see that the requests are processed by three servers.

The output on the client console will be as follows:

```
[java] Request: 1 ==> Response from server: MyServer1
[java] Request: 2 ==> Response from server: MyServer2
[java] Request: 3 ==> Response from server: MyServer3
[java] Request: 4 ==> Response from server: MyServer1
[java] Request: 5 ==> Response from server: MyServer2
[java] Request: 6 ==> Response from server: MyServer3
[java] Request: 7 ==> Response from server: MyServer1
...
...
```

When you run the client without the `-Di=100` parameter, the client sends infinite requests. When you stop the server named `MyServer1` while running the client, you will see that the requests are distributed only among `MyServer2` and `MyServer3`.

You can see this by analyzing the log output on the client console, which will be as follows:

```
...
[java] Request: 61 ==> Response from server: MyServer1
[java] Request: 62 ==> Response from server: MyServer2
[java] Request: 63 ==> Response from server: MyServer3
[java] Request: 64 ==> Response from server: MyServer2
[java] Request: 65 ==> Response from server: MyServer3
[java] Request: 66 ==> Response from server: MyServer2
[java] Request: 67 ==> Response from server: MyServer3
...
...
```

By analysing the above output, you will understand that `MyServer1` was stopped after request 63. You can come to this conclusion because all requests coming after request 63 are distributed only among `MyServer2` and `MyServer3`.

When you restart MyServer1, you will see that the requests are now sent again to all three servers.

### Sample 59: Weighted load balancing between 3 endpoints

**Objective:** Demonstrate the weighted load balancing among a set of endpoints

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main" onError="errorHandler">
 <in>
 <send>
 <endpoint>
 <loadbalance

algorithm="org.apache.synapse.endpoints.algorithms.WeightedRoundRobin">
 <endpoint>
 <address uri="http://localhost:9001/services/LBService1">
 <enableAddressing/>
 <suspendOnFailure>
 <initialDuration>20000</initialDuration>
 <progressionFactor>1.0</progressionFactor>
 </suspendOnFailure>
 </address>
 <property name="loadbalance.weight" value="1"/>
 </endpoint>
 <endpoint>
 <address uri="http://localhost:9002/services/LBService1">
 <enableAddressing/>
 <suspendOnFailure>
 <initialDuration>20000</initialDuration>
 <progressionFactor>1.0</progressionFactor>
 </suspendOnFailure>
 </address>
 <property name="loadbalance.weight" value="2"/>
 </endpoint>
 <endpoint>
 <address uri="http://localhost:9003/services/LBService1">
 <enableAddressing/>
 <suspendOnFailure>
 <initialDuration>20000</initialDuration>
 <progressionFactor>1.0</progressionFactor>
 </suspendOnFailure>
 </address>
 <property name="loadbalance.weight" value="3"/>
 </endpoint>
 </loadbalance>
 </endpoint>
 </send>
 <drop/>
 </in>
 <out>
 <send/>
 </out>
</sequence>
<sequence name="errorHandler">
 <makefault response="true">
 <code xmlns:tns="http://www.w3.org/2003/05/soap-envelope"
value="tns:Receiver"/>
 <reason value="COULDN'T SEND THE MESSAGE TO THE SERVER."/>
 </makefault>
 <send/>
</sequence>
</definitions>

```

### Prerequisites:

- Start ESB with sample configuration 59: i.e. `wso2esb-samples.sh -sn 59`
- Deploy the `LoadbalanceFailoverService` and start three instances of sample Axis2 server as mentioned in sample 52.

Above configuration sends messages with the weighted loadbalance behaviour. Weight of each leaf address endpoint is defined by integer value of "loadbalance.weight" property associated with each endpoint. If weight of a endpoint is x, x number of requests will send to that endpoint before switch to next active endpoint.

To test this, run the `loadbalancefailover` client to send 100 requests as follows:

```
ant loadbalancefailover -Di=100
```

This client sends 100 requests to the `LoadbalanceFailoverService` through ESB. ESB will distribute the load among the three endpoints mentioned in the configuration in weighted round-robin manner. `LoadbalanceFailoverService` appends the name of the server to the response, so that client can determine which server has processed the message. If you examine the console output of the client, you can see that requests are processed by three servers as follows:

```
[java] Request: 1 ==> Response from server: MyServer1
[java] Request: 2 ==> Response from server: MyServer2
[java] Request: 3 ==> Response from server: MyServer2
[java] Request: 4 ==> Response from server: MyServer3
[java] Request: 5 ==> Response from server: MyServer3
[java] Request: 6 ==> Response from server: MyServer3
[java] Request: 7 ==> Response from server: MyServer1
[java] Request: 8 ==> Response from server: MyServer2
[java] Request: 9 ==> Response from server: MyServer2
[java] Request: 10 ==> Response from server: MyServer3
[java] Request: 11 ==> Response from server: MyServer3
[java] Request: 12 ==> Response from server: MyServer3
...
...
```

As logs, endpoint with weight 1 received a 1 request and endpoint with weight 2 received 2 requests and etc... in a cycle

### Sample 60: Routing a Message to a Static List of Recipients

Objective: Demonstrate message routing to a set of static endpoints.

### Prerequisites

Start ESB with the following sample configuration:

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main" onError="errorHandler">
 <in>
 <send>
 <endpoint>
 <!--List of Recipients (static)-->
 <recipientlist>
 <endpoint>
 <address
uri="http://localhost:9001/services/SimpleStockQuoteService" />
 </endpoint>
 <endpoint>
 <address
uri="http://localhost:9002/services/SimpleStockQuoteService" />
 </endpoint>
 <endpoint>
 <address
uri="http://localhost:9003/services/SimpleStockQuoteService" />
 </endpoint>
 </recipientlist>
 </endpoint>
 </send>
 <drop/>
 </in>
 <out>
 <log level="full"/>
 <drop/>
 </out>
 </sequence>
 <sequence name="errorHandler">
 <makefault response="true">
 <code xmlns:tns="http://www.w3.org/2003/05/soap-envelope"
value="tns:Receiver"/>
 <reason value="COULDN'T SEND THE MESSAGE TO THE SERVER."/>
 </makefault>
 <send/>
 </sequence>
 </definitions>

```

Deploy the SimpleStockQuoteService and start three instances of sample Axis2 server as mentioned in sample 52 Sessionless Load Balancing Between 3 Endpoints.

The above configuration routes a cloned copy of a message to each recipient defined within the static recipient list. To test this, run the StockQuote client to send an out-only message as follows:

```
ant stockquote -Dmode=placeorder -Dtrpurl=http://localhost:8280/
```

This client sends a request to the SimpleStockQuoteService through the ESB. ESB will create cloned copies of the message and route to the three endpoints mentioned in the configuration. SimpleStockQuoteService prints the details of the placed order. If you examine the console output of each server, you can see that requests are processed by the three servers as follows:

```
Accepted order #1 for : 15738 stocks of IBM at $ 185.51155223506518
```

Now shutdown MyServer1 and resend the request. You will observe that requests are still processed by MyServer2 and MyServer3.

### Sample 61: Routing a Message to a Dynamic List of Recipients

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)

#### ***Introduction***

This sample demonstrates message routing to a set of dynamic endpoints.

#### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

#### ***Building the sample***

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="errorHandler">
 <makefault response="true">
 <code xmlns:tns="http://www.w3.org/2003/05/soap-envelope"
value="tns:Receiver" />
 <reason value="COULDN'T SEND THE MESSAGE TO THE SERVER." />
 </makefault>
 <send />
 </sequence>
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default "fault" sequence"
/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')" />
 <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')" />
 </log>
 <drop />
 </sequence>
 <sequence name="main" onError="errorHandler">
 <in>
 <property name="EP_LIST"
value="http://localhost:9001/services/SimpleStockQuoteService,http://localhost:9002/se
rvices/SimpleStockQuoteService,http://localhost:9003/services/SimpleStockQuoteService"
/>
 <property name="OUT_ONLY" value="true" />
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2" />
 <send>
 <endpoint>
 <recipientlist>
 <endpoints value="{get-property('EP_LIST')}" max-cache="20" />
 </recipientlist>
 </endpoint>
 </send>
 <drop/>
 </in>
 </sequence>
 </definitions>

```

This configuration file `synapse_sample_61.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 61 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start three instances of the sample Axis2 server on HTTP ports 9001, 9002 and 9003 and give unique names to each server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

## **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

## To execute the sample client

- Run the following command from the <ESB\_HOME>/samples/axis2Client directory to send an out-only message:

```
ant stockquote -Dmode=placeorder -Dtrpurl=http://localhost:8280/
```

## Sample 62: Routing a Message to a Dynamic List of Recipients and Aggregating Responses

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

### ***Introduction***

This sample demonstrates message routing to a set of dynamic endpoints and aggregate responses.

### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="errorHandler">
 <makefault response="true">
 <code xmlns:tns="http://www.w3.org/2003/05/soap-envelope"
value="tns:Receiver" />
 <reason value="COULDN'T SEND THE MESSAGE TO THE SERVER." />
 </makefault>
 <send />
 </sequence>
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default "fault" sequence"
/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop />
 </sequence>
 <sequence name="main" onError="errorHandler">
 <in>
 <property name="EP_LIST"
value="http://localhost:9001/services/SimpleStockQuoteService,http://localhost:9002/se
rvices/SimpleStockQuoteService,http://localhost:9003/services/SimpleStockQuoteService"
/>
 <send>
 <endpoint>
 <recipientlist>
 <endpoints value="{get-property('EP_LIST')}" max-cache="20" />
 </recipientlist>
 </endpoint>
 </send>
 <drop/>
 </in>
 <out>
 <!--Aggregate responses-->
 <aggregate>
 <onComplete xmlns:m0="http://services.samples"
expression="//m0:getQuoteResponse">
 <log level="full"/>
 <send/>
 </onComplete>
 </aggregate>
 </out>
 </sequence>
 </definitions>

```

This configuration file `synapse_sample_62.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 62 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start three instances of the sample Axis2 server on HTTP ports 9001, 9002 and 9003 and give unique names

- to each server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#) .
- Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

#### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

#### **To execute the sample client**

- Run the following command from the <ESB\_HOME>/samples/axis2Client directory:

```
ant stockquote -Dtrpurl=http://localhost:8280/
```

#### **Analyzing the output**

When you have a look at the `synapse_sample_62.xml` configuration, you will see that it routes a cloned copy of a message to each recipient defined within the dynamic recipient list , and that each recipient responds back with a stock quote. When all the responses reach the ESB, the responses are aggregated to form the final response, which will be sent back to the client.

If you sent the client request through a TCP based conversation monitoring tool such as TCPMon, you will see the structure of the aggregated response message.

### **Quality of Service Samples in Message Mediation**

The following QoS sample in message mediation is available with WSO2 Enterprise Service Bus (ESB) :

- [Sample 100: Using WS-Security for Outgoing Messages](#)

#### **Sample 100: Using WS-Security for Outgoing Messages**

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

#### **Introduction**

This sample demonstrates how you can use the ESB to connect to endpoints with WS-Security for outgoing messages.

In this sample the stock quote client sends a request without WS-Security. The ESB is configured to enable WS-Security as per the policy specified in the `policy_3.xml` file, for outgoing messages to the `SecureStockQuoteService` endpoint hosted on the Axis2 instance.

#### **Prerequisites**

- Download and install the unlimited strength policy files for your JDK before using Apache Rampart. To download the policy files, go to <http://www.oracle.com/technetwork/java/javase/downloads/jce-6-download-429243.html>.
- For a list of general prerequisites, see [Prerequisites to Start the ESB Samples](#).

#### **Building the sample**

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <localEntry key="sec_policy"
src="file:repository/samples/resources/policy/policy_3.xml"/>
 <sequence name="main">
 <in>
 <send>
 <endpoint name="secure">
 <address
uri="http://localhost:9000/services/SecureStockQuoteService">
 <enableSec policy="sec_policy"/>
 <enableAddressing/>
 </address>
 </endpoint>
 </send>
 </in>
 <out>
 <header name="wsse:Security" action="remove"
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"/>
 <send/>
 </out>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_100.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### To build the sample

1. Start the ESB with the sample 100 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service `SecureStockQuoteService`. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

### Executing the sample

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### To execute the sample client

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant stockquote -Dtrpurl=http://localhost:8280/
```

### Analyzing the output

When you analyze the debug log output on the ESB console, you will see the encrypted message flowing to the service and the encrypted response being received by the ESB. You will also see that the `wsse:Security` header is removed from the decrypted message and the response is delivered back to the client, as expected.

If you use TCPMon and send the message through it, you will see the message sent by the ESB to the secure service as follows:

```

POST http://localhost:9001/services/SecureStockQuoteService HTTP/1.1
Host: 127.0.0.1
SOAPAction: urn:getQuote
Content-Type: text/xml; charset=UTF-8
Transfer-Encoding: chunked
Connection: Keep-Alive
User-Agent: Synapse-HttpComponents-NIO

800
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
xmlns:wsa="http://www.w3.org/2005/08/addressing" ...>
 <soapenv:Header>
 <wsse:Security ...>
 <wsu:Timestamp ...>
 ...
 </wsu:Timestamp>
 <xenc:EncryptedKey ...>
 ...
 </xenc:EncryptedKey>
 <wsse:BinarySecurityToken ...>
 <ds:SignedInfo>
 ...
 </ds:SignedInfo>
 <ds:SignatureValue>
 ...
 </ds:SignatureValue>
 <ds:KeyInfo Id="KeyId-29551621">
 ...
 </ds:KeyInfo>
 </ds:Signature>
 </wsse:Security>
 <wsa:To>http://localhost:9001/services/SecureStockQuoteService</wsa:To>
 <wsa:MessageID>urn:uuid:1C4CE88B8A1A9C09D91177500753443</wsa:MessageID>
 <wsa:Action>urn:getQuote</wsa:Action>
 </soapenv:Header>
 <soapenv:Body
 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" wsu:Id="Id-3789605">
 <xenc:EncryptedData Id="EncDataId-3789605" Type="http://www.w3.org/2001/04/xmlenc#Content">
 <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes256-cbc" />
 <xenc:CipherData>
 <xenc:CipherValue>Layg0xQcnH....6UKm5nKU6Qqr</xenc:CipherValue>
 </xenc:CipherData>
 </xenc:EncryptedData>
 </soapenv:Body>
</soapenv:Envelope>
```

## Proxy Service Samples

The following Proxy Services samples are available with WSO2 Enterprise Service Bus (ESB) :

- Sample 150: Introduction to Proxy Services
- Sample 151: Custom Sequences and Endpoints with Proxy Services
- Sample 152: Switching Transports and Message Format from SOAP to REST POX
- Sample 153: Routing Messages that Arrive to a Proxy Service without Processing Security Headers
- Sample 154: Load Balancing with Proxy Services
- Sample 155: Dual Channel Invocation on Both Client Side and Server Side of Synapse with Proxy Services
- Sample 156: Service Integration with specifying the receiving sequence
- Sample 157: Conditional Router for Routing Messages based on HTTP URL, HTTP Headers and Query Parameters
- Quality of Service Samples in Service Mediation

## Sample 150: Introduction to Proxy Services

Objective: Introduction to ESB Proxy Services.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy">
 <target>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl" />
 </proxy>
</definitions>
```

### Prerequisites

- Start the Synapse configuration numbered 150: i.e. wso2esb-samples.sh -sn 150
- Start the Axis2 server and deploy the SimpleStockQuoteService if not already done.

Once ESB starts, you could go to <http://localhost:8280/services/StockQuoteProxy?wsdl> and view the WSDL generated for the Proxy Service defined in the configuration. This WSDL is based on the source WSDL supplied in the Proxy Service definition and is updated to reflect the Proxy Service EPR.

Execute the stock quote client by requesting for a stock quote on the Proxy Service as follows:

#### 1. Stock Quote Client

This is a simple SOAP client that could send stock quote requests, and receive and display the last sale price for a stock symbol.

The client is able to operate in the following modes, and send the payloads listed below as SOAP messages:

- quote - sends a quote request for a single stock as follows. The response contains the last sales price for the stock which would be displayed.

```
<m:getQuote xmlns:m="http://services.samples/xsd">
 <m:request>
 <m:symbol>IBM</m:symbol>
 </m:request>
</m:getQuote>
```

- **customquote** - sends a quote request in a custom format. The ESB would transform this custom request into the standard stock quote request format and send it to the service. Upon receipt of the response it would be transformed again to a custom response format and returned to the client, which will then display the last sales price.

```
<m0:checkPriceRequest xmlns:m0="http://services.samples/xsd">
<m0:Code>symbol</m0:Code>
</m0:checkPriceRequest>
```

- **fullquote** - gets quote reports for the stock over a number of days (i.e. last 100 days of the year).

```
<m:getFullQuote xmlns:m="http://services.samples/xsd">
<m:request>
<m:symbol>IBM</m:symbol>
</m:request>
</m:getFullQuote>
```

- **placeorder** - places an order for stocks using a one way request

```
<m:placeOrder xmlns:m="http://services.samples/xsd">
<m:order>
<m:price>3.141593E0</m:price>
<m:quantity>4</m:quantity>
<m:symbol>IBM</m:symbol>
</m:order>
</m:placeOrder>
```

- **marketactivity** - gets a market activity report for the day (i.e. quotes for multiple symbols)

```
<m:getMarketActivity xmlns:m="http://services.samples/xsd">
<m:request>
<m:symbol>IBM</m:symbol>
...
<m:symbol>MSFT</m:symbol>
</m:request>
</m:getMarketActivity>
```

## Note

See `samples/axis2Client/src/samples/common/StockQuoteHandler.java` for sample responses expected by the clients.

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
```

An `inSequence` or `endpoint` or both of these would decide how the message would be handled after the Proxy Service receives the message. In the above example the request received is forwarded to the sample service

hosted on Axis2. The `outSequence` defines how the response is handled before it is sent back to the client. By default, a Proxy Service is exposed over all transports configured for ESB, unless these are specifically mentioned through the `transports` attribute.

### Sample 151: Custom Sequences and Endpoints with Proxy Services

Objective: Using custom sequences and endpoints for message mediation with Proxy Services.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="proxy_1">
 <send>
 <endpoint><address
uri="http://localhost:9000/services/SimpleStockQuoteService"/></endpoint>
 </send>
 </sequence>
 <sequence name="out">
 <send/>
 </sequence>
 <endpoint name="proxy_2_endpoint">
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 <localEntry key="proxy_wsdl"
src="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 <proxy name="StockQuoteProxy1">
 <publishWSDL key="proxy_wsdl"/>
 <target inSequence="proxy_1" outSequence="out" />
 </proxy>
 <proxy name="StockQuoteProxy2">
 <publishWSDL key="proxy_wsdl"/>
 <target endpoint="proxy_2_endpoint" outSequence="out" />
 </proxy>
 </definitions>
```

#### Prerequisites

- Start the Synapse configuration numbered 151: i.e. `wso2esb-samples.sh -sn 151`
- Start the Axis2 server and deploy the `SimpleStockQuoteService` if not already done.

This configuration creates two Proxy Services. The first Proxy Service "StockQuoteProxy1" uses the sequence named "proxy\_1" to process incoming messages and the sequence named "out" to process outgoing responses. The second Proxy Service "StockQuoteProxy2" is set to directly forward messages that are received to the endpoint named "proxy\_2\_endpoint" without any mediation.

You could send a stock quote request to each of these Proxy Services and receive the reply generated by the actual service hosted on the Axis2 instance.

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy1
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy2
```

### Sample 152: Switching Transports and Message Format from SOAP to REST POX

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

## Introduction

This sample demonstrates how a proxy service can be exposed on a subset of available transports and how it could switch from one transport to another.

## Prerequisites

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

## Building the sample

The XML configuration for this sample is as follows:

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="https">
 <target>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"
format="pox"/>
 </endpoint>
 <outSequence>
 <property name="messageType" value="text/xml" scope="axis2"/>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 </proxy>
</definitions>
```

This configuration file `synapse_sample_152.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 152 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds*".

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

## Executing the sample

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

## To execute the sample client

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant stockquote -Dtrpurl=https://localhost:8243/services/StockQuoteProxy
```

## Analyzing the output

This example exposes the created proxy service only on HTTPS. Therefore, if you try to access it over HTTP, it would result in a fault.

```
ant stockquote -Dtrpurl=http://localhost:8280/services/StockQuoteProxy
...
[java] org.apache.axis2.AxisFault: The service cannot be found for the endpoint
reference (EPR) /services/StockQuoteProxy
```

Accessing this over HTTPS using the command `ant stock quote -Dtrpurl=href="https://localhost:8243/services/StockQuoteProxy">https://localhost:8243/services/StockQuoteProxy` causes the proxy service to access the SimpleStockQuoteService on the sample Axis2 server using REST/POX.

If you capture the message exchange using TCPMon, you will see that the REST/POX response is transformed back into a SOAP message and returned to the client as follows:

```
POST http://localhost:9000/services/SimpleStockQuoteService HTTP/1.1
Host: 127.0.0.1
SOAPAction: urn:getQuote
Content-Type: application/xml; charset=UTF-8;action="urn:getQuote";
Transfer-Encoding: chunked
Connection: Keep-Alive
User-Agent: Synapse-HttpComponents-NIO

75
<m0:getQuote xmlns:m0="http://services.samples/xsd">
 <m0:request>
 <m0:symbol>IBM</m0:symbol>
 </m0:request>
</m0:getQuote>
```

```

HTTP/1.1 200 OK
Content-Type: application/xml;
charset=UTF-8;action="http://services.samples/SimpleStockQuoteServicePortType/getQuote
Response";
Date: Tue, 24 Apr 2007 14:42:11 GMT
Server: Synapse-HttpComponents-NIO
Transfer-Encoding: chunked
Connection: Keep-Alive

2b3
<ns:getQuoteResponse xmlns:ns="http://services.samples/xsd">
 <ns:return>
 <ns:change>3.7730036841862384</ns:change>
 <ns:earnings>-9.950236235550818</ns:earnings>
 <ns:high>-80.23868444613285</ns:high>
 <ns:last>80.50750970812187</ns:last>
 <ns:lastTradeTimestamp>Tue Apr 24 20:42:11 LKT 2007</ns:lastTradeTimestamp>
 <ns:low>-79.67368355714606</ns:low>
 <ns:marketCap>4.502043663670823E7</ns:marketCap>
 <ns:name>IBM Company</ns:name>
 <ns:open>-80.02229531286982</ns:open>
 <ns:peRatio>25.089295161182022</ns:peRatio>
 <ns:percentageChange>4.28842665653824</ns:percentageChange>
 <ns:prevClose>87.98107059692451</ns:prevClose>
 <ns:symbol>IBM</ns:symbol>
 <ns:volume>19941</ns:volume>
 </ns:return>
</ns:getQuoteResponse>

```

## Sample 153: Routing Messages that Arrive to a Proxy Service without Processing Security Headers

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### ***Introduction***

This sample demonstrates how you can route messages that arrive to a proxy service without processing the MustUnderstand headers.

In this sample the proxy service will receive a secure message with the MustUnderstand header. Since the element enableSec is not present in the proxy configuration, the ESB will not engage Apache Rampart on this proxy service. It is expected that a MustUnderstand failure exception should occur at the AxisEngine before the message reaches the ESB, but here since the ESB handles this message and gets it in by setting all the headers that are MustUnderstand and not processed to the processed state, this will enable the ESB to route the messages without processing the security headers.

### ***Prerequisites***

- Download and install the unlimited strength policy files for your JDK before using Apache Rampart. To download the policy files, go to <http://www.oracle.com/technetwork/java/javase/downloads/jce-6-download-429243.html>.
- For a list of general prerequisites, see [Prerequisites to Start the ESB Samples](#).

### **Building the sample**

The XML configuration for this sample is as follows:

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy">
 <target>
 <inSequence>
 <property name="preserveProcessedHeaders" value="true" />
 <send>
 <endpoint>
 <address
 uri="http://localhost:9000/services/SecureStockQuoteService" />
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
 uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl" />
 </proxy>
</definitions>
```

This configuration file `synapse_sample_153.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### **To build the sample**

1. Start the ESB with the sample 153 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds*".

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service `SecureStockQuoteService`. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

### **Note**

When you run this sample, the `bouncyCastle.jar` file that is used for encryption does not load into the axis2 client. This is due to an issue with the axis2Client shipped with ESB 4.8.1. Therefore, before running the client, you need to copy the `bcprov-jdk15.jar` file from the `<ESB_HOME>/repository/axis2/client/lib` directory to the `<ESB_HOME>/repository/components/plugins` directory.

### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### **To execute the sample client**

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant stockquote -Dtrpurl=http://localhost:8280/services/StockQuoteProxy
-Dpolicy=../../../../repository/samples/resources/policy/client_policy_3.xml
```

This sends a stock quote request to the proxy service and also signs and encrypts the request by specifying the client side security policy.

#### **Analyzing the output**

By analyzing the debug log output or the TCPMon output, you will see that the request received by the proxy service is signed and encrypted.

You can look up the WSDL of the proxy service by requesting the URL <http://localhost:8280/services/StockQuoteProxy?wsdl> , in order to confirm that the security policy attachments are not available and that security is not engaged.

When sending the message to the backend service, you can verify that the security headers were present as in the original message to the ESB from the client, and that the response received does use WS-Security and forwards the message back to the client without any modification. Since the message inside the ESB is signed and encrypted and can only be forwarded to a secure service, you will see that this is not a security loophole.

#### **Sample 154: Load Balancing with Proxy Services**

Objective: Load Balancing with Proxy Service.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="LBProxy" transports="https http" startOnLoad="true">
 <target faultSequence="errorHandler">
 <inSequence>
 <send>
 <endpoint>
 <session type="simpleClientSession"/>
 <loadbalance
algorithm="org.apache.synapse.endpoints.algorithms.RoundRobin">
 <endpoint>
 <address
uri="http://localhost:9001/services/LBService1">
 <enableAddressing/>

<suspendDurationOnFailure>20</suspendDurationOnFailure>
 </address>
 </endpoint>
 <endpoint>
 <address
uri="http://localhost:9002/services/LBService1">
 <enableAddressing/>

<suspendDurationOnFailure>20</suspendDurationOnFailure>
 </address>
 </endpoint>
 <endpoint>
 <address
uri="http://localhost:9003/services/LBService1">
 <enableAddressing/>

<suspendDurationOnFailure>20</suspendDurationOnFailure>
 </address>
 </endpoint>
 </loadbalance>
 </endpoint>
 </send>
 <drop/>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_2.wsdl"/>
 </proxy>
 <sequence name="errorHandler">
 <makefault>
 <code value="tns:Receiver"
xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason value="COULDN'T SEND THE MESSAGE TO THE SERVER."/>
 </makefault>
 <header name="To" action="remove"/>
 <property name="RESPONSE" value="true"/>
 <send/>
 </sequence>
 </definitions>

```

## Prerequisites

- Start the Synapse configuration numbered 154: `wso2esb-samples.sh -sn 154`
- Start three instances of Axis2 server (on ports 9001, 9002, and 9003), and deploy the `LBSERVICE` to each of them. For more information, see [Setting Up the ESB Samples](#).

Run the client:

```
ant loadbalancefailover -Dmode=session -Dtrpurl=http://localhost:8280/services/LBProxy
```

## Sample 155: Dual Channel Invocation on Both Client Side and Server Side of Synapse with Proxy Services

Objective: Demonstrate the dual channel invocation with Synapse proxy services.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy">
 <target>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService">
 <enableAddressing separateListener="true"/>
 </address>
 </endpoint>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 </proxy>
</definitions>
```

### Prerequisites

- Start the Synapse configuration numbered 155: i.e. `wso2esb-samples.sh -sn 155`
- Start the Axis2 server and deploy the `SimpleStockQuoteService` if not already done.

This sample shows the action of the dual channel invocation within client and Synapse as well as within synapse and the actual server.

### Note

If you want to enable dual channel invocation, you need to set the `separateListener` attribute to true of the `enableAddressing` element of the endpoint.

Execute the stock quote client by requesting for a stock quote on a dual channel from the Proxy Service as follows:

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
-Dmode=dualquote
```

In the above example, the request received is forwarded to the sample service hosted on Axis2 and the endpoint specifies to enable addressing and do the invocation over a dual channel. If you observe this message flow by using

a TCPmon, you could see that on the channel you send the request to synapse the response has been written as an HTTP 202 Accepted, where as the real response from synapse will come over a different channel which cannot be observed unless you use tcpdump to dump all the TCP level messages.

At the same time you can observe the behavior of the invocation between synapse and the actual Axis2 service, where you can see a 202 Accepted message being delivered to synapse as the response to the request. The actual response will be delivered to synapse over a different channel.

### Sample 156: Service Integration with specifying the receiving sequence

**Objective:** Using synapse to integrate services.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <localEntry key="sec_policy"
src="file:repository/conf/sample/resources/policy/policy_3.xml"/>
 <proxy name="StockQuoteProxy">
 <target>
 <inSequence>
 <enrich>
 <source type="body" />
 <target type="property" property="REQUEST" />
 </enrich>

 <send receive="SimpleServiceSeq">
 <endpoint name="secure">
 <address
uri="http://localhost:9000/services/SecureStockQuoteService">
 <enableSec policy="sec_policy" />
 </address>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <drop/>
 </outSequence>
 </target>
 </proxy>
 <sequence name="SimpleServiceSeq">
 <property name="SECURE_SER_AMT"
expression="//ns:getQuoteResponse/ns:return/ax21:last"
 xmlns:ns="http://services.samples"
 xmlns:ax21="http://services.samples/xsd"/>
 <log level="custom">
 <property name="SecureStockQuoteService-Amount"
expression="get-property('SECURE_SER_AMT')"/>
 </log>
 <enrich>
 <source type="body" />
 <target type="property" property="SecureService_Res" />
 </enrich>
 <enrich>
 <source type="property" property="REQUEST" />
 <target type="body" />
 </enrich>
 <header name="Action" scope="default" value="urn:getQuote" />
 <send receive="ClientOutSeq">
 <endpoint name="SimpleStockQuoteService">
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </sequence>

```

```
</send>
</sequence>
<sequence name="ClientOutSeq">
 <property name="SIMPLE_SER_AMT"
expression="//ns:getQuoteResponse/ns:return/ax21:last"
 xmlns:ns="http://services.samples"
 xmlns:ax21="http://services.samples/xsd"/>
 <log level="custom">
 <property name="SimpleStockQuoteService-Amount"
expression="get-property('SIMPLE_SER_AMT')"/>
 </log>
 <enrich>
 <source type="body"/>
 <target type="property" property="SimpleService_Res"/>
 </enrich>
 <filter xpath="fn:number(get-property('SIMPLE_SER_AMT')) >
fn:number(get-property('SECURE_SER_AMT'))">
 <then>
 <log>
 <property name="StockQuote" value="SecureStockQuoteService"/>
 </log>
 <enrich>
 <source type="property" property="SecureService_Res"/>
 <target type="body"/>
 </enrich>
 </then>
 <else>
 <log>
 <property name="StockQuote" value="SimpleStockQuoteService"/>
 </log>
 </else>
 </filter>
```

```

 <send/>
 </sequence>
</definitions>

```

**Prerequisites:**

We will be using two services in this sample; i.e. SimpleStockQuoteService and SecureStockQuoteService . Therefore the prerequisites of sample 100 is also applied here.

- Start the Synapse configuration numbered 156: i.e. wso2esb-samples.sh -sn 156
- Start the Axis2 server and deploy the SimpleStockQuoteService if not already done.

This sample contains a proxy service which provides the seamless integration of SimpleStockQuoteService and SecureStockQuoteService. Once a client send a request to this proxy service, it sends the same request to both these services and resolve the service with cheapest stock quote price.

Execute the stock quote client by requesting for a cheapest stock quote from the proxy service as follows:

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
```

Above sample uses the concept of specifying the receiving sequence in the send mediator. In this case once the message is sent from the in sequence then the response won't come to out sequence as the receiving sequence is specified in the send mediator.

### Sample 157: Conditional Router for Routing Messages based on HTTP URL, HTTP Headers and Query Parameters

**Objective:** Routing Messages based on the HTTP Transport properties using the [Conditional Router mediator](#).

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="https http" startOnLoad="true"
trace="disable">
 <target>
 <inSequence>
 <conditionalRouter continueAfter="false">
 <conditionalRoute breakRoute="false">
 <condition>
 <match xmlns="" type="header" source="foo" regex="bar.*"/>
 </condition>
 <target sequence="cndl_seq" />
 </conditionalRoute>
 <conditionalRoute breakRoute="false">
 <condition>
 <and xmlns="">
 <match type="header" source="my_custom_header1"
regex="foo.*"/>
 <match type="url"
regex="/services/StockQuoteProxy.*"/>
 </and>
 </condition>
 <target sequence="cnd2_seq" />
 </conditionalRoute>
 <conditionalRoute breakRoute="false">
 <condition>
 <and xmlns="">
 <match type="header" source="my_custom_header2"
regex="bar.*"/>

```

```

 <equal type="param" source="qparam1" value="qpv_foo"/>
 <or>
 <match type="url"
regex="/services/StockQuoteProxy.*"/>
 <match type="header" source="my_custom_header3"
regex="foo.*"/>
 </or>
 <not>
 <equal type="param" source="qparam2"
value="qpv_bar"/>
 </not>
 </and>
</condition>
<target sequence="cnd3_seq" />
</conditionalRoute>
</conditionalRouter>
</inSequence>
<outSequence>
 <send/>
</outSequence>
</target>
</proxy>
<sequence name="cnd1_seq">
 <log level="custom">
 <property name="MSG_FLOW" value="Condition (I) Satisfied"/>
 </log>
 <sequence key="send_seq" />
</sequence>
<sequence name="cnd2_seq">
 <log level="custom">
 <property name="MSG_FLOW" value="Condition (II) Satisfied"/>
 </log>
 <sequence key="send_seq" />
</sequence>
<sequence name="cnd3_seq">
 <log level="custom">
 <property name="MSG_FLOW" value="Condition (III) Satisfied"/>
 </log>
 <sequence key="send_seq" />
</sequence>
<sequence name="send_seq">
 <log level="custom">
 <property name="DEBUG" value="Condition Satisfied"/>
 </log>
 <send>
 <endpoint name="simple">
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>

```

```

 </send>
 </sequence>
</definitions>

```

### Prerequisites:

- Start the Synapse configuration numbered 157: i.e. wso2esb-samples.sh -sn 157
- Start the Axis2 server and deploy the SimpleStockQuoteService if not already done. For this particular case we will be using 'curl' to send requests with custom HTTP Headers to the proxy service. You may use a similar tool with facilitate those requirements.

The request file stockQuoteReq.xml should contain the following request.

```

<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
 xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
 <soap:Header/>
 <soap:Body>
 <ser:getQuote>
 <ser:request>
 <xsd:symbol>IBM</xsd:symbol>
 </ser:request>
 </ser:getQuote>
 </soap:Body>
</soap:Envelope>

```

### Condition I : Matching HTTP Header

```
curl -d @stockQuoteReq.xml -H "Content-Type: application/soap+xml; charset=UTF-8" -H "foo:bar" "http://localhost:8280/services/StockQuoteProxy"
```

### Condition II : Matching HTTP Header AND Url

```
curl -d @stockQuoteReq.xml -H "Content-Type: application/soap+xml; charset=UTF-8" -H "my_custom_header1:foo1" "http://localhost:8280/services/StockQuoteProxy"
```

### Condition III :

Complex conditions with AND, OR and NOT

```
curl -d @stockQuoteReq.xml -H "Content-Type: application/soap+xml; charset=UTF-8" -H "my_custom_header2:bar" -H "my_custom_header3:foo" "http://localhost:8280/services/StockQuoteProxy?qparam1=qpv_foo&qparam2=qpv_foo2"
```

## Quality of Service Samples in Service Mediation

The following QoS sample is available for proxy services with WSO2 Enterprise Service Bus (ESB) :

- Sample 200: Using WS-Security with policy attachments for proxy services

### Sample 200: Using WS-Security with policy attachments for proxy services

- Introduction

- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

## Introduction

This sample demonstrates how you can use WS-Security signing and encryption with proxy services through WS-Policy.

In this sample the proxy service expects to receive a signed and encrypted message as specified by the security policy. To understand the format of the policy file, have a look at the Apache Rampart and Axis2 documentation. The element `engageSec` specifies that Apache Rampart should be engaged on this proxy service. Hence if Rampart rejects any request message that does not conform to the specified policy, that message will never reach the `inSequence` in order to be processed. Since the proxy service is forwarding the received request to the simple stock quote service that does not use WS-Security, you are instructing the ESB to remove the `wsse:Security` header from the outgoing message.

## Prerequisites

- For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).
- Download and install the unlimited strength policy files for your JDK before using Apache Rampart. To download the policy files, go to <http://www.oracle.com/technetwork/java/javase/downloads/jce-6-download-429243.html>.

## Building the sample

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <localEntry key="sec_policy"
src="file:repository/samples/resources/policy/policy_3.xml"/>
 <proxy name="StockQuoteProxy">
 <target>
 <inSequence>
 <header name="wsse:Security" action="remove"
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"/>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 <policy key="sec_policy"/>
 <enableSec/>
 </proxy>
</definitions>

```

This configuration file `synapse_sample_200.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### To build the sample

1. Start the ESB with the sample 200 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service `SimpleStockQuoteService`. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

### Note

When you run this sample, the `bouncyCastle.jar` file that is used for encryption does not load into the axis2 client. This is due to an issue with the axis2Client shipped with ESB 4.8.1. Therefore, before running the client, you need to copy the `bcprov-jdk15.jar` file from the `<ESB_HOME>/repository/axis2/client/lib` directory to the `<ESB_HOME>/repository/components/plugins` directory.

### Executing the sample

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

#### To execute the sample client

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant stockquote -Dtrpurl=http://localhost:8280/services/StockQuoteProxy
-Dpolicy=../../../../repository/samples/resources/policy/client_policy_3.xml
```

This sends a stock quote request to the proxy service and also signs and encrypts the request by specifying the client side security policy.

### Analyzing the output

By analyzing the debug log output or the TCPMon output, you will see that the request received by the proxy service is signed and encrypted.

You can look up the WSDL of the proxy service by requesting the URL `http://localhost:8280/services/StockQuoteProxy?wsdl`, in order to confirm the security policy attachment to the supplied base WSDL.

When sending the message to the backend service, you can verify that the security headers were removed and that the response received does not use WS-Security, but that the response being forwarded back to the client is signed and encrypted as expected by the client.

### Transports Samples and Switching Transports

The following transport samples are available with WSO2 Enterprise Service Bus (ESB) :

- Sample 250: Introduction to Switching Transports

- Sample 251: Switching from HTTP(S) to JMS
- Sample 252: Pure Text (Binary) and POX Message Support with JMS
- Sample 253: Bridging from JMS to HTTP and Replying with a 202 Accepted Response
- Sample 254: Using the File System as Transport Medium (VFS)
- Sample 255: Switching from FTP Transport Listener to Mail Transport Sender
- Sample 256: Proxy Services with the MailTo Transport
- Sample 257: Proxy Services with the FIX Transport
- Sample 258: Switching from HTTP to FIX
- Sample 259: Switch from FIX to HTTP
- Sample 260: Switch from FIX to AMQP
- Sample 261: Switching between FIX Versions
- Sample 262: CBR of FIX Messages
- Sample 263: Transport Switching - JMS to http/s Using JBoss Messaging(JBM)
- Sample 264: Sending Two-Way Messages Using JMS transport
- Sample 265: Accessing a Windows Share Using the VFS Transport
- Sample 266: Switching from TCP to HTTP/S
- Sample 267: Switching from UDP to HTTP/S
- Sample 268: Proxy Services with the Local Transport
- Sample 270: Transport switching from HTTP to MSMQ and MSMQ to HTTP
- Sample 271: File Processing
- Sample 272: Publishing and Subscribing using WSO2 ESB's MQ Telemetry Transport

## Sample 250: Introduction to Switching Transports

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

### ***Introduction***

This sample demonstrates how the ESB receives a messages over the JMS transport and forwards it over a HTTP/S transport.. In this sample, the client sends a request message to the proxy service exposed in JMS. The ESB forwards this message to the HTTP EPR of the simple stock quote service hosted on the sample Axis2 server, and returns the reply back to the client through a JMS temporary queue.

### ***Prerequisites***

- Configure WSO2 ESB's JMS transport with ActiveMQ and enable the JMS transport listener and sender. For information on how to configure with ActiveMQ, and how to enable the JMS transport listener and sender, see [Configure with ActiveMQ](#).

### **Note**

If you are using ActiveMQ version 5.8.0 or above, you need to copy the `hawtbuf-1.2.jar` file from the <ActiveMQ>/lib directory to the <ESB\_HOME>/repository/components/lib directory.

- For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="jms">
 <target>
 <inSequence>
 <property action="set" name="OUT_ONLY" value="true" />
 </inSequence>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 <parameter name="transport.jms.ContentType">
 <rules>
 <jmsProperty>contentType</jmsProperty>
 <default>application/xml</default>
 </rules>
 </parameter>
 </proxy>
</definitions>

```

This configuration file `synapse_sample_250.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### To build the sample

1. Start the ESB with the sample 250 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

### Executing the sample

#### To execute the sample JMS client

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory:

```
ant jmsclient -Djms_type=pox -Djms_dest=dynamicQueues/StockQuoteProxy
-Djms_payload=MSFT
```

### Analyzing the output

When you run the client and analyze the ESB debug log, you will see the following entry, which shows that the JMS listener received the request message:

When you run the client, it sends a plain XML formatted place order request to a JMS queue named StockQuoteProxy. The ESB polls on this queue for any incoming messages and picks up the request. If you run the ESB in the DEBUG mode, you will see the following entry on the console.

```
[JMSWorker-1] DEBUG ProxyServiceMessageReceiver -Proxy Service StockQuoteProxy
received a new message...
```

Then, the ESB mediated the request and forwards it to the sample Axis2 server over HTTP. If you analyze the console running the sample Axis2 server, you will see the following message indicating that the server has accepted an order:

```
Accepted order for : 16517 stocks of MSFT at $ 169.14622538721846
```

## Note

It is possible to instruct a JMS proxy service to listen to an already existing destination without creating a new one. To do this, use the property elements on the proxy service definition to specify the destination and connection factory.

```
<property name="transport.jms.Destination"
value="dynamicTopics/something.TestTopic"/>
```

### Sample 251: Switching from HTTP(S) to JMS

Objective: Demonstrate switching from HTTP to JMS.

#### Prerequisites

- Download, install and start a JMS server.
- Configure sample Axis2 server for JMS.
- Start the Axis2 server and deploy the SimpleStockQuoteService.
- Configure the Synapse JMS transport.
- Start the Synapse configuration numbered 251: wso2esb-samples.sh -sn 251

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="http">
 <target>
 <endpoint>
 <address
uri="jms:/SimpleStockQuoteService?transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory&
java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory&
;java.naming.provider.url=tcp://localhost:61616"/>
 </endpoint>
 <inSequence>
 <property action="set" name="OUT_ONLY" value="true" />
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl" />
 </proxy>
</definitions>

```

To switch from HTTP to JMS, edit the samples/axis2Server/repository/conf/axis2.xml for the sample Axis2 server and enable JMS, and restart the server. Now you can see that the simple stock quote service is available in both JMS and HTTP in the sample Axis2 server. To see this, point your browser to the WSDL of the service at <http://localhost:9000/services/SimpleStockQuoteService?wsdl>. JMS URL for the service is mentioned as below:

```
jms:/SimpleStockQuoteService?transport.jms.ConnectionFactoryJNDIName=
QueueConnectionFactory&java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory&
java.naming.provider.url=tcp://localhost:61616
```

You may also notice that the simple stock quote Proxy Service exposed in ESB is now available only in HTTP as we have specified transport for that service as HTTP. To observe this, access the WSDL of stock quote proxy service at <http://localhost:8280/services/StockQuoteProxy?wsdl>.

This ESB configuration creates a Proxy Service Samples over HTTP and forwards received messages to the above EPR using JMS, and sends back the response to the client over HTTP once the simple stock quote service responds with the stock quote reply over JMS to the ESB server. To test this, send a place order request to ESB using HTTP as follows:

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
-Dmode=placeorder -Dsymbol=MSFT
```

The sample Axis2 server console will print a message indicating that it has accepted the order as follows:

```
Accepted order for : 18406 stocks of MSFT at $ 83.58806051152119
```

## Sample 252: Pure Text (Binary) and POX Message Support with JMS

Objective: Pure POX/Text and Binary JMS Proxy services, including MTOM.

#### Prerequisites

- Configure JMS for ESB.
- Start the Synapse configuration numbered 252: `wso2esb-samples.sh -sn 252`
- Start the Axis2 server and deploy the `SimpleStockQuoteService` and the `MTOMSwASampleService` if not already done.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="text_proxy">
 <header name="Action" value="urn:placeOrder"/>
 <script language="js"><![CDATA[
 var args = mc.getPayloadXML().toString().split(" ");
 mc.setPayloadXML(
 <m:placeOrder xmlns:m="http://services.samples/xsd">
 <m:order>
 <m:price>{args[0]}</m:price>
 <m:quantity>{args[1]}</m:quantity>
 <m:symbol>{args[2]}</m:symbol>
 </m:order>
 </m:placeOrder>);
]]></script>
 <property action="set" name="OUT_ONLY" value="true"/>
 <property name="messageType" value="text/xml" scope="axis2"/>
 <send>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"
format="pox"/>
 </endpoint>
 </send>
 </sequence>
 <sequence name="mtom_proxy">
 <property action="set" name="OUT_ONLY" value="true"/>
 <header name="Action" value="urn:oneWayUploadUsingMTOM"/>
 <send>
 <endpoint>
 <address uri="http://localhost:9000/services/MTOMSwASampleService"
optimize="mtom"/>
 </endpoint>
 </send>
 </sequence>
 <sequence name="pox_proxy">
 <property action="set" name="OUT_ONLY" value="true"/>
 <header name="Action" value="urn:placeOrder"/>
 <send>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"
format="soap11"/>
 </endpoint>
 </send>
 </sequence>
 <sequence name="out">
 <send/>
 </sequence>
 <proxy name="JMSFileUploadProxy" transports="jms">
 <target inSequence="mtom_proxy" outSequence="out"/>
 <parameter
name="transport.jmsWrapper">{http://services.samples/xsd}element</parameter>
 </proxy>

```

```
<proxy name="JMSTextProxy" transports="jms">
 <target inSequence="text_proxy" outSequence="out" />
 <parameter
name="transport.jms.Wrapper">{http://services.samples/xsd}text</parameter>
</proxy>
<proxy name="JMSPoxProxy" transports="jms">
 <target inSequence="pox_proxy" outSequence="out" />
 <parameter name="transport.jms.ContentType">application/xml</parameter>
```

```
</proxy>
</definitions>
```

This configuration creates three JMS Proxy Services named `JMSFileUploadProxy`, `JMSTextProxy` and `JMSPOXProxy` exposed over JMS queues with the same names as the services. The first part of this example demonstrates the pure text message support with JMS, where a user sends a space separated text JMS message of the form "`<price> <qty> <symbol>`". ESB converts this message into a SOAP message and sends this to the `SimpleStockQuoteServices`' `placeOrder` operation. ESB uses the Script Mediator to transform the text message into a XML payload using the Javascript support available to tokenize the string. The Proxy Service property named `transport.jms.Wrapper` defines a custom wrapper element `QName`, to be used when wrapping text/binary content into a SOAP envelope.

Execute JMS client as follows. This will post a pure text JMS message with the content defined (for example, "12.33 1000 ACP") to the specified JMS destination - `dynamicQueues/JMSTextProxy`.

```
ant jmsclient -Djms_type=text -Djms_payload="12.33 1000 ACP"
-Djms_dest=dynamicQueues/JMSTextProxy
```

Following the debug logs, you could notice that ESB received the JMS text message and transformed it into a SOAP payload as follows.

## Note

The wrapper element "[<http://services.samples/xsd>]text" has been used to hold the text message content.

```
INFO - To: , WSAction: urn:mediate, SOAPAction: urn:mediate, MessageID:
ID:orxus.vedehen.org-50631-1225235276233-1:0:1:1:1, Direction: request,
Envelope:
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
 <soapenv:Body>
 <axis2ns1:text xmlns:axis2ns1="http://services.samples/xsd">12.33 1000
 ACP</axis2ns1:text>
 </soapenv:Body>
</soapenv:Envelope>
```

Now, you could see how the Script Mediator created a stock quote request by tokenizing the text as follows, and sent the message to the `placeOrder` operation of the `SimpleStockQuoteService`.

```

INFO - To: , WSAction: urn:placeOrder, SOAPAction: urn:placeOrder, MessageID:
ID:orxus.vedehen.org-50631-1225235276233-1:0:1:1:1, Direction: request,
Envelope:
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body>
<placeOrder xmlns="http://services.samples">
<order xmlns="http://services.samples/xsd">
<price>12.33</price>
<quantity>1000</quantity>
<symbol>ACP</symbol>
</order>
</placeOrder>
</soapenv:Body>
</soapenv:Envelope>

```

The sample Axis2 server would now accept the one way message and issue the following message:

```

Wed Apr 25 19:50:56 LKT 2007 samples.services.SimpleStockQuoteService :: Accepted
order for : 1000 stocks of ACP at $ 12.33

```

The next section of this example demonstrates how a pure binary JMS message could be received and processed through ESB. The configuration creates a Proxy Service Sample named `JMSFileUploadProxy` that accepts binary messages and wraps them into a custom element "`(http://services.samples/xsd)element`." The received message is then forwarded to the `MTOMSwASampleService` using the SOAP action `urn:oneWayUploadUsingMTOM` and optimizing binary content using MTOM. To execute this sample, use the JMS client to publish a pure binary JMS message containing the file `../../../../repository/samples/resources/mtom/asf-logo.gif` to the JMS destination `dynamicQueues/JMSFileUploadProxy` as follows:

```

ant jmsclient -Djms_type=binary -Djms_dest=dynamicQueues/JMSFileUploadProxy
-Djms_payload=../../../../repository/samples/resources/mtom/asf-logo.gif

```

Examining the ESB debug logs reveals that the binary content was received over JMS and wrapped with the specified element into a SOAP infoset as follows:

```

INFO - To: , WSAction: urn:mediate, SOAPAction: urn:mediate, MessageID:
ID:orxus.vedehen.org-50702-1225236039556-1:0:1:1:1, Direction: request,
Envelope:
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body>
<axis2ns3:element
xmlns:axis2ns3="http://services.samples/xsd">R0lGODlhgw...AAOw==</axis2ns3:element>
</soapenv:Body>
</soapenv:Envelope>

```

Thereafter the message was sent as a MTOM optimized message as specified by the `format=mtom` attribute of the endpoint, to the `MTOMSwASampleService` using the SOAP action `urn:oneWayUploadUsingMTOM`. Once received by the sample service, it is saved into a temporary file and could be verified for correctness.

```
Wrote to file : ../../work/temp/sampleServer/mtom-29208.gif
```

The final section of this example shows how a POX JMS message received by ESB is sent to the `SimpleStockQuoteService` as a SOAP message. Use the JMS client as follows to create a POX (Plain Old XML) message with a stock quote request payload (without a SOAP envelope), and send it to the JMS destination `dynamicQueues/JMSPoxProxy` as follows:

```
ant jmsclient -Djms_type=pox -Djms_dest=dynamicQueues/JMSPoxProxy -Djms_payload=MSFT
```

ESB converts the POX message into a SOAP payload and sends to the `SimpleStockQuoteService` after setting the SOAP action as `urn:placeOrder`. The sample Axis2 server displays a successful message on the receipt of the message as:

```
Wed Apr 25 20:24:50 LKT 2007 samples.services.SimpleStockQuoteService :: Accepted
order for : 19211 stocks of MSFT at $ 172.39703010684752
```

### Sample 253: Bridging from JMS to HTTP and Replying with a 202 Accepted Response

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

#### ***Introduction***

This sample demonstrates how the ESB performs transport switching between JMS and HTTP, and also demonstrates how to configure a one-way HTTP proxy.

#### ***Prerequisites***

- Configure WSO2 ESB's JMS transport with ActiveMQ and enable the JMS transport listener. For information on how to configure WSO2 ESB's JMS transport with ActiveMQ, see [Configure with ActiveMQ](#).

#### **Note**

If you are using ActiveMQ version 5.8.0 or above, you need to copy the `hawtbuf-1.2.jar` file from the <ActiveMQ>/lib directory to the <ESB\_HOME>/repository/components/lib directory.

- For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

#### ***Building the sample***

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="JMSToHTTPStockQuoteProxy" transports="jms">
 <target>
 <inSequence>
 <property action="set" name="OUT_ONLY" value="true" />
 </inSequence>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 </proxy>
 <proxy name="OneWayProxy" transports="http">
 <target>
 <inSequence>
 <log level="full"/>
 </inSequence>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 </proxy>
</definitions>

```

This configuration file `synapse_sample_253.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 253 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

## Executing the sample

This example invokes the one-way `placeOrder` operation on the `SimpleStockQuoteService` using the Axis2 `ServiceClient.fireAndForget()` API at the client. To test this, run the the following command from the `<ESB_HOME>/samples/axis2Client` directory:

```
ant stockquote -Dmode=placeorder
-Dtrpurl="jms:/JMSToHTTPStockQuoteProxy?transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory&java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory&java.naming.provider.url=tcp://localhost:61616&transport.jms.ContentTypeProperty=Content-Type&transport.jms.DestinationType=queue"
```

To test how the ESB responds with a *HTTP 202 Accepted* response to a request received, run the the following command from the <ESB\_HOME>/samples/axis2Client directory:

```
ant stockquote -Dmode=placeorder -Dtrpurl=http://localhost:8280/services/OneWayProxy
```

#### **Analyzing the output**

When you run the first command specified above, you will see the one-way JMS message flowing through the ESB into the sample Axis2 server instance over HTTP, and you will also see how Axis2 acknowledges it with a HTTP 202 Accepted response.

```
SimpleStockQuoteService :: Accepted order for : 7482 stocks of IBM at $
169.27205579038733
```

When you run the next command, you will see that the proxy service simply logs the message received and acknowledges it. On the ESB console you will see the logged message, and if TCPMon is used at the client, you will also see the 202 Accepted response that is sent back to the client from the ESB.

```
HTTP/1.1 202 Accepted
Content-Type: text/xml; charset=UTF-8
Host: 127.0.0.1
SOAPAction: "urn:placeOrder"
Date: Sun, 06 May 2007 17:20:19 GMT
Server: Synapse-HttpComponents-NIO
Transfer-Encoding: chunked

0
```

## **Sample 254: Using the File System as Transport Medium (VFS)**

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

#### ***Introduction***

The ESB can access the local file system using the [VFS transport](#) sender and receiver. This sample demonstrates the VFS transport in action, using the file system as a transport medium.

#### ***Prerequisites***

- Enable the VFS transport. For details, see [Enable VFS](#).
- Create 3 new directories (folders) named **in**, **out** and **original** in a suitable location in a test directory (e.g.,

/home/user/test) in the local file system. Then, open the repository/sample/synapse\_sample\_254.xml file in a text editor and change the transport.vfs.FileURI, transport.vfs.MoveAfterProcess, transport.vfs.MoveAfterFailure parameter values to the **in**, **original** and **original** directory locations respectively. You need to set both transport.vfs.MoveAfterProcess and transport.vfs.MoveAfterFailure parameter values to point to the **original** directory location. Change the endpoint in the <outSequence> to point to the **out** directory location. Make sure that the prefix vfs: in the endpoint URL is not removed or changed.

- For a list of general prerequisites, see [Prerequisites to Start the ESB Samples](#).

### **Building the sample**

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="vfs">
 <parameter name="transport.vfs.FileURI">file:///home/user/test/in</parameter>
 <!--CHANGE-->
 <parameter name="transport.vfs.ContentType">text/xml</parameter>
 <parameter name="transport.vfs.FileNamePattern">.*\.xml</parameter>
 <parameter name="transport.PollInterval">15</parameter>
 <parameter
name="transport.vfs.MoveAfterProcess">file:///home/user/test/original</parameter>
 <!--CHANGE-->
 <parameter
name="transport.vfs.MoveAfterFailure">file:///home/user/test/original</parameter>
 <!--CHANGE-->
 <parameter name="transport.vfs.ActionAfterProcess">MOVE</parameter>
 <parameter name="transport.vfs.ActionAfterFailure">MOVE</parameter>
 <target>
 <endpoint>
 <address format="soap12"
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 <outSequence>
 <property name="transport.vfs.ReplyFileName"

expression="fn:concat(fn:substring-after(get-property('MessageID'), 'urn:uuid:'),
'.xml')"
 scope="transport" />
 <property action="set" name="OUT_ONLY" value="true" />
 <send>
 <endpoint>
 <address uri="vfs:file:///home/user/test/out" />
 <!--CHANGE-->
 </endpoint>
 </send>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl" />
 </proxy>
</definitions>
```

This configuration file synapse\_sample\_254.xml is available in the <ESB\_HOME>/repository/samples directory and the values you have to change as specified in the prerequisites section are marked with <!--CHANGE-->.

### **To build the sample**

1. Start the ESB with the sample 254 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).  
The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "WSO2 Carbon started in n seconds."
2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

#### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

#### **To execute the sample client**

Move the `test.xml` file from the `<ESB_HOME>/repository/samples/resources/vfs` directory to the location specified in `transport.vfs.FileURI` in the configuration (i.e., the `in` directory).

The `test.xml` file contains a simple stock quote request in XML/SOAP format and is as follows:

#### **test.xml**

```
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wsa="http://www.w3.org/2005/08/addressing">
 <soapenv:Body>
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>IBM</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </soapenv:Body>
</soapenv:Envelope>
```

#### **Analyzing the output**

You will see that the VFS transport listener picks the file from the `in` directory and sends it to the Axis2 service over HTTP. Then you will see that the request XML file is moved to the `original` directory and that the response from the Axis2 server is saved to the `out` directory.

#### **Sample 255: Switching from FTP Transport Listener to Mail Transport Sender**

**Objective:** Switching from FTP transport listener to mail transport sender.

#### **Prerequisites**

- You will need access to an FTP server and an SMTP server to try this sample.
- Start the Axis2 server and deploy the `SimpleStockQuoteService` if not already done.
- Enable mail transport sender in the ESB `axis2.xml`.
- Create a new test directory in the FTP server. Open `ESB_HOME/repository/samples/synapse_sample_255.xml` and edit the following values. Change `transport.vfs.FileURI` parameter value point to the test directory at the FTP server. Change `outSequence` endpoint address URI email address to a working email address. Values you have to change are marked with "`<!--CHANGE-->`".
- Start the Synapse configuration numbered 255: `wso2esb-samples.sh -sn 255`
- Copy `ESB_HOME/repository/samples/resources/vfs/test.xml` to the `ftp` directory given in `transport.vfs.FileURI` below.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="vfs">
 <parameter
name="transport.vfs.FileURI">vfs:ftp://guest:guest@localhost/test?vfs.passive=true</pa
rameter> <!--CHANGE-->
 <parameter name="transport.vfs.ContentType">text/xml</parameter>
 <parameter name="transport.vfs.FileNamePattern">.**.xml</parameter>
 <parameter name="transport.PollInterval">15</parameter>
 <target>
 <inSequence>
 <header name="Action" value="urn:getQuote"/>
 </inSequence>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 <outSequence>
 <property action="set" name="OUT_ONLY" value="true"/>
 <send>
 <endpoint>
 <address uri="mailto:user@host"/> <!--CHANGE-->
 </endpoint>
 </send>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 </proxy>
</definitions>

```

VFS transport listener will pick the file from the directory in the FTP server and send it to the Axis2 service. The file in the FTP directory will be deleted. The response will be sent to the given e-mail address.

#### **Setting up Mail Transport Sender**

To enable the mail transport sender for samples, you need to uncomment the mail transport sender configuration in the repository/conf/axis2/axis2.xml. Uncomment the mail transport sender sample configuration and make sure it points to a valid SMTP configuration for any actual scenarios.

```

<transportSender name="mailto"
class="org.apache.synapse.transport.mail.MailTransportSender">
 <parameter name="mail.smtp.host">smtp.gmail.com</parameter>
 <parameter name="mail.smtp.port">587</parameter>
 <parameter name="mail.smtp.starttls.enable">true</parameter>
 <parameter name="mail.smtp.auth">true</parameter>
 <parameter name="mail.smtp.user">synapse.demo.0</parameter>
 <parameter name="mail.smtp.password">mailpassword</parameter>
 <parameter name="mail.smtp.from">synapse.demo.0@gmail.com</parameter>
</transportSender>

```

#### **Sample 256: Proxy Services with the MailTo Transport**

Objective: Using the [MailTo transport](#) with Proxy Services.

##### **Prerequisites**

- You will need access to an e-mail account.

- Start the Axis2 server and deploy the `SimpleStockQuoteService` if not already done.
- Enable the mail transport listener in the ESB `axis2.xml`. Simply uncomment the relevant transport receiver entry in the file.
- Enable mail transport sender in the ESB `axis2.xml`. See [MailTo transport](#) for details.
- Start the Synapse configuration numbered 256: `wso2esb-samples.sh -sn 256`.
- Send a plain/text e-mail (Make sure you switch to **Plain text mode** when you are composing the email) with the following body and any custom Subject from your mail account to the mail address `synapse.demo.1@gmail.com`.

```
<m0:getQuote xmlns:m0="http://services.samples">
<m0:request>
<m0:symbol>IBM</m0:symbol>
</m0:request>
</m0:getQuote>
```

- After a few seconds (for example 30 seconds), you should receive a POX response in your e-mail account with the stock quote reply.

```

<!-- Using the mail transport -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="mailto">
 <parameter name="transport.mail.Address">synapse.demo.1@gmail.com</parameter>
 <parameter name="transport.mail.Protocol">pop3</parameter>
 <parameter name="transport.PollInterval">5</parameter>
 <parameter name="mail.pop3.host">pop.gmail.com</parameter>
 <parameter name="mail.pop3.port">995</parameter>
 <parameter name="mail.pop3.user">synapse.demo.1</parameter>
 <parameter name="mail.pop3.password">mailpassword1</parameter>
 <parameter name="mail.pop3.socketFactory.class">javax.net.ssl.SSLSocketFactory</parameter>
 <parameter name="mail.pop3.socketFactory.fallback">false</parameter>
 <parameter name="mail.pop3.socketFactory.port">995</parameter>
 <parameter name="transport.mail.ContentType">application/xml</parameter>
 <target>
 <inSequence>
 <property name="senderAddress" expression="get-property('transport',
'From')"/>
 <log level="full">
 <property name="Sender Address"
expression="get-property('senderAddress')"/>
 </log>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <property name="Subject" value="Custom Subject for Response"
scope="transport"/>
 <header name="To" expression="fn:concat('mailto:',
get-property('senderAddress'))"/>
 <log level="full">
 <property name="message" value="Response message"/>
 <property name="Sender Address"
expression="get-property('senderAddress')"/>
 </log>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 </proxy>
</definitions>

```

## Note

In this sample, we used the `transport.mail.ContentType` property to make sure that the transport parses the request message as POX. If you remove this property, you may still be able to send requests using a standard mail client if instead of writing the XML in the body of the message, you add it as an attachment. In that case, you should use XML as a suffix for the attachment and format the request as a SOAP 1.1 message. Indeed, for a file with suffix XML the mail client will most likely use text/XML as the

content type, exactly as required for SOAP 1.1. Sending a POX message using this approach will be a lot trickier, because most standard mail clients do not allow the user to explicitly set the content type.

## Sample 257: Proxy Services with the FIX Transport

**Objective:** Demonstrate the usage of the FIX (Financial Information eXchange) transport with Proxy Services.  
**Prerequisites**

- You will need the two sample FIX applications that come with Quickfix/J (Banzai and Executor). Configure the two applications to establish sessions with the ESB.
- Start Banzai and Executor.
- Enable FIX transport in the Synapse axis2.xml.
- Configure Synapse for FIX samples.
- Open the <ESB\_HOME>/repository/samples/synapse\_sample\_257.xml file and make sure that `transport.fix.AcceptorConfigURL` property points to the `fix-synapse.cfg` file you created. Also make sure that `transport.fix.InitiatorConfigURL` property points to the `synapse-sender.cfg` file you created. Once done, you can start the Synapse configuration numbered 257: `wso2esb-samples.sh -sn 257`

### Note

The ESB creates a new FIX session with Banzai at this point.

- Send an order request from Banzai to the ESB.

WSO2 ESB will create a session with an Executor and forward the order request. The responses coming from the Executor will be sent back to Banzai.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="FIXProxy" transports="fix">
 <parameter
name="transport.fix.AcceptorConfigURL">file:/home/synapse_user/fix-config/fix-synapse.
cfg</parameter>
 <parameter
name="transport.fix.InitiatorConfigURL">file:/home/synapse_user/fix-config/synapse-sen
der.cfg</parameter>
 <parameter name="transport.fix.AcceptorMessageStore">file</parameter>
 <parameter name="transport.fix.InitiatorMessageStore">file</parameter>
 <target>
 <endpoint>
 <address
uri="fix://localhost:19876?BeginString=FIX.4.0&SenderCompID=SYNAPSE&TargetComp
ID=EXEC"/>
 </endpoint>
 <inSequence>
 <log level="full"/>
 </inSequence>
 <outSequence>
 <log level="full"/>
 </outSequence>
 <send/>
 </target>
 </proxy>
 </definitions>
```

### **Configuring Sample FIX Applications**

If you use a binary distribution of Quickfix/J, the two samples and their configuration files are all packed to a single JAR file called `quickfixj-examples.jar`. You will have to extract the JAR file, modify the configuration files and pack them to a JAR file again under the same name.

You can pass the new configuration file as a command line parameter too, in that case you do not need to modify the `quickfixj-examples.jar`. You can copy the config files from `<ESB_HOME>/repository/samples/resources/fix folder` to `$QFJ_HOME/etc` folder. Execute the sample apps from `<QFJ_HOME>/bin, " ./banzai.sh/bat .. /etc/banbai.cfg executor.sh/bat .. /etc/executor.cfg.`

Locate and edit the FIX configuration file of Executor to be as follows. This file is usually named `executor.cfg`.

```
[default]
FileStorePath=examples/target/data/executor
ConnectionType=acceptor
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ValidOrderTypes=1,2,F
SenderCompID=EXEC
TargetCompID=SYNAPSE
UseDataDictionary=Y
DefaultMarketPrice=12.30

[session]
BeginString=FIX.4.0
SocketAcceptPort=19876
```

Locate and edit the FIX configuration file of Banzai to be as follows. This file is usually named `banzai.cfg`.

```
[default]
FileStorePath=examples/target/data/banbai
ConnectionType=initiator
SenderCompID=BANZAI
TargetCompID=SYNAPSE
SocketConnectHost=localhost
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ReconnectInterval=5

[session]
BeginString=FIX.4.0
SocketConnectPort=9876
```

The `FileStorePath` property in the above two files should point to two directories in your local file system. The launcher scripts for the sample application can be found in the `bin` directory of Quickfix/J distribution.

### **Setting up FIX Transport**

To run the FIX samples used in this guide, you need a local Quickfix/J installation (<http://www.quickfixj.org>). Download Quickfix/J from: <http://www.quickfixj.org/downloads>.

To enable the FIX transport for samples, first you must deploy the Quickfix/J libraries into the `repository/components/lib` directory of the ESB. Generally the following libraries should be deployed into the ESB.

- quickfixj-core-1.4.0.jar
- quickfixj-msg-fix40-1.4.0.jar
- quickfixj-msg-fix42-1.4.0.jar
- quickfixj-msg-fix41-1.4.0.jar
- quickfixj-msg-fix43-1.4.0.jar
- quickfixj-msg-fix44-1.4.0.jar
- quickfixj-msg-fix50-1.4.0.jar
- mina-core-1.1.0.jar
- slf4j-jdk14-1.5.3.jar
- slf4j-api-1.5.3.jar

Then uncomment the FIX transport sender and FIX transport receiver configurations in the `repository/conf/axis2.xml`. Simply locate and uncomment the `FIXTransportSender` and `FIXTransportListener` sample configurations. Alternatively if the FIX transport management bundle is in use, you can enable the FIX transport listener and the sender from the WSO2 ESB management console. Login to the console and navigate to "Transports" on management menu. Scroll down to locate the sections related to the FIX transport. Simply click on the "Enable" links to enable the FIX listener and the sender.

#### **Configuring WSO2 ESB for FIX Samples**

In order to configure WSO2 ESB to run the FIX samples given in this guide, you will need to create some FIX configuration files as specified below (you can find the config files from `<ESB_HOME>/repository/samples/resources/fix` folder).

The `FileStorePath` property in the following two files should point to two directories in your local file system. Once the samples are executed, Synapse will create FIX message stores in these two directories.

Put the following entries in a file called `fix-synapse.cfg`.

```
[default]
FileStorePath=repository/logs/fix/data
ConnectionType=acceptor
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ValidOrderTypes=1,2,F
SenderCompID=SYNAPSE
TargetCompID=BANZAI
UseDataDictionary=Y
DefaultMarketPrice=12.30

[session]
BeginString=FIX.4.0
SocketAcceptPort=9876
```

Put the following entries in a file called `synapse-sender.cfg`.

```
[default]
FileStorePath=repository/logs/fix/data
SocketConnectHost=localhost
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ReconnectInterval=5
 SenderCompID=SYNAPSE
 TargetCompID=EXEC
 ConnectionType=initiator

[session]
BeginString=FIX.4.0
SocketConnectPort=19876
```

## Sample 258: Switching from HTTP to FIX

Objective: Demonstrate switching from HTTP to FIX.

### Prerequisites

- You will need the Executor sample application that comes with Quickfix/J. Configure the Executor to establish a session with Synapse. Configuring Sample FIX Applications.
- Start the Executor.
- Enable FIX transport sender in the Synapse axis2.xml.
- Configure Synapse for FIX samples. See Configuring WSO2 ESB for FIX Samples. There is no need to create the fix-synapse.cfg file for this sample. Having only the synapse-sender.cfg file is sufficient.
- Go to the ESB\_HOME/repository/samples/synapse\_sample\_258.xml file and make sure that trans port.fix.InitiatorConfigURL property points to the synapse-sender.cfg file you created. Once done, you can start the Synapse configuration numbered 258: wso2esb-samples.sh -sn 258
- Invoke the FIX Client as follows. This command sends a FIX message embedded in a SOAP message over HTTP.

```
ant fixclient -Dsymbol=IBM -Dqty=5 -Dmode=buy
-Daddurl=http://localhost:8280/services/FIXProxy
```

Synapse will create a session with Executor and forward the order request. The first response coming from the Executor will be sent back over HTTP. Executor generally sends two responses for each incoming order request. But since the response has to be forwarded over HTTP, only one can be sent back to the client.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="FIXProxy">
 <parameter
name="transport.fix.InitiatorConfigURL">file:/home/synapse_user/fix-config/synapse-sen
der.cfg</parameter>
 <parameter name="transport.fix.InitiatorMessageStore">file</parameter>
 <parameter name="transport.fix.SendAllToInSequence">false</parameter>
 <parameter name="transport.fix.DropExtraResponses">true</parameter>
 <target>
 <endpoint>
 <address
uri="fix:/localhost:19876?BeginString=FIX.4.0&SenderCompID=SYNAPSE&TargetCompID=EXEC"
/>
 </endpoint>
 <inSequence>
 <property name="transport.fix.ServiceName" value="FIXProxy" scope="axis2-client" />
 <log level="full" />
 </inSequence>
 <outSequence>
 <log level="full" />
 <send/>
 </outSequence>
 </target>
 </proxy>
 </definitions>

```

### **Configuring Sample FIX Applications**

If you are using a binary distribution of Quickfix/J, the two samples and their configuration files are all packed to a single JAR file called `quickfixj-examples.jar`. You will have to extract the JAR file, modify the configuration files and pack them to a JAR file again under the same name.

You can pass the new configuration file as a command line parameter too, in that case you do not need to modify the `quickfixj-examples.jar`. You can copy the config files from `$ESB_HOME/repository/samples/resources/fix` folder to `$QFJ_HOME/etc` folder. Execute the sample apps from `$QFJ_HOME/bin, ./banzai.sh/bat ..etc/ban` `zai.cfg executor.sh/bat ..etc/executor.cfg`.

Locate and edit the FIX configuration file of Executor to be as follows. This file is usually named `executor.cfg`.

```

[default]
FileStorePath=examples/target/data/executor
ConnectionType=acceptor
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ValidOrderTypes=1,2,F
SenderCompID=EXEC
TargetCompID=SYNAPSE
UseDataDictionary=Y
DefaultMarketPrice=12.30

[session]
BeginString=FIX.4.0
SocketAcceptPort=19876

```

Locate and edit the FIX configuration file of Banzai to be as follows. This file is usually named `banzai.cfg`.

```
[default]
FileStorePath=examples/target/data/banhai
ConnectionType=initiator
SenderCompID=BANZAI
TargetCompID=SYNAPSE
SocketConnectHost=localhost
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ReconnectInterval=5

[session]
BeginString=FIX.4.0
SocketConnectPort=9876
```

The `FileStorePath` property in the above two files should point to two directories in your local file system. The launcher scripts for the sample application can be found in the `bin` directory of Quickfix/J distribution.

#### ***Setting up FIX Transport***

To run the FIX samples used in this guide, you need a local Quickfix/J (<http://www.quickfixj.org>) installation. Download Quickfix/J from: <http://www.quickfixj.org/downloads>.

To enable the FIX transport for samples, first you must deploy the Quickfix/J libraries into the `repository/components/lib` directory of the ESB. Generally following libraries should be deployed into the ESB.

- `quickfixj-core-1.4.0.jar`
- `quickfixj-msg-fix40-1.4.0.jar`
- `quickfixj-msg-fix42-1.4.0.jar`
- `quickfixj-msg-fix41-1.4.0.jar`
- `quickfixj-msg-fix43-1.4.0.jar`
- `quickfixj-msg-fix44-1.4.0.jar`
- `quickfixj-msg-fix50-1.4.0.jar`
- `mina-core-1.1.0.jar`
- `slf4j-jdk14-1.5.3.jar`
- `slf4j-api-1.5.3.jar`

Then uncomment the FIX transport sender and FIX transport receiver configurations in the `repository/conf/axis2.xml`. Simply locate and uncomment the `FIXTransportSender` and `FIXTransportListener` sample configurations. Alternatively, if the FIX transport management bundle is in use, you can enable the FIX transport listener and the sender from the WSO2 ESB management console. Login to the console and navigate to "Transports" on management menu. Scroll down to locate the sections related to the FIX transport. Simply click on the "Enable" links to enable the FIX listener and the sender.

#### ***Configuring WSO2 ESB for FIX Samples***

In order to configure WSO2 ESB to run the FIX samples given in this guide, you will need to create some FIX configuration files as specified below (you can find the config files from `$ESB_HOME/repository/samples/resources/fixed` folder).

The `FileStorePath` property in the following two files should point to two directories in your local file system. Once the samples are executed, Synapse will create FIX message stores in these two directories.

Put the following entries in a file called `fix-synapse.cfg`.

```
[default]
FileStorePath=repository/logs/fix/data
ConnectionType=acceptor
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ValidOrderTypes=1,2,F
SenderCompID=SYNAPSE
TargetCompID=BANZAI
UseDataDictionary=Y
DefaultMarketPrice=12.30

[session]
BeginString=FIX.4.0
SocketAcceptPort=9876
```

Put the following entries in a file called `synapse-sender.cfg`.

```
[default]
FileStorePath=repository/logs/fix/data
SocketConnectHost=localhost
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ReconnectInterval=5
SenderCompID=SYNAPSE
TargetCompID=EXEC
ConnectionType=initiator

[session]
BeginString=FIX.4.0
SocketConnectPort=19876
```

### Sample 259: Switch from FIX to HTTP

Objective: Demonstrate the capability of switching form FIX to HTTP.

#### Prerequisites

- You will need the sample FIX blotter that come with Quickfix/J (Banzai). Configure the blotter to establish sessions with Synapse.
- Start the Axis2 server and deploy the `SimpleStockQuoteService` if not already deployed.
- Start Banzai.
- Enable FIX transport in the Synapse `axis2.xml`.
- Configure ESB for FIX samples.
- Open up the `ESB_HOME/repository/samples/synapse_sample_259.xml` file and make sure that `transport.fix.AcceptorConfigURL` property points to the `fix-synapse.cfg` file you created. Once done you can start the Synapse configuration numbered 259: `wso2esb-samples.sh -sn 259`

#### Note

Synapse creates a new FIX session with Banzai at this point.

- Send an order request from Banzai to Synapse. For example, Buy DELL 1000 @ 100. User has to send a

"Limit" Order because price is a mandatory field for placeOrder operation.

ESB will forward the order request to one-way placeOrder operation on the SimpleStockQuoteService. ESB uses a simple XSLT Mediator to transform the incoming FIX to a SOAP message.

```
<xsl:stylesheet version="2.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:fn="http://www.w3.org/2005/02/xpath-functions">
<xsl:output method="xml" omit-xml-declaration="yes" indent="yes" />
<xsl:template match="/" />
<m0:placeOrder xmlns:m0="http://services.samples">
 <m0:order>
 <m0:price><xsl:value-of select="//message/body/field[]"/></m0:price>
 <m0:quantity><xsl:value-of select="//message/body/field[]"/></m0:quantity>
 <m0:symbol><xsl:value-of select="//message/body/field[]"/></m0:symbol>
 </m0:order>
</m0:placeOrder>
</xsl:template>
</xsl:stylesheet>
```

To get an idea about the various transport parameters being used in this sample, see FIX Transport.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <localEntry key="xslt-key-req"
src="file:repository/samples/resources/transform/transform_fix_to_http.xslt" />
 <proxy name="FIXProxy" transports="fix">
 <target>
 <endpoint>
 <address
 uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 <inSequence>
 <log level="full" />
 <xslt key="xslt-key-req" />
 <log level="full" />
 </inSequence>
 <outSequence>
 <log level="full" />
 </outSequence>
 </target>
 <parameter name="transport.fix.AcceptorConfigURL">
 file:repository/samples/resources/fix/fix-synapse.cfg
 </parameter>
 <parameter name="transport.fix.AcceptorMessageStore">file</parameter>
 </proxy>
</definitions>
```

### Configuring Sample FIX Applications

If you use a binary distribution of Quickfix/J, the two samples and their configuration files are all packed to a single JAR file called quickfixj-examples.jar. You will have to extract the JAR file, modify the configuration files and pack them to a JAR file again under the same name.

You can pass the new configuration file as a command line parameter too, in that case you do not need to modify the quickfixj-examples.jar. You can copy the config files from \$ESB\_HOME/repository/samples/resources/fix folder to \$QFJ\_HOME/etc folder. Execute the sample applications from \$QFJ\_HOME/bin,"

```
" ./banzai.sh/bat ../etc/banzai.cfg executor.sh/bat ../etc/executor.cfg.
```

Locate and edit the FIX configuration file of Executor to be as follows. This file is usually named `executor.cfg`.

```
[default]
FileStorePath=examples/target/data/executor
ConnectionType=acceptor
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ValidOrderTypes=1,2,F
SenderCompID=EXEC
TargetCompID=SYNAPSE
UseDataDictionary=Y
DefaultMarketPrice=12.30

[session]
BeginString=FIX.4.0
SocketAcceptPort=19876
```

Locate and edit the FIX configuration file of Banzai to be as follows. This file is usually named `banzai.cfg`.

```
[default]
FileStorePath=examples/target/data/banzai
ConnectionType=initiator
SenderCompID=BANZAI
TargetCompID=SYNAPSE
SocketConnectHost=localhost
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ReconnectInterval=5

[session]
BeginString=FIX.4.0
SocketConnectPort=9876
```

The `FileStorePath` property in the above two files should point to two directories in your local file system. The launcher scripts for the sample application can be found in the `bin` directory of Quickfix/J distribution.

#### ***Setting up FIX Transport***

To run the FIX samples used in this guide, you need a local Quickfix/J (<http://www.quickfixj.org/downloads>) installation. Download Quickfix/J from: <http://www.quickfixj.org/downloads>.

To enable the FIX transport for samples, first you must deploy the Quickfix/J libraries into the `repository/components/lib` directory of the ESB. Generally following libraries should be deployed into the ESB.

- **quickfixj-core-1.4.0.jar**
- **quickfixj-msg-fix40-1.4.0.jar**
- **quickfixj-msg-fix42-1.4.0.jar**
- **quickfixj-msg-fix41-1.4.0.jar**
- **quickfixj-msg-fix43-1.4.0.jar**
- **quickfixj-msg-fix44-1.4.0.jar**
- **quickfixj-msg-fix50-1.4.0.jar**
- **mina-core-1.1.0.jar**

- **slf4j-jdk14-1.5.3.jar**
- **slf4j-api-1.5.3.jar**

Then uncomment the FIX transport sender and FIX transport receiver configurations in the `repository/conf/axis2.xml`. Simply locate and uncomment the `FIXTransportSender` and `FIXTransportListener` sample configurations. Alternatively, if the FIX transport management bundle is in use, you can enable the FIX transport listener and the sender from the WSO2 ESB management console. Login to the console and navigate to "Transports" on management menu. Scroll down to locate the sections related to the FIX transport. Simply click on the "Enable" links to enable the FIX listener and the sender.

#### **Configuring WSO2 ESB for FIX Samples**

In order to configure WSO2 ESB to run the FIX samples given in this guide, you will need to create some FIX configuration files as specified below (You can find the config files from `$ESB_HOME/repository/samples/resources/fix` folder).

The `FileStorePath` property in the following two files should point to two directories in your local file system. Once the samples are executed, Synapse will create FIX message stores in these two directories.

Put the following entries in a file called `fix-synapse.cfg`.

```
[default]
FileStorePath=repository/logs/fix/data
ConnectionType=acceptor
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ValidOrderTypes=1,2,F
SenderCompID=SYNAPSE
TargetCompID=BANZAI
UseDataDictionary=Y
DefaultMarketPrice=12.30

[session]
BeginString=FIX.4.0
SocketAcceptPort=9876
```

Put the following entries in a file called `synapse-sender.cfg`.

```
[default]
FileStorePath=repository/logs/fix/data
SocketConnectHost=localhost
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ReconnectInterval=5
SenderCompID=SYNAPSE
TargetCompID=EXEC
ConnectionType=initiator

[session]
BeginString=FIX.4.0
SocketConnectPort=19876
```

#### **Sample 260: Switch from FIX to AMQP**

**Objective:** Demonstrate the capability of switching between FIX and AMQP protocols.

#### Prerequisites

- You will need the sample FIX blotter that comes with Quickfix/J (Banzai). Configure the blotter to establish sessions with Synapse.
- Configure the AMQP transport for WSO2 ESB.
- Start the AMQP consumer, by switching to `samples/axis2Client` directory and running the consumer using the following command. Consumer will listen to the queue `QpidStockQuoteService`, accept the orders and reply to the queue `replyQueue`.

```
ant amqpconsumer
\-\-Dpropfile=\$ESB_HOME/repository/samples/resources/fix/direct.properties
```

- Start Banzai.
- Enable FIX transport in the Synapse `axis2.xml`.
- Configure Synapse for FIX samples.
- Open up the `ESB_HOME/repository/samples/synapse_sample_260.xml` file and make sure that the `transport.fix.AcceptorConfigURL` property points to the `fix-synapse.cfg` file you created. Once done, you can start the Synapse configuration numbered 260: `wso2esb-samples.sh -sn 260`

#### Note

Synapse creates a new FIX session with Banzai at this point.

- Send an order request from Banzai to Synapse. For example, Buy DELL 1000 @ MKT.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="FIXProxy" transports="fix">
 <target>
 <endpoint>
 <address
uri="jms:/QpidStockQuoteService?transport.jms.ConnectionFactoryJNDIName=qpidConnection
factory&&java.naming.factory.initial=org.apache.qpid.jndi.PropertiesFileInitialCont
extFactory&&java.naming.provider.url=repository/samples/resources/fix/con.propertie
s&&transport.jms.ReplyDestination=replyQueue"/>
 </endpoint>
 <inSequence>
 <log level="full" />
 </inSequence>
 <outSequence>
 <property name="transport.fix.ServiceName" value="FIXProxy"
scope="axis2-client" />
 <log level="full" />
 <send />
 </outSequence>
 </target>
 <parameter name="transport.fix.AcceptorConfigURL">
 file:repository/samples/resources/fix/fix-synapse.cfg
 </parameter>
 <parameter name="transport.fix.AcceptorMessageStore">file</parameter>
 </proxy>
</definitions>
```

Synapse will forward the order request by binding it to a JMS message payload and sending it to the AMQP

consumer. AMQP consumer will send a execution back to Banzai.

### ***Configuring Sample FIX Applications***

If you use a binary distribution of Quickfix/J, the two samples and their configuration files are all packed to a single JAR file called `quickfixj-examples.jar`. You will have to extract the JAR file, modify the configuration files and pack them to a JAR file again under the same name.

You can pass the new configuration file as a command line parameter too, in that case you do not need to modify the `quickfixj-examples.jar`. You can copy the config files from `$ESB_HOME/repository/samples/resources/fix` folder to `$QFJ_HOME/etc` folder. Execute the sample applications from `$QFJ_HOME/bin`, `./banza i.sh/bat ..etc/banhai.cfg executor.sh/bat ..etc/executor.cfg`.

Locate and edit the FIX configuration file of Executor to be as follows. This file is usually named `executor.cfg`.

```
[default]
FileStorePath=examples/target/data/executor
ConnectionType=acceptor
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ValidOrderTypes=1,2,F
SenderCompID=EXEC
TargetCompID=SYNAPSE
UseDataDictionary=Y
DefaultMarketPrice=12.30

[session]
BeginString=FIX.4.0
SocketAcceptPort=19876
```

Locate and edit the FIX configuration file of Banzai to be as follows. This file is usually named `banhai.cfg`.

```
[default]
FileStorePath=examples/target/data/banhai
ConnectionType=initiator
SenderCompID=BANZAI
TargetCompID=SYNAPSE
SocketConnectHost=localhost
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ReconnectInterval=5

[session]
BeginString=FIX.4.0
SocketConnectPort=9876
```

The `FileStorePath` property in the above two files should point to two directories in your local file system. The launcher scripts for the sample application can be found in the `bin` directory of Quickfix/J distribution.

### ***Setting up FIX Transport***

To run the FIX samples used in this guide, you need a local Quickfix/J (<http://www.quickfixj.org>) installation. Download Quickfix/J from: <http://www.quickfixj.org/downloads>.

To enable the FIX transport for samples, first you must deploy the Quickfix/J libraries into the `repository/compon`

ents/lib directory of the ESB. Generally, the following libraries should be deployed into the ESB.

- quickfixj-core-1.4.0.jar
- quickfixj-msg-fix40-1.4.0.jar
- quickfixj-msg-fix42-1.4.0.jar
- quickfixj-msg-fix41-1.4.0.jar
- quickfixj-msg-fix43-1.4.0.jar
- quickfixj-msg-fix44-1.4.0.jar
- quickfixj-msg-fix50-1.4.0.jar
- mina-core-1.1.0.jar
- slf4j-jdk14-1.5.3.jar
- slf4j-api-1.5.3.jar

Then uncomment the FIX transport sender and FIX transport receiver configurations in the repository/conf/axis2.xml. Simply locate and uncomment the FIXTransportSender and FIXTransportListener sample configurations. Alternatively if the FIX transport management bundle is in use, you can enable the FIX transport listener and the sender from the WSO2 ESB management console. Login to the console and navigate to "Transports" on management menu. Scroll down to locate the sections related to the FIX transport. Simply click on the "Enable" links to enable the FIX listener and the sender.

#### **Configuring WSO2 ESB for FIX Samples**

In order to configure WSO2 ESB to run the FIX samples given in this guide, you will need to create some FIX configuration files as specified below (You can find the config files from "\$ESB\_HOME/repository/samples/resources/fix folder.")

The FileStorePath property in the following two files should point to two directories in your local file system. Once the samples are executed, Synapse will create FIX message stores in these two directories.

Put the following entries in a file called fix-synapse.cfg.

```
[default]
FileStorePath=repository/logs/fix/data
ConnectionType=acceptor
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ValidOrderTypes=1,2,F
SenderCompID=SYNAPSE
TargetCompID=BANZAI
UseDataDictionary=Y
DefaultMarketPrice=12.30

[session]
BeginString=FIX.4.0
SocketAcceptPort=9876
```

Put the following entries in a file called synapse-sender.cfg.

```
[default]
FileStorePath=repository/logs/fix/data
SocketConnectHost=localhost
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ReconnectInterval=5
 SenderCompID=SYNAPSE
 TargetCompID=EXEC
 ConnectionType=initiator

[session]
BeginString=FIX.4.0
SocketConnectPort=19876
```

### **Configure ESB for AMQP Transport**

The samples used in this guide assumes the existence of a local QPid (1.0-M2 or higher) installation properly installed and started up. You also need to copy the following client JAR files into the `repository/components/lib` directory to support AMQP. These files are found in the `lib` directory of the QPid installation.

```
qpidd-client-1.0-incubating-M2.jar
qpidd-common-1.0-incubating-M2.jar
geronimo-jms_1.1_spec-1.0.jar
slf4j-api-1.4.0.jar **
slf4j-log4j12-1.4.0.jar **
```

### **Note**

To configure FIX (Quickfix/J 1.3) with AMQP (QPid-1.0-M2), copy the "sl4j" - libraries comes with QPid and ignore the "sl4j" - libraries that come with Quickfix/J.

To enable the AMQP over JMS transport, you need to uncomment the JMS transport listener configuration. To enable AMQP over JMS for ESB, the `repository/conf/axis2.xml` must be updated, while to enable JMS support for the sample Axis2 server the `samples/axis2Server/repository/conf/axis2.xml` file must be updated.

```
<!--Uncomment this and configure as appropriate for JMS transport support, after
setting up your JMS environment -->
<transportReceiver name="jms" class="org.apache.synapse.transport.jms.JMSListener">
</transportReceiver>

<transportSender name="jms" class="org.apache.synapse.transport.jms.JMSSender">
</transportReceiver>
```

Locate and edit the AMQP connection settings file for the message consumer, this file is usually named `direct.properties` and can be found in the `repository/samples/resources/fix` directory.

```

java.naming.factory.initial = org.apache.qpid.jndi.PropertiesFileInitialContextFactory
register some connection factories
connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.qpidConnectionfactory =
amqp://guest:guest@clientid/test?brokerlist='tcp://localhost:5672'
Register an AMQP destination in JNDI
destination.[jniName] = [BindingURL]
destination.directQueue =
direct://amq.direct//QpidStockQuoteService?routingkey='QpidStockQuoteService'
destination.replyQueue = direct://amq.direct//replyQueue?routingkey='replyQueue'

```

Locate and edit the AMQP connection settings file for WSO2 ESB, this file is usually named `con.properties` and can be found in the `repository/samples/resources/fix` directory.

```

#initial context factory
#java.naming.factory.initial =org.apache.qpid.jndi.PropertiesFileInitialContextFactory
register some connection factories
connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.qpidConnectionfactory=amqp://guest:guest@clientid/test?brokerlist='t
cp://localhost:5672'
Register an AMQP destination in JNDI
destination.[jniName] = [BindingURL]
destination.directQueue=direct://amq.direct//QpidStockQuoteService

```

## Sample 261: Switching between FIX Versions

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### ***Introduction***

This sample demonstrates how you can use WSO2 ESB to accept FIX input via the FIX transport layer and dispatch to another FIX acceptor that accept messages in a different FIX version. Here you will see how the ESB receives a FIX 4.0 messages and simply forwards it to the FIX 4.1 endpoint.

### ***Prerequisites***

- For a list of general prerequisites, see [Prerequisites to Start the ESB Samples](#).
- To configure the ESB to use the FIX transport, see [Configure the ESB to use the FIX transport](#).
- You will need the two sample FIX applications (*Banzai* and *Executor*) that are provided with Quickfix/J. Configure the two applications to establish sessions with the ESB.
- Add the following line to the `fix-synapse-m40.cfg` and `synapse-sender-m.cfg` configuration files.

```
DataDictionary=~/etc/spec/FIX40-synapse.xml
```

### **Note**

The `FIX40-synapse.xml` file can be found in the `<ESB_HOME>/repository/samples/resour`

`ces/fix` directory. This is a custom FIX data dictionary file that adds the tag 150 and 151 to the execution message(35=8) of FIX4.0. Make sure the `DataDictionary` property in the `banzai.cfg` file points to this data dictionary file.

- Add the following lines to `executor.cfg`, which is the *Executor* configuration file:

```
[session]
BeginString=FIX.4.1
SocketAcceptPort=19877
```

- Start *Banzai* and *Executor* using the custom configuration files.
- You need to use two custom configuration files for the ESB in this sample. The two custom configuration files can be found in the `<ESB_HOME>/repository/samples/resources/fix` directory. The two files are `fix-synapse-m40.cfg` and `synapse-sender-m.cfg`. You need to point your ESB configuration to these two files (this is already done in the provided `synapse_sample_261.xml` file) or you should create copies of them and point the ESB configuration to the copies. In either case, make sure that the properties `FileStoragePath` and `FileLogPath` in the two files point to valid locations in your local file system.
- Open the `<ESB_HOME>/repository/samples/synapse_sample_261.xml` file and make sure that the `transport.fix.AcceptorConfigURL` property points to the `fix-synapse-m40.cfg` file and the `transport.fix.InitiatorConfigURL` property points to the `synapse-sender-m.cfg` file.

## Note

The ESB creates a new FIX session with *Banzai* at this point.

### **Building the sample**

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="OrderProcessorProxy41" transports="fix">
 <target>
 <endpoint>
 <address
uri="fix://localhost:19877?BeginString=FIX.4.1&SenderCompID=SYNAPSE&TargetCompID=EXEC"
/>
 </endpoint>
 <inSequence>
 <log level="full"/>
 </inSequence>
 <outSequence>
 <log level="full"/>
 <send/>
 </outSequence>
 </target>
 <parameter
name="transport.fix.AcceptorConfigURL">file:repository/samples/resources/fix/fix-synapse-m40.cfg</parameter>
 <parameter name="transport.fix.AcceptorMessageStore">file</parameter>
 <parameter
name="transport.fix.InitiatorConfigURL">file:repository/samples/resources/fix/synapse-sender-m.cfg</parameter>
 <parameter name="transport.fix.InitiatorMessageStore">file</parameter>
 </proxy>
 </definitions>

```

This configuration file `synapse_sample_261.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

- Start the ESB with the sample 261 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).  
The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds*".

### **Executing the sample**

Send an order request from *Banzai* to the ESB. For example, Buy DELL 1000 @ MKT.

### **Analyzing the output**

You will see that the ESB forwards the FIX4.0 order request to the *Executor* that accepts FIX4.1 messages, and that the *Executor* processes the request and sends a response back to *Banzai*.

## **Sample 262: CBR of FIX Messages**

Objective: Demonstrate the capability of CBR FIX messages.

### **Prerequisites**

- You will need the two sample FIX applications that come with Quickfix/J (*Banzai* and *Executor*). Configure the two applications to establish sessions with Synapse.
- Add the following lines to `banzai.cfg`.

```
DataDictionary=~/etc/spec/FIX40-synapse.xml
```

## Note

`FIX40-synapse.xml` can be found at `$ESB_HOME/repository/samples/resources/fix`. This is a custom FIX data dictionary file that has added tag 150,151 to the execution messages (35=8) of FIX4.0. Make sure the `DataDictionary` property of the `banzai.cfg` points to this data dictionary file.

- Add the following lines to `executor.cfg`.

```
[session]
BeginString=FIX.4.1
SocketAcceptPort=19877
```

- Start Banzai and Executor using the custom config files.
- Enable FIX transport in the Synapse `axis2.xml`.
- Configure Synapse for FIX samples.
- Open up the `ESB_HOME/repository/samples/synapse_sample_262.xml` file and make sure that `transport.fix.AcceptorConfigURL` property points to the `fix-synapse.cfg` file you created and `transport.fix.InitiatorConfigURL` points to the `synapse-sender.cfg` file you created. Once done, you can start the Synapse configuration numbered 262: `wso2esb-samples.sh -sn 262`

## Note

Synapse creates a new FIX session with Banzai at this point.

- Send an order request from Banzai to Synapse. For example, Buy GOOG 1000 @ MKT, Buy MSFT 3000 @ MKT, Buy SUNW 500 @ 100.20.
- Synapse will forward the order requests with symbol "GOOG" to FIX endpoint FIX-4.0 @ localhost:19876.
- Synapse will forward the order requests with symbol "MSFT" to FIX endpoint FIX-4.1 @ localhost:19877.
- Synapse will not forward the orders with other symbols to any endpoint (default case has kept blank in the configuration).

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="CBR_SEQ">
 <in>
 <switch source="//message/body/field@id='55'">
 <case regex="GOOG">
 <send>
 <endpoint>
 <address
 uri="fix://localhost:19876?BeginString=FIX.4.0&SenderCompID=SYNAPSE&TargetCompID=EXEC"
 />
 </endpoint>
 </send>
 </case>
 <case regex="MSFT">
 <send>
 <endpoint>
 <address
 uri="fix://localhost:19877?BeginString=FIX.4.1&SenderCompID=SYNAPSE&TargetCompID=EXEC"
 />
 </endpoint>
 </send>
 </case>
 <default></default>
 </switch>
 </in>
 <out>
 <send />
 </out>
 </sequence>
 <proxy name="FIXProxy" transports="fix">
 <target inSequence="CBR_SEQ" />
 <parameter name="transport.fix.AcceptorConfigURL">
 file:repository/samples/resources/fix/fix-synapse.cfg
 </parameter>
 <parameter name="transport.fix.AcceptorMessageStore">
 file
 </parameter>
 <parameter name="transport.fix.InitiatorConfigURL">
 file:repository/samples/resources/fix/synapse-sender.cfg
 </parameter>
 <parameter name="transport.fix.InitiatorMessageStore">
 file
 </parameter>
 </proxy>
</definitions>

```

### **Configuring Sample FIX Applications**

If you use a binary distribution of Quickfix/J, the two samples and their configuration files are all packed to a single JAR file called `quickfixj-examples.jar`. You will have to extract the JAR file, modify the configuration files and pack them to a JAR file again under the same name.

You can pass the new configuration file as a command line parameter too, in that case you do not need to modify the `quickfixj-examples.jar`. You can copy the config files from `$ESB_HOME/repository/samples/resou`

rces/fix folder to \$QFJ\_HOME/etc folder. Execute the sample apps from \$QFJ\_HOME/bin, ./banzai.sh/bat ..../etc/banhai.cfg executor.sh/bat ..../etc/executor.cfg.

Locate and edit the FIX configuration file of Executor to be as follows. This file is usually named executor.cfg.

```
[default]
FileStorePath=examples/target/data/executor
ConnectionType=acceptor
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ValidOrderTypes=1,2,F
SenderCompID=EXEC
TargetCompID=SYNAPSE
UseDataDictionary=Y
DefaultMarketPrice=12.30

[session]
BeginString=FIX.4.0
SocketAcceptPort=19876
```

Locate and edit the FIX configuration file of Banzai to be as follows. This file is usually named banzai.cfg.

```
[default]
FileStorePath=examples/target/data/banhai
ConnectionType=initiator
SenderCompID=BANZAI
TargetCompID=SYNAPSE
SocketConnectHost=localhost
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ReconnectInterval=5

[session]
BeginString=FIX.4.0
SocketConnectPort=9876
```

The FileStorePath property in the above two files should point to two directories in your local file system. The launcher scripts for the sample application can be found in the bin directory of Quickfix/J distribution.

#### **Configuring WSO2 ESB for FIX Samples**

In order to configure WSO2 ESB to run the FIX samples given in this guide, you will need to create some FIX configuration files as specified below (you can find the config files from \$ESB\_HOME/repository/samples/resources/fix folder).

The FileStorePath property in the following two files should point to two directories in your local file system. Once the samples are executed, Synapse will create FIX message stores in these two directories.

Put the following entries in the file called fix-synapse.cfg.

```
[default]
FileStorePath=repository/logs/fix/data
ConnectionType=acceptor
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ValidOrderTypes=1,2,F
SenderCompID=SYNAPSE
TargetCompID=BANZAI
UseDataDictionary=Y
DefaultMarketPrice=12.30

[session]
BeginString=FIX.4.0
SocketAcceptPort=9876
```

Put the following entries in the file called `synapse-sender.cfg`.

```
[default]
FileStorePath=repository/logs/fix/data
SocketConnectHost=localhost
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ReconnectInterval=5
SenderCompID=SYNAPSE
TargetCompID=EXEC
ConnectionType=initiator

[session]
BeginString=FIX.4.0
SocketConnectPort=19876
```

### Sample 263: Transport Switching - JMS to http/s Using JBoss Messaging(JBM)

**Objective:** Introduction to switching transports with proxy services. The JMS provider will be JBoss Messaging(JBM) (<http://www.jboss.org/jbossmessaging/>).

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="jms">
 <target>
 <inSequence>
 <property action="set" name="OUT_ONLY" value="true" />
 </inSequence>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/conf/sample/resources/proxy/sample_proxy_1.wsdl"/>
 <parameter name="transport.jms.ContentType">
 <rules>
 <jmsProperty>contentType</jmsProperty>
 <default>application/xml</default>
 </rules>
 </parameter>
 </proxy>
</definitions>

```

### Prerequisites:

- Start the Axis2 server and deploy the SimpleStockQuoteService (Refer steps above)
- Download , install and start JBM server (<http://www.jboss.org/jbossmessaging>), and configure Synapse to listen on JBM (refer notes below)

Start the ESB configuration numbered 263: i.e. wso2esb-samples -sn 263. We need to configure the required queues in JBM. Add the following entry to JBM jms configuration inside file-config/stand-alone/non-clustered/jbm-jms.xml.

```

<queue name="StockQuoteProxy">
 <entry name="StockQuoteProxy" />
</queue>

```

Once you started the JBM server with the above changes you'll be able to see the following on STDOUT.

```

10:18:02,673 INFO [org.jboss.messaging.core.server.impl.MessagingServerImpl] JBoss
Messaging Server version 2.0.0.BETA3 (maggot, 104) started

```

You also need to copy the jbm-core-client.jar, jbm-jms-client.jar, jnp-client.jar(these jars are inside \$JBOSS\_MESSAGEING/client folder) and jbm-transports.jar, netty.jar(these jars are from \$JBOSS\_MESSAGEING/lib folder) jars from JBM into the \$ESB\_HOME/repository/components/lib directory. This was tested with JBM 2.0.0.BETA3

You need to add the following configuration for Axis2 JMS transport listener in axis2.xml found at repository/conf/axis2.xml.

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="default" locked="false">
 <parameter
name="java.naming.factory.initial">org.jnp.interfaces.NamingContextFactory</parameter>
 <parameter name="java.naming.provider.url">jnp://localhost:1099</parameter>
 <parameter
name="java.naming.factory.url.pkgs">org.jboss.naming:org.jnp.interfaces</parameter>
 <parameter
name="transport.jms.ConnectionFactoryJNDIName">ConnectionFactory</parameter>
 </parameter>
 </transportReceiver>

```

Once you start ESB configuration 263 and request for the WSDL of the proxy service (<http://localhost:8280/services/StockQuoteProxy?wsdl>) you will notice that its exposed only on the JMS transport. This is because the configuration specified this requirement in the proxy service definition.

Before running the JMS client you need to open the build.xml ant script and uncomment the following block under the jmsclient target.

```

<!--<sysproperty key="java.naming.provider.url" value="${java.naming.provider.url}"/>
<sysproperty key="java.naming.factory.initial"
value="${java.naming.factory.initial}"/> <sysproperty
key="java.naming.factory.url.pkg" value="${java.naming.factory.url.pkg}"/>-->

```

Now lets send a stock quote request on JMS, using the dumb stock quote client as follows:

```

ant jmsclient -Djms_type=pox -Djms_dest=StockQuoteProxy -Djms_payload=MSFT
-Djava.naming.provider.url=jnp://localhost:1099
-Djava.naming.factory.initial=org.jnp.interfaces.NamingContextFactory

```

-Djava.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces

On the ESB debug(you'll need to enable debug log in ESB) log you will notice that the JMS listener received the request message as:

```

[JMSWorker-1] DEBUG ProxyServiceMessageReceiver -Proxy Service StockQuoteProxy
received a new message...

```

Now if you examine the console running the sample Axis2 server, you will see a message indicating that the server has accepted an order as follows:

```

Accepted order for : 16517 stocks of MSFT at $ 169.14622538721846

```

In this sample, the client sends the request message to the proxy service exposed over JMS in Synapse. Synapse forwards this message to the HTTP EPR of the simple stock quote service hosted on the sample Axis2 server. Note that the operation is out-only and no response is sent back to the client. The transport.jms.ContentType property is necessary to allow the JMS transport to determine the content type of incoming messages. With the given configuration it will first try to read the content type from the 'contentType' message property and fall back to 'application/xml' (i.e. POX) if this property is not set. Note that the JMS client used in this example doesn't send any content type information.

## Note

It is possible to instruct a JMS proxy service to listen to an already existing destination without creating a new one. To do this, use the parameter elements on the proxy service definition to specify the destination and connection factory etc. For example:

```
<parameter
name="transport.jms.Destination">dynamicTopics/something.TestTopic</parameter>
```

### Sample 264: Sending Two-Way Messages Using JMS transport

**Objective:** Demonstrate sending request response scenario with JMS transport

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="http">
 <target>
 <endpoint>
 <address
uri="jms:/SimpleStockQuoteService?transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory&

java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory&jav
a.naming.provider.url=tcp://localhost:61616&transport.jms.DestinationType=queue"/>
 </endpoint>
 <inSequence>
 <property action="set" name="transport.jms.ContentTypeProperty"
value="Content-Type" scope="axis2"/>
 </inSequence>
 <outSequence>
 <property action="remove" name="TRANSPORT_HEADERS" scope="axis2"/>
 <send/>
 </outSequence>
 <target>
 <publishWSDL
uri="file:repository/conf/sample/resources/proxy/sample_proxy_1.wsdl"/>
 </target>
 </proxy>
</definitions>
```

#### Prerequisites:

- You need to set up ESB and axis2 server to use the JMS transport. See [Sample 251](#) for more details.

This sample is similar to [Sample 251](#). Only difference is we are expecting a response from the server. JMS transport uses **transport.jms.ContentTypeProperty** to determine the content type of the response message. If this property is not set JMS transport treats the incoming message as plain text.

In the out path we remove the message context property **TRANSPORT\_HEADERS**. If these property is not removed JMS headers will be passed to the client.

Start ESB using sample 264.

```
./wso2esb-samples.sh -sn 264
```

Start Axis2 server with SimpleStockService deployed

Invoke the stockquote client using the following command.

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy -Dsymbol=MSFT
```

The sample Axis2 server console will print a message indicating that it has received the request:

```
Generating quote for : MSFT
```

In the client side it should print a message indicating it has received the price.

```
Standard :: Stock price = $154.31851804993238
```

## Sample 265: Accessing a Windows Share Using the VFS Transport

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### ***Introduction***

This sample demonstrates how to use the [VFS transport](#) to access a windows share.

### ***Prerequisites***

- Create a directory named test on a windows machine and create three sub directories named in, out and original within the test directory.
- Grant permission to the network users to read from and write to the **test** directory and sub directories.
- Open the ESB\_HOME/repository/samples/synapse\_sample\_265.xml file in a text editor and change the `transport.vfs.FileURI`, `transport.vfs.MoveAfterProcess`, `transport.vfs.MoveAfterFailure` parameter values to the **in**, **original** and **original** directory locations respectively. You have to also change the `<outSequence>` endpoint address uri to the **out** directory location. Make sure that the prefix `vfs:` in the endpoint address uri is not removed or changed.
- Enable the VFS transport. For details, see [Enable VFS](#).
- For a list of general prerequisites, see [Prerequisites to Start the ESB Samples](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

```

<!-- Using the vfs transport to access a windows share -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" transports="vfs">
 <parameter name="transport.vfs.FileURI">smb://host/test/in</parameter> <!--CHANGE-->
 <parameter name="transport.vfs.ContentType">text/xml</parameter>
 <parameter name="transport.vfs.FileNamePattern">.*\..xml</parameter>
 <parameter name="transport.PollInterval">15</parameter>
 <parameter
name="transport.vfs.MoveAfterProcess">smb://host/test/original</parameter>
<!--CHANGE-->
 <parameter
name="transport.vfs.MoveAfterFailure">smb://host/test/original</parameter>
<!--CHANGE-->
 <parameter name="transport.vfs.ActionAfterProcess">MOVE</parameter>
 <parameter name="transport.vfs.ActionAfterFailure">MOVE</parameter>

 <target>
 <endpoint>
 <address format="soap12"
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 <outSequence>
 <property name="transport.vfs.ReplyFileName"

expression="fn:concat(fn:substring-after(get-property('MessageID'), 'urn:uuid:'),
'.xml')" scope="transport"/>
 <property action="set" name="OUT_ONLY" value="true"/>
 <send>
 <endpoint>
<address uri="vfs:smb://host/test/out"/> <!--CHANGE-->
 </endpoint>
 </send>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 </proxy>
</definitions>

```

This configuration file `synapse_sample_265.xml` is available in the `<ESB_HOME>/repository/samples` directory and the values you have to change as specified in the prerequisites section are marked with `<!--CHANGE-->`.

## To build the sample

1. Start the ESB with the sample 265 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).  
The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"
2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

## **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

## To execute the sample client

Move the `test.xml` file from the `<ESB_HOME>/repository/samples/resources/vfs` directory to the location specified in `transport.vfs.FileURI` in the configuration (i.e., the `in` directory).

The `test.xml` file contains a simple stock quote request in XML/SOAP format and is as follows:

### test.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:wsa="http://www.w3.org/2005/08/addressing">
 <soapenv:Body>
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>IBM</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </soapenv:Body>
</soapenv:Envelope>
```

## Analyzing the output

You will see that the VFS transport listener picks the file from the `in` directory and sends it to the Axis2 service over HTTP. Then you will see that the request XML file is moved to the `original` directory and that the response from the Axis2 server is saved to the `out` directory.

## Sample 266: Switching from TCP to HTTP/S

**Objective:** Demonstrate receiving SOAP messages over TCP and forwarding them over HTTP

```
<definitions xmlns="http://ws.apache.org/ns/synapse"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ws.apache.org/ns/synapse
http://synapse.apache.org/ns/2010/04/configuration/synapse_config.xsd">

 <proxy name="StockQuoteProxy" transports="tcp">
 <target>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 <inSequence>
 <log level="full"/>
 <property name="OUT_ONLY" value="true"/>
 </inSequence>
 </target>
 </proxy>

</definitions>
```

## Prerequisites:

- Configure ESB to use the TCP transport and configure sample Axis2 client to send TCP requests. See [here](#) for details on how to do this.
- Start Synapse using sample 266: ie `synapse -sample 266`
- Start Axis2 server with SimpleStockService deployed

This sample is similar to Sample 250 . Only difference is instead of the JMS transport we will be using the TCP transport to receive messages. TCP is not an application layer protocol. Hence there are no application level headers available in the requests. ESB has to simply read the XML content coming through the socket and dispatch it to the right proxy service based on the information available in the message payload itself. The TCP transport is capable of dispatching requests based on addressing headers or the first element in the SOAP body. In this sample, we will get the sample client to send WS-Addressing headers in the request. Therefore the dispatching will take place based on the addressing header values.

Invoke the stockquote client using the following command. Note the TCP URL in the command.

```
ant stockquote -Daddurl=tcp://localhost:6060/services/StockQuoteProxy
-Dmode=placeorder
```

The TCP transport will receive the message and hand it over to the mediation engine. Synapse will dispatch the request to the StockQuoteProxy service based on the addressing header values.

The sample Axis2 server console will print a message indicating that it has received the request:

```
Thu May 20 12:25:01 IST 2010 samples.services.SimpleStockQuoteService :: Accepted
order #1 for : 17621 stocks of IBM at $ 73.48068475255796
```

## Sample 267: Switching from UDP to HTTP/S

**Objective:** Demonstrate receiving SOAP messages over UDP and forwarding them over HTTP

```
<definitions xmlns="http://ws.apache.org/ns/synapse"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ws.apache.org/ns/synapse
http://synapse.apache.org/ns/2010/04/configuration/synapse_config.xsd">

 <proxy name="StockQuoteProxy" transports="udp">
 <target>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 <inSequence>
 <log level="full"/>
 <property name="OUT_ONLY" value="true" />
 </inSequence>
 </target>
 <parameter name="transport.udp.port">9999</parameter>
 <parameter name="transport.udp.contentType">text/xml</parameter>
 <publishWSDL
uri="file:repository/conf/sample/resources/proxy/sample_proxy_1.wsdl"/>
 </proxy>
</definitions>
```

### Prerequisites:

- Configure Synapse to use the UDP transport. The sample Axis2 client should also be setup to send UDP requests.
- Start Synapse: `synapse -sample 267`
- Start Axis2 server with SimpleStockService deployed

This sample is similar to [Sample 266](#). Only difference is instead of the TCP transport we will be using the UDP transport to receive messages. Invoke the stockquote client using the following command. Note the TCP URL in the command.

```
ant stockquote -Daddurl=udp://localhost:9999?contentType=text/xml -Dmode=placeorder
```

Since we have configured the content type as text/xml for the proxy service, incoming messages will be processed as SOAP 1.1 messages. The sample Axis2 server console will print a message indicating that it has received the request:

```
Thu May 20 12:25:01 IST 2010 samples.services.SimpleStockQuoteService :: Accepted
order #1 for : 17621 stocks of IBM at $ 73.48068475255796
```

## Sample 268: Proxy Services with the Local Transport

**Objective:** Proxy services with the Local transport.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy xmlns="http://ws.apache.org/ns/synapse" name="LocalTransportProxy"
 transports="https http" startOnLoad="true" trace="disable">
 <target>
 <endpoint name="ep1">
 <address uri="local://services/SecondProxy"/>
 </endpoint>
 <inSequence>
 <log level="full"/>
 <log level="custom">
 <property name="LocalTransportProxy" value="In sequence of
LocalTransportProxy invoked!"/>
 </log>
 </inSequence>
 <outSequence>
 <log level="custom">
 <property name="LocalTransportProxy" value="Out sequence of
LocalTransportProxy invoked!"/>
 </log>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 </proxy>
 <proxy xmlns="http://ws.apache.org/ns/synapse" name="SecondProxy"
 transports="https http" startOnLoad="true" trace="disable">
 <target>
 <endpoint name="ep2">
 <address uri="local://services/StockQuoteProxy"/>
 </endpoint>
 <inSequence>
 <log level="full"/>
 <log level="custom">
 <property name="SecondProxy" value="In sequence of Second proxy
invoked!"/>
 </log>
 </inSequence>
```

```
<outSequence>
 <log level="custom">
 <property name="SecondProxy" value="Out sequence of Second proxy
invoked! "/>
 </log>
 <send/>
</outSequence>
</target>
<publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
</proxy>
<proxy xmlns="http://ws.apache.org/ns/synapse" name="StockQuoteProxy"
startOnLoad="true">
 <target>
 <endpoint name="ep3">
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 <outSequence>
 <log level="custom">
 <property name="StockQuoteProxy"
value="Out sequence of StockQuote proxy invoked! "/>
 </log>
 <log level="full"/>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
```

```

 uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 </proxy>
</definitions>

```

This sample contains three proxy services. The stockquote client invokes the LocalTransportProxy. Then the message will be sent to the SecondProxy and then it will be sent to the StockQuoteProxy. The StockQuoteProxy will invoke the backend service and return the response to the client. In this sample, the communication between proxy services are done through the Local transport. Since Local transport calls are in-JVM calls, it will reduce the time taken for the communication between proxy services.

1. Enable [Non Blocking Local Transport](#) for the ESB.
2. Start the Synapse configuration numbered 268: `wso2esb-samples -sn 268`
3. Start the Axis2 server and deploy the SimpleStockQuoteService if not already done
4. Execute the stock quote client by requesting for a stock quote on the proxy service as follows:

```
ant stockquote -Daddurl=http://localhost:8280/services/LocalTransportProxy
```

### Sample 270: Transport switching from HTTP to MSMQ and MSMQ to HTTP

Objective: Demonstrate the capability of working MSMQ transport messages.

#### Objective: Introduction to switching transports with proxy services

##### Prerequisites:

please make sure that the given MSMQ sample is ONLY working on windows environment, since  
it invokes Microsoft C++ API for MSMQ via JNI invocation.

- Start the Axis2 server and deploy the SimpleStockQuoteService (Refer steps above).
- Download axis2-transport-msmq-1.1.0-wso2v6.jar ([click here for 64x , 32x](#)) and place that in ESB\_HOME/repository/components/dropins.
  - MSMQ bridging requires JNI invocation and we are shipping two dlls as required for 64bit and 32bit O/S, so make sure that you are downloading the correct one.
- Please make sure MQ installed and running for more information please refer <http://msdn.microsoft.com/en-us/library/aa967729.aspx>.
- Make sure that you have installed Visual C++ 2008 (VC9), it works with Microsoft Visual Studio 2008 Express.

For a default MSMQ v4.0 installation, you may place following in the Axis2 transport sender/ listener configuration at repository/conf/axis2/axis2.xml as,

```

<transportSender name="msmq" class="org.apache.axis2.transport.msmq.MSMQSender" />
<transportReceiver name="msmq" class="org.apache.axis2.transport.msmq.MSMQListener">

 <parameter name="msmq.receiver.host" locked="false">localhost</parameter>
</transportReceiver>

```

Synapse Configuration for MSMQ,

```

<proxy name="msmqTest" transports="msmq" startOnLoad="true">
 <target>
 <inSequence>
 <property name="OUT_ONLY" value="true" />
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
</proxy>
<proxy name="StockQuoteProxy" transports="http" startOnLoad="true" trace="disable">
 <description/>
 <target>
 <endpoint>
 <address uri="msmq:DIRECT=OS:localhost\private$\msmqTest" />
 </endpoint>
 <inSequence>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"
type="STRING"/>
 <property name="OUT_ONLY" value="true" scope="default" type="STRING"/>
 </inSequence>
 <outSequence>
 <log level="custom">
 <property name="MESSAGE" value="OUT SEQUENCE CALLED" />
 </log>
 <send/>
 </outSequence>
 </target>
 <publishWSDL uri="http://localhost:9000/services/SimpleStockQuoteService?wsdl" />
</proxy>

```

Invoke the sample as follows,

```

ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
-Dmode=placeorder -Dsymbol=MSFT

```

The sample Axis2 server console will print a message indicating that it has accepted the order as follows,

```

Accepted order for : 18406 stocks of MSFT at $ 83.58806051152119

```

Above samples works as follows,

- Sending Place stockquote request to the ESB proxy.
- Proxy sends the incoming message to the MSMQ server.

```
msmq:DIRECT=OS:localhost\private$msmqTest
```

- Another proxy known as 'msmqTest' listening to the MSMQ queue, invoke the message from MSMQ and send to the Aix2 back-end server.

## Sample 271: File Processing

- Introduction
- Building the sample
  - Set up the database
  - Create the main and fault sequences
  - Configure the ESB
  - Add database drivers
  - Add smooks libraries
  - Create and configure FileProxy
  - Create and configure databaseSequence
  - Create and Configure fileWriteSequence
  - Create and configure sendMailSequence
  - Create the input file
- Executing the sample
- Analyzing the output

### ***Introduction***

This sample demonstrates how to pick a file from a directory and process it within the ESB. In this sample scenario you pick a file from the local directory, insert the records in the file to a database, send an email with the file content, trace and write the log and finally move the file to another directory.

The following diagram displays the entities involved in this example.



### **Note**

This example works with the MySQL database.

### ***Building the sample***

All files required for this sample are in [sample.zip](#).

Follow the steps given below to build this sample.

### Set up the database

1. Manually set up the database.
2. Create a table named `info` in your schema. You can run the following commands to do this.

```
delimiter $$

CREATE TABLE `info` (
 `name` varchar(45) DEFAULT '',
 `surname` varchar(45) DEFAULT NULL,
 `phone` varchar(45) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8$$
```

3. Make sure the `info` table is created and that it contains the following columns:

- `name`
- `surname`
- `phone`

The result of the query should be as follows when you query to view the records in the `test.info` table. You will see that there is no data in the table.

The screenshot shows the MySQL Workbench interface. On the left, the Object Browser displays the schema structure under the 'test' database, with the 'info' table highlighted. In the center, the Query Editor window contains the SQL command `select * from test.info;`. Below it, the Results tab shows a table header with columns `name`, `surname`, and `phone`. At the bottom, the Object Information pane provides detailed information about the 'info' table, listing its columns: `name` (45), `surname` (45), and `phone` (45), all defined as `varchar`.

### Create the main and fault sequences

1. Find the `main.xml` and `fault.xml` files in the attached `sample.zip` archive. You can find the files in the `<SAMPLE_HOME>/conf/synapse-config/sequences` directory.
2. Copy the files to `<ESB_HOME>/repository/deployment/server/synapse-configs/default/sequences` folder.

### Note

The `main` and `fault` sequences are created and preconfigured automatically when you install WSO2 ESB.

## Configure the ESB

You need to configure the ESB to use the [VFS transport](#) for processing the files, and the [MailTo transport](#) for sending the email message. You also need to configure the message formatter as specified.

- Edit the `<ESB_HOME>/repository/conf/axis2/axis2.xml` file and uncomment the VFS listener and the VFS sender as follows:

```
<transportReceiver name="vfs"
class="org.apache.synapse.transport.vfs.VFSTransportListener" />
...
<transportSender name="vfs"
class="org.apache.synapse.transport.vfs.VFSTransportSender" />
```

- Edit the `<ESB_HOME>/repository/conf/axis2/axis2.xml` file and configure the mailto transport sender as follows to use a mailbox for sending the messages:

```
<transportSender name="mailto"
class="org.apache.axis2.transport.mail.MailTransportSender">
 <parameter name="mail.smtp.host">smtp.gmail.com</parameter>
 <parameter name="mail.smtp.port">587</parameter>
 <parameter name="mail.smtp.starttls.enable">true</parameter>
 <parameter name="mail.smtp.auth">true</parameter>
 <parameter name="mail.smtp.user">username</parameter>
 <parameter name="mail.smtp.password">userpassword</parameter>
 <parameter name="mail.smtp.from">username@gmail.com</parameter>
</transportSender>
```

### Note

In this sample, you will not retrieve mails from a mailbox. Therefore, you do not need to enable the `mailto` transport receiver.

- Add the following message formatter in the `<ESB_HOME>/repository/conf/Axis2/axis2.xml` file under the `Message Formatters` section:

```
<messageFormatter contentType="text/html"
class="org.apache.axis2.transport.http.ApplicationXMLFormatter"/>
```

## Add database drivers

1. Find the MySQL database driver `mysql-connector-java-5.1.10-bin.jar` in the attached `sample.zip` archive. You can find the file in the `<SAMPLE_HOME>/lib` directory.
2. Copy the driver to the `<WSO2ESB_HOME>/repository/components/lib` directory.

## Add smooks libraries

This example uses a CSV smooks library.

1. You can find the CSV smooks library `milyn-smooks-csv-1.2.4.jar` in the attached `sample.zip` archive. You can find the file in the `<SAMPLE_HOME>/lib` directory.
2. Copy the library to the `<WSO2ESB_HOME>/repository/components/lib` directory.

### Note

These configuration changes make system-wide changes to the ESB and the ESB has to be restarted for these changes to take effect.

3. Configure a local entry as follows. For information on how to add a local entry, see [Adding a Local Entry via the Management Console](#). This local entry will be used to refer to the smooks configuration saved in the `<SAMPLE_HOME>/resources/smooks-config.xml` file.

```
<localEntry key="smooks" src="file:resources/smooks-config.xml"/>
```

## Create and configure FileProxy

1. You can find the `FileProxy.xml` file in the attached `sample.zip` archive. It is located in the `<SAMPLE_HOME>/conf/synapse-config/proxy-services` directory.

The XML code of the sequence is as follows:

```

<proxy xmlns="http://ws.apache.org/ns/synapse" name="FileProxy" transports="vfs"
startOnLoad="true" trace="disable">
 <target>
 <inSequence>
 <log level="full"/>
 <clone>
 <target sequence="fileWriteSequence" />
 <target sequence="sendMailSequence" />
 <target sequence="databaseSequence" />
 </clone>
 </inSequence>
 </target>
 <parameter name="transport.vfs.ActionAfterProcess">MOVE</parameter>
 <parameter name="transport.PollInterval">15</parameter>
 <parameter
name="transport.vfs.MoveAfterProcess">file:///home/username/test/original</parameter>
 <parameter
name="transport.vfs.FileURI">file:///home/username/test/in</parameter>
 <parameter
name="transport.vfs.MoveAfterFailure">file:///home/username/test/failure</parameter>
 <parameter name="transport.vfs.FileNamePattern">.*.txt</parameter>
 <parameter name="transport.vfs.ContentType">text/plain</parameter>
 <parameter name="transport.vfs.ActionAfterFailure">MOVE</parameter>
</proxy>

```

2. Edit the FileProxy.xml file, and define the directory to which the original file should be moved after processing.

```

<parameter
name="transport.vfs.MoveAfterProcess">[file:///home/]<username>/test/original</parameter>

```

3. Edit the FileProxy.xml file, and define where the input file should be placed.

```

<parameter
name="transport.vfs.FileURI">[file:///home/]<username>/test/in</parameter>

```

4. Edit the FileProxy.xml file, and define the directory to which the file should be moved if an error occurs.

```

<parameter
name="transport.vfs.MoveAfterFailure">[file:///home/]<username>/test/failure</parameter>

```

5. Save the FileProxy.xml file to the <ESB\_HOME>/repository/deployment/server/synapse-configs/default/proxy-services directory.

#### Create and configure databaseSequence

Follow the instructions below to create a sequence that can be used to connect to the database to insert the data.

1. You can find the databaseSequence.xml file in the attached sample.zip archive. It is located in the <SAMPLE\_HOME>/conf/synapse-config/sequences directory.

The XML code of the database sequence is as follows.

```

<sequence xmlns="http://ws.apache.org/ns/synapse" name="databaseSequence">
 <log level="full">
 <property name="sequence" value="before-smooks"/>
 </log>
 <smooks config-key="smooks">
 <input type="text"/>
 <output type="xml"/>
 </smooks>
 <log level="full">
 <property name="sequence" value="after-smooks"/>
 </log>
 <iterate xmlns:ns2="http://org.apache.synapse/xsd"
 xmlns:ns="http://org.apache.synapse/xsd"
 xmlns:sec="http://secservice.samples.esb.wso2.org"
 expression="//csv-records/csv-record">
 <target>
 <sequence>
 <dbreport>
 <connection>
 <pool>
 <password>db-password</password>
 <user>db-username</user>
 <url>jdbc:mysql://localhost:3306/test</url>
 <driver>com.mysql.jdbc.Driver</driver>
 </pool>
 </connection>
 <statement>
 <sql>insert into userdetails values (?, ?, ?, ?, ?, ?)</sql>
 <parameter expression="//csv-record/name/text()" type="VARCHAR"/>
 <parameter expression="//csv-record/surname/text()" type="VARCHAR"/>
 <parameter expression="//csv-record/phone/text()" type="VARCHAR"/>
 </statement>
 </dbreport>
 </sequence>
 </target>
 </iterate>
</sequence>
```

2. Specify your database username, password, and URL in the <pool> section of the sequence.
3. Save the databaseSequence.xml file to the <ESB\_HOME>/repository/deployment/server/synapse-configs/default/sequences directory.

## Create and Configure fileWriteSequence

1. You can find the fileWriteSequence.xml in the attached sample.zip archive. It is located in the <SAMPLE\_HOME>/conf/synapse-config/sequences directory.

The XML code of the sequence is as follows:

```

<sequence xmlns="http://ws.apache.org/ns/synapse" name="fileWriteSequence">
 <log level="custom">
 <property name="sequence" value="fileWriteSequence"/>
 </log>
 <property xmlns:ns2="http://org.apache.synapse/xsd"
 name="transport.vfs.ReplyFileName"
 expression="fn:concat(fn:substring-after(get-property('MessageID'), 'urn:uuid:'),
 '.txt')" scope="transport"/>
 <property name="OUT_ONLY" value="true"/>
 <send>
 <endpoint name="FileEpr">
 <address uri="vfs:file:///home/username/test/out"/>
 </endpoint>
 </send>
</sequence>

```

2. Edit the `fileWriteSequence.xml` file, and define the directory to which the file should be moved.
3. Save the `fileWriteSequence.xml` file to the `<ESB_HOME>/repository/deployment/server/synapse-configs/default/sequences` directory.

### Create and configure sendMailSequence

1. You can find the `sendMailSequence.xml` file in the attached `sample.zip` archive. It is located in the `<SAMPLE_HOME>/conf/synapse-config/sequences` directory.

The XML code of the sequence is as follows:

```

<sequence xmlns="http://ws.apache.org/ns/synapse" name="sendMailSequence">
 <log level="custom">
 <property name="sequence" value="sendMailSequence"/>
 </log>
 <property name="messageType" value="text/html" scope="axis2"/>
 <property name="ContentType" value="text/html" scope="axis2"/>
 <property name="Subject" value="File Received" scope="transport"/>
 <property name="OUT_ONLY" value="true"/>
 <send>
 <endpoint name="FileEpr">
 <address uri="mailto:username@gmail.com"/>
 </endpoint>
 </send>
</sequence>

```

2. Edit the `sendMailSequence.xml` file, and define the e-mail address to which the notification should be sent.
3. Save the `sendMailSequence.xml` file to the `<ESB_HOME>/repository/deployment/server/synapse-configs/default/sequences` directory.

### Create the input file

- Create a text file in the following format.

```
name_1, surname_1, phone_1
name_2, surname_2, phone_2
```

```
GNU nano 2.2.4 File: input.txt
Don,Smith,123456789
John,Smith,987654321
```

### Executing the sample

- Save the file in the .txt format to the `in` directory that you specified in step 3, under the create and configure FileProxy section.

### Analyzing the output

In this sample, the ESB listens on a local file system directory. When a file is dropped into the `in` directory, the ESB picks this file.

- Make sure the file appears in the `out` directory.
- The ESB inserts the records from the text file to the database. Make sure the data is in the `info` table. The following screenshot displays the content of the `test.info` table with the data from the file.

	name	surname	phone
▶	Don	Smith	123456789
▶	John	Smith	987654321

- Make sure the original file is moved to the `/home/<username>/test/original` directory.
- Make sure the e-mail notification is sent to the email address that is specified. The message should contain the file data. The following screenshot displays a notification received.

wso2examples@email.com to me [show details](#) 10:40 (3 minutes ago) [Reply](#) ▾

Don,Smith,123456789  
John,Smith,987654321

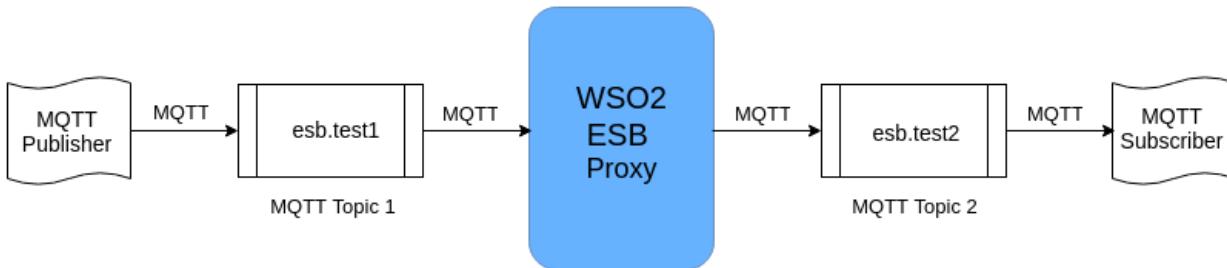
[Reply](#) [Forward](#)

## Sample 272: Publishing and Subscribing using WSO2 ESB's MQ Telemetry Transport

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

### **Introduction**

This sample demonstrates how WSO2 ESB's MQ Telemetry Transport (MQTT) listener consumes messages from a MQTT topic, and how the MQ Telemetry Transport (MQTT) sender publishes messages to a MQTT topic.



### **Prerequisites**

- Download the `org.eclipse.paho.client.mqttv3-1.1.0.jar` file.
- Download the Mosquitto-clients (<http://mosquitto.org/>)
- Download **WSO2 MB 3.1.0** or the mosquito MQTT broker (<http://mosquitto.org/>)

### **Building the sample**

1. Copy the `org.eclipse.paho.client.mqttv3-1.1.0.jar` file to the `<ESB_HOME>/repository/components/lib` directory.
2. Edit the `<ESB_HOME>/repository/conf/axis2/axis2.xml` file and change the MQTT sender and listener configuration to be as follows:

```

<transportReceiver class="org.apache.axis2.transport.mqtt.MqttListener"
name="mqtt">
 <parameter locked="false" name="mqttConFactory">
 <parameter locked="false"
name="mqtt.server.host.name">localhost</parameter>
 <parameter locked="false"
name="mqtt.server.port">1883</parameter>
 <parameter locked="false"
name="mqtt.client.id">esb.test.listener</parameter>
 <parameter locked="false"
name="mqtt.topic.name">esb.test1</parameter>
 </parameter>
</transportReceiver>

<transportSender class="org.apache.axis2.transport.mqtt.MqttSender" name="mqtt">
</transportSender>

```

3. Start the MQTT broker. If you are using WSO2 MB as the MQTT broker, you should set the WSO2 ESB port offset to 1 before running the ESB. To set the port offset in WSO2 ESB, open the `<ESB_HOME>/repository/conf/carbon.xml` file and set the offset to 1 as follows:

```
<Offset>1</Offset>
```

4. Start WSO2 MB, open the Management Console and [create a topic named esb.test2](#).
5. Start the ESB with the sample 272 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#). The XML configuration for this sample is as follows:

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="SampleProxy" transports="mqtt" startOnLoad="true"
trace="disable">
 <description/>
 <target>
 <endpoint>
 <address
uri="mqtt:/SampleProxy?mqtt.server.host.name=localhost&mqtt.server.port=1883&
amp;mqtt.client.id=esb.test.sender&mqtt.topic.name=esb.test2&mqtt.subscription.qos=2&mqtt.blocking.sender=true"/>
 </endpoint>
 <inSequence>
 <property name="OUT_ONLY" value="true"/>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"
type="STRING"/>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <parameter name="mqtt.connection.factory">mqttConFactory</parameter>
 <parameter name="mqtt.topic.name">esb.test1</parameter>
 <parameter name="mqtt.subscription.qos">2</parameter>
 <parameter name="mqtt.content.type">text/plain</parameter>
 <parameter name="mqtt.session.clean">false</parameter>
 </proxy>
</definitions>
```

This configuration file `synapse_sample_272.xml` is available in the `<ESB_HOME>/repository/samples` directory.

#### ***Executing the sample***

- Execute the following command to start the MQTT subscriber on the `esb.test2` topic:

```
mosquitto_sub -h localhost -t esb.test2
```

- Execute the following command to run the MQTT publisher to publish to the `esb.test1` topic:

```
mosquitto_pub -h localhost -p 1883 -t esb.test1 -m {"company": "WSO2"}
```

#### ***Analyzing the output***

When you analyze the output messages on the MQTT subscriber console, you will see the following log:

```
{ "company" : "WSO2" }
```

## Introduction to ESB Tasks

The following Tasks samples are available with WSO2 Enterprise Service Bus (ESB) :

- Sample 300: Introduction to Tasks with a Simple Trigger

### Sample 300: Introduction to Tasks with a Simple Trigger

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

#### *Introduction*

This sample introduces the concept of [tasks](#) and demonstrates how a simple trigger works. Here the `MessageInjector` class is used, which injects a specified message to the ESB environment. You can write your own task class implementing the `org.apache.synapse.startup.Task` interface and implement the `execute` method to run the task.

#### *Prerequisites*

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

#### *Building the sample*

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <task class="org.apache.synapse.startup.tasks.MessageInjector"
group="synapse.simple.quartz" name="CheckPrice">
 <property name="to"
value="http://localhost:9000/services/SimpleStockQuoteService"/>
 <property name="soapAction" value="urn:getQuote"/>
 <property name="message">
 <m0:quote xmlns:m0="http://services.samples"
xmlns="http://ws.apache.org/ns/synapse">
 <m0:request>
 <m0:symbol>IBM</m0:symbol>
 </m0:request>
 </m0:quote>
 </property>
 <trigger interval="5" />
 </task>
 <sequence name="main">
 <in>
 <send/>
 </in>
 <out>
 <log level="custom">
 <property name="Stock_Quote_on"
expression="//ns:return/ax21:lastTradeTimestamp/child::text()"
 xmlns:ax21="http://services.samples/xsd"
xmlns:ns="http://services.samples"/>
 <property name="For_the_organization"
expression="//ns:return/ax21:name/child::text()"
 xmlns:ax21="http://services.samples/xsd"
xmlns:ns="http://services.samples"/>
 <property name="Last_Value"
expression="//ns:return/ax21:last/child::text()"
 xmlns:ax21="http://services.samples/xsd"
xmlns:ns="http://services.samples"/>
 </log>
 </out>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_300.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Build and deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).
2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).

### **Executing the sample**

- Start the ESB with the sample 300 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The `synapse_sample_300.xml` configuration adds a scheduled task to the ESB runtime so that when you start the ESB, this task will run periodically every 5 seconds. You can limit the number of times that you want the task to run by adding a `count` attribute with an integer as the `value`. If the `count` is not present as in this sample, the task will run forever.

### **Analyzing the output**

You will see that the Axis2 server generates a quote every 5 seconds and that the ESB receives the stock quote response. This is because the injected message is sent to the sample Axis2 server, which sends back a response to the ESB.

## **Advanced Mediations with Advanced Mediators**

The following advanced mediation scenarios are available with WSO2 Enterprise Service Bus (ESB) :

- Using Scripts in Mediation (Script Mediator)
  - Sample 350: Introduction to the Script Mediator Using JavaScript
  - Sample 351: Inline script mediation with JavaScript
  - Sample 352: Accessing Synapse message context API using a scripting language
  - Sample 353: Using Ruby Scripts for Mediation
  - Sample 354: Using Inline Ruby Scripts for Mediation
- Database Interactions in Mediation (DBLookup / DBReport)
  - Sample 360: Introduction to DBLookup Mediator
  - Sample 361: Introduction to DBReport Mediator
  - Sample 362: DBReport and DBLookup Mediators Together
  - Sample 363: Reusable Database Connection Pools
  - Sample 364: Using Mediators to Execute Database Stored Procedures
- Throttling Messages (Throttle Mediator)
  - Sample 370: Introduction to Throttle Mediator and Concurrency Throttling
  - Sample 371: Restricting Requests Based on Policies
  - Sample 372: Use of Both Concurrency Throttling and Request-Rate-Based Throttling
- Extending the Mediation in Java (Class Mediator)
  - Sample 380: Writing your own Custom Mediation in Java
  - Sample 381: Class Mediator to CBR Binary Messages
- Evaluating XQuery for Mediation (XQuery Mediator)
  - Sample 390: Introduction to XQuery Mediator
  - Sample 391: Using Data from an External XML Document within XQuery
- Splitting Messages into Parts and Processing in Parallel (Iterate/Aggregate)
  - Sample 400: Message Splitting and Aggregating the Responses
- Caching Responses Over Requests (Cache Mediator)
  - Sample 420: Simple Cache Implemented on ESB for the Actual Service
- Mediating JSON Messages
  - Sample 440: Converting JSON to XML Using XSLT
  - Sample 441: Converting JSON to XML Using JavaScript
- Rewriting the URL (URL Rewrite Mediator)
  - Sample 450: Introduction to the URL Rewrite Mediator
  - Sample 451: Conditional URL Rewriting
  - Sample 452: Conditional URL Rewriting with Multiple Rules
- Eventing (Event Mediator)
  - Sample 460: Introduction to Eventing and Event Mediator
- Mediating with Spring
  - Sample 470: How to Initialize and use a Spring Bean as a Mediator

### **Using Scripts in Mediation (Script Mediator)**

The following samples demonstrate how to use the [Script mediator](#):

- Sample 350: Introduction to the Script Mediator Using JavaScript
- Sample 351: Inline script mediation with JavaScript
- Sample 352: Accessing Synapse message context API using a scripting language
- Sample 353: Using Ruby Scripts for Mediation
- Sample 354: Using Inline Ruby Scripts for Mediation

See also [Sample 441: Converting JSON to XML Using JavaScript](#).

For tips and best practices for using JavaScript, see <http://jstherightway.org/>.

## Sample 350: Introduction to the Script Mediator Using JavaScript

**Objective:** Demonstrate how to use scripts in mediation using the Script mediator

The ESB Script Mediator is a ESB extension, and thus all prerequisites are not bundled by default with the ESB distribution. Before you use some scripts, you may need to manually add the required JAR files to the ESB lib directory, and optionally perform other installation tasks as may be required by the individual scripting language. This is explained in the [Samples Setup guide](#).

### Prerequisites:

- Start the Synapse configuration numbered 350: i.e. `wso2esb-samples.sh -sn 350`
- Start the Axis2 server and deploy the `SimpleStockQuoteService` if not already done.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <localEntry key="stockquoteScript"
src="file:repository/samples/resources/script/stockquoteTransform.js"/>
 <sequence name="main">
 <in>
 <!-- transform the custom quote request into a standard quote request
expected by the service -->
 <script language="js" key="stockquoteScript" function="transformRequest"/>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </in>
 <out>
 <!-- transform the standard response back into the custom format the
client expects -->
 <script language="js" key="stockquoteScript"
function="transformResponse"/>
 <send/>
 </out>
 </sequence>
</definitions>
```

The content of `stockquoteTransform.js` is as follows:

```

function transformRequest(mc) {
 var symbol = mc.getPayloadXML()...*::Code.toString();
 mc.setPayloadXML(
 <m:getQuote xmlns:m="http://services.samples">
 <m:request>
 <m:symbol>{symbol}</m:symbol>
 </m:request>
 </m:getQuote>);
}

function transformResponse(mc) {
 var symbol = mc.getPayloadXML()...*::symbol.toString();
 var price = mc.getPayloadXML()...*::last.toString();
 mc.setPayloadXML(
 <m:CheckPriceResponse xmlns:m="http://services.samples/xsd">
 <m:Code>{symbol}</m:Code>
 <m:Price>{price}</m:Price>
 </m:CheckPriceResponse>);
}

```

This sample is similar to [sample 8](#), but instead of using XSLT, the transformation is done with JavaScript and E4X. The script used in this example has two functions, 'transformRequest' and 'transformResponse', and the Synapse configuration uses the function attribute to specify which function should be invoked. Use the stock quote client to issue a custom quote client as follows.

```

ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dmode=customquote

```

ESB uses the Script mediator and the specified JavaScript function to convert the custom request to a standard quote request. Subsequently the response received is transformed and sent back to the client.

For additional examples, see [Using Scripts in Mediation \(Script Mediator\)](#).

### Sample 351: Inline script mediation with JavaScript

Objective: Introduction to inline script mediation

#### Prerequisites:

- Start the Synapse configuration numbered 351: i.e. `wso2esb-samples.sh -sn 351`
- Start the Axis2 server and deploy the `SimpleStockQuoteService` if not already done.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <in>
 <!-- transform the custom quote request into a standard quote request expected
by the service -->
 <script language="js"><![CDATA[
 var symbol = mc.getPayloadXML()...*::Code.toString();
 mc.setPayloadXML(
 <m:getQuote xmlns:m="http://services.samples/xsd">
 <m:request>
 <m:symbol>{symbol}</m:symbol>
 </m:request>
 </m:getQuote>);
]]></script>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </in>
 <out>
 <!-- transform the standard response back into the custom format the client
expects -->
 <script language="js"><![CDATA[
 var symbol = mc.getPayloadXML()...*::symbol.toString();
 var price = mc.getPayloadXML()...*::last.toString();
 mc.setPayloadXML(
 <m:CheckPriceResponse xmlns:m="http://services.samples/xsd">
 <m:Code>{symbol}</m:Code>
 <m:Price>{price}</m:Price>
 </m:CheckPriceResponse>);
]]></script>
 <send/>
 </out>
</definitions>

```

The functionality of this sample is similar to that of [sample 350](#) and [sample 8](#). It demonstrates how to use inline scripts in the mediation within the ESB. Use the stock quote client to issue a custom quote client as follows.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dmode=customquote
```

ESB uses the script mediator and the specified Javascript function to convert the custom request to a standard quote request. Subsequently the response received is transformed and sent back to the client.

## Sample 352: Accessing Synapse message context API using a scripting language

Objective: Accessing the Synapse APIs from scripting languages

Prerequisites:

- Start the Synapse configuration numbered 352: i.e. `wso2esb-samples.sh -sn 352`
- Start the Axis2 server and deploy the SimpleStockQuoteService if not already done.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <in>
 <!-- change the MessageContext into a response and set a response payload -->
 <script language="js"><![CDATA[
 mc.setTo(mc.getReplyTo());
 mc.setProperty("RESPONSE", "true");
 mc.setPayloadXML(
 <ns:getQuoteResponse xmlns:ns="http://services.samples/xsd">
 <ns:return>
 <ns:last>99.9</ns:last>
 </ns:return>
 </ns:getQuoteResponse>);
]]></script>
 </in>
 <send/>
</definitions>

```

This example shows how an inline JavaScript mediator script could access the Synapse message context API to set its 'To' EPR and to set a custom property to mark it as a response. Execute the stock quote client, and you will receive the response "99.9" as the last sale price as per the above script.

```

ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/

```

### Sample 353: Using Ruby Scripts for Mediation

**Objective:** Script mediators using Ruby

## XML

```
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <localEntry key="stockquoteScript"
src="file:repository/samples/resources/script/stockquoteTransform.rb"/>
 <in>
 <!-- transform the custom quote request into a standard quote request expected
by the service -->
 <script language="rb" key="stockquoteScript" function="transformRequest"/>

 <!-- send message to real endpoint referenced by name "stockquote" and stop
-->
 <send>
 <endpoint name="stockquote">
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </in>
 <out>
 <!-- transform the standard response back into the custom format the client
expects -->
 <script language="rb" key="stockquoteScript" function="transformResponse"/>
 <send/>
 </out>
 </definitions>
```

## Ruby

```
<x><![CDATA[
require 'rexml/document'
include REXML

def transformRequest(mc)
 newRequest= Document.new '<m:getQuote xmlns:m="http://services.samples/xsd">'<<
 '<m:request><m:symbol></m:symbol></m:request></m:getQuote>'
 newRequest.root.elements[1].elements[1].text =
mc.getPayloadXML().root.elements[1].get_text
 mc.setPayloadXML(newRequest)
end

def transformResponse(mc)
 newResponse = Document.new '<m:CheckPriceResponse
xmlns:m="http://services.samples/xsd"><m:Code>' <<
 '</m:Code><m:Price></m:Price></m:CheckPriceResponse>'
 newResponse.root.elements[1].text =
mc.getPayloadXML().root.elements[1].elements[1].get_text
 newResponse.root.elements[2].text =
mc.getPayloadXML().root.elements[1].elements[2].get_text
 mc.setPayloadXML(newResponse)
end
]]></x>
```

**Prerequisites:**

This sample uses Ruby so first set up Ruby support as described in [Configuring the ESB for Script Mediator Support](#).

- Start the Synapse configuration numbered 353: i.e. wso2esb-samples -sn 353
- Start the Axis2 server and deploy the SimpleStockQuoteService if not already done

This sample is functionally equivalent to sample # 350 (#351 and #8) but instead uses a Ruby script using the JRuby interpreter. The script has two functions, 'transformRequest' and 'transformResponse', and the Synapse configuration specifies to be invoked when used. Execute the stock quote client to send a custom stock quote as per [Sample 350](#) and check the received stock quote response.

**Sample 354: Using Inline Ruby Scripts for Mediation**

**Objective:** Script mediators using Ruby (inline Ruby script)

```
<!-- Using In-lined Ruby scripts for mediation -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <in>
 <script language="rb">
 <![CDATA[
 require 'rexml/document'
 include REXML
 newRequest= Document.new '<m:getQuote
xmlns:m="http://services.samples/xsd"><m:request><m:symbol>...test...</m:symbol></m:request></m:getQuote>
 newRequest.root.elements[1].elements[1].text =
$mc.getPayloadXML().root.elements[1].get_text
 $mc.setPayloadXML(newRequest)
]]>
 </script>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </in>
 <out>
 <script language="rb">
 <![CDATA[
 require 'rexml/document'
 include REXML
 newResponse = Document.new '<m:CheckPriceResponse
xmlns:m="http://services.samples/xsd"><m:Code></m:Code><m:Price></m:Price></m:CheckPriceResponse>
 newResponse.root.elements[1].text =
$mc.getPayloadXML().root.elements[1].elements[1].get_text
 newResponse.root.elements[2].text =
$mc.getPayloadXML().root.elements[1].elements[2].get_text
 $mc.setPayloadXML(newResponse)
]]>
 </script>
 <send/>
 </out>
</definitions>
```

**Prerequisites:**

- This sample uses Ruby, so first set up Ruby support as described at [Configuring the ESB for Script Mediator Support](#).
- Start the Synapse configuration numbered 354: i.e. wso2esb-samples -sn 354
- Start the Axis2 server and deploy the SimpleStockQuoteService if not already done

This sample is functionally equivalent to [Sample 353](#).

Run the client with

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dmode=customquote
```

### **Database Interactions in Mediation (DBLookup / DBReport)**

Following database mediators use Derby in a client/server configuration by using the network server. Therefore, to proceed with the following samples, you need a working Derby database server and you have to follow the steps in [ESB Sample Setup](#) before going through the samples.

### **Database Interactions in Mediation (DBLookup / DBReport)**

The following samples demonstrate how to use the [DBLookup](#) and [DBReport](#) mediators:

- Sample 360: Introduction to DBLookup Mediator
- Sample 361: Introduction to DBReport Mediator
- Sample 362: DBReport and DBLookup Mediators Together
- Sample 363: Reusable Database Connection Pools
- Sample 364: Using Mediators to Execute Database Stored Procedures

#### **Sample 360: Introduction to DBLookup Mediator**

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

#### **Introduction**

This sample demonstrates how to perform a simple database read operation database using the [DBLookup](#) mediator.

In this sample, when a message arrives at the DBLookup mediator, it opens a connection to the database and executes the SQL query. The SQL query uses the ? character for attributes that are specified at runtime, and the parameters define how to calculate the value of those attributes. Here, the DBLookup mediator is used to extract the *id* of the company from the company database using the symbol that is evaluated using an XPath against the SOAP envelope, and the *id* base switching is done by a [Switch](#) mediator.

#### **Prerequisites**

- Set up the Apache Derby database. For information on how to set up the Derby database, see [Setting up Derby](#).
- For a list of general prerequisites, see [Prerequisites to Start the ESB Samples](#).

#### **Building the sample**

The XML configuration for this sample is as follows:

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
```

```

<sequence name="myFaultHandler">
 <makefault>
 <code value="tns:Receiver"
xmlns:tns="http://www.w3.org/2003/05/soap-envelope" />
 <reason expression="get-property('ERROR_MESSAGE')"/>
 </makefault>

 <property name="RESPONSE" value="true"/>
 <header name="To" expression="get-property('ReplyTo')"/>
 <send/>
 <drop/>
</sequence>

<sequence name="main" onError="myFaultHandler">
 <in>
 <log level="custom">
 <property name="text"
 value="** Looking up from the Database **"/>
 </log>
 <dblookup xmlns="http://ws.apache.org/ns/synapse">
 <connection>
 <pool>
 <driver>org.apache.derby.jdbc.ClientDriver</driver>
 <url>jdbc:derby://localhost:1527/esbdb;create=false</url>
 <user>esb</user>
 <password>esb</password>
 </pool>
 </connection>
 <statement>
 <sql>select * from company where name =?</sql>
 <parameter expression="//m0:getQuote/m0:request/m0:symbol"
 xmlns:m0="http://services.samples" type="VARCHAR" />
 <result name="company_id" column="id" />
 </statement>
 </dblookup>

 <switch source="get-property('company_id')">
 <case regex="c1">
 <log level="custom">
 <property name="text"
 expression="fn:concat('Company ID - "
',get-property('company_id'))"/>
 </log>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </case>
 <case regex="c2">
 <log level="custom">
 <property name="text"
 expression="fn:concat('Company ID - "
',get-property('company_id'))"/>
 </log>
 <send>
 <endpoint>
 <address

```

```

 uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </case>
 <case regex="c3">
 <log level="custom">
 <property name="text"
 expression="fn:concat('Company ID -
',get-property('company_id'))" />
 </log>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </case>
 <default>
 <log level="custom">
 <property name="text" value="** Unrecognized Company ID **"/>
 </log>
 <makefault>
 <code value="tns:Receiver"
 xmlns:tns="http://www.w3.org/2003/05/soap-envelope" />
 <reason value="** Unrecognized Company ID **"/>
 </makefault>
 <property name="RESPONSE" value="true" />
 <header name="To" action="remove" />
 <send/>
 <drop/>
 </default>
 </switch>
 <drop/>
 </in>

 <out>
 <send/>
 </out>

```

```
</sequence>

</definitions>
```

This configuration file `synapse_sample_360.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### To build the sample

1. Start the ESB with the sample 360 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds*".

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

### Executing the sample

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

#### To execute the sample client

- Run each of the following commands from the `<ESB_HOME>/samples/axis2Client` directory.

First, run the following command, specifying IBM as the stock symbol:

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=IBM
```

Then, run the following command, specifying SUN as the stock symbol.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=SUN
```

Next, run the following command, specifying MSFT as the stock symbol.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=MSFT
```

### Analyzing the output

When you run the command that requests the IBM stock quote, you will see the following output on the ESB console:

```
INFO LogMediator text = ** Looking up from the Database **INFO LogMediator text =
Company ID ? c1
```

When you run the command that requests the SUN stock quote, you will see the following output on the ESB console:

```
INFO LogMediator text = ** Looking up from the Database **INFO LogMediator text = Company ID ? c2
```

When you run the command that requests the MSFT stock quote, you will see the following output on the ESB console:

```
INFO LogMediator text = ** Looking up from the Database **
INFO LogMediator text = Company ID ? c2
```

If you run a command for any other symbol, you will see that the ESB console displays the following:

```
INFO LogMediator text = ** Unrecognized Company ID **
```

You will also observe that the client gets a response with the following message.

```
** Unrecognized Company ID **
```

### Sample 361: Introduction to DBReport Mediator

**Objective:** Demonstrate simple database write operations with the [DBReport mediator](#).

```

<definitions xmlns="http://ws.apache.org/ns/synapse">

 <sequence name="main">
 <in>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </in>

 <out>
 <log level="custom">
 <property name="text"
value="** Reporting to the Database **"/>
 </log>
 <dbreport xmlns="http://ws.apache.org/ns/synapse">
 <connection>
 <pool>
 <driver>org.apache.derby.jdbc.ClientDriver</driver>
 <url>jdbc:derby://localhost:1527/esbdb;create=false</url>
 <user>esb</user>
 <password>esb</password>
 </pool>
 </connection>
 <statement>
 <sql>update company set price=? where name =?</sql>
 <parameter expression="//m0:return/m1:last/child::text()"
xmlns:m0="http://services.samples"
xmlns:m1="http://services.samples/xsd" type="DOUBLE"/>
 <parameter expression="//m0:return/m1:symbol/child::text()"
xmlns:m0="http://services.samples"
xmlns:m1="http://services.samples/xsd" type="VARCHAR"/>
 </statement>
 </dbreport>
 <send/>
 </out>
 </sequence>
 </definitions>

```

## Prerequisites:

- Start the Synapse configuration numbered 361: i.e. wso2esb-samples -sn 361
  - Start the Axis2 server and deploy the SimpleStockQuoteService if not already done

The dbreport mediator writes (inserts one row) to a table using the message details. It works the same as the [DBLookup mediator](#). In this sample, the DBReport mediator is used for updating the stock price of the company using the last quote value, which is calculated by evaluating an XPath against the response message. After running this sample, user can check the company table using the Derby client tool. It will show the value inserted by the DBReport mediator.

Run the client using the following command:

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=IBM
```

and then execute the following query using the database client tool against synapseDB.

```
select price from company where name='IBM' ;
```

It will show some value as follows.

```
96.39535981018865
```

### Sample 362: DBReport and DBLookup Mediators Together

**Objective:** Demonstrate the use of the **DBReport** and **DBLookup** mediators together.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <sequence name="main">
 <in>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </in>

 <out>
 <log level="custom">
 <property name="text"
value="** Reporting to the Database **" />
 </log>

 <dbreport xmlns="http://ws.apache.org/ns/synapse">
 <connection>
 <pool>
 <driver>org.apache.derby.jdbc.ClientDriver</driver>
 <url>jdbc:derby://localhost:1527/esbdb;create=false</url>
 <user>esb</user>
 <password>esb</password>
 </pool>
 </connection>
 <statement>
 <sql>update company set price=? where name =?</sql>
 <parameter expression="//m0:return/m1:last/child::text()">
 <!-- m0 = http://services.samples -->
 </parameter>
 <parameter expression="//m0:return/m1:symbol/child::text()">
 <!-- m0 = http://services.samples -->
 </parameter>
 </statement>
 </dbreport>
 </out>
 </sequence>
</definitions>
```

```
<log level="custom">
 <property name="text"
 value="** Looking up from the Database **"/>
</log>
<dblookup xmlns="http://ws.apache.org/ns/synapse">
 <connection>
 <pool>
 <driver>org.apache.derby.jdbc.ClientDriver</driver>
 <url>jdbc:derby://localhost:1527/esbdb;create=false</url>
 <user>esb</user>
 <password>esb</password>
 </pool>
 </connection>
 <statement>
 <sql>select * from company where name =?</sql>
 <parameter expression="//m0:return/m1:symbol/child::text()">
 <!-- m0:services.samples -->
 <!-- m1:services.samples -->
 </parameter>
 </statement>
</dblookup>
<log level="custom">
 <property name="text"
 expression="fn:concat('Stock price -
',get-property('stock_price'))"/>
</log>
<send/>
</out>
```

```
</sequence>

</definitions>
```

### Prerequisites:

- Start the Synapse configuration numbered 362: i.e. wso2esb-samples -sn 362
- Start the Axis2 server and deploy the SimpleStockQuoteService if not already done

In this sample, the DBReport mediator works the same as the above sample. It updates the price for the given company using the response messages content. Then the DBLookup mediator reads the last updated value from the company database and logs it.

When running the client as follows:

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=IBM
```

The ESB console shows:

```
INFO LogMediator text = ** Reporting to the Database **
...
INFO LogMediator text = ** Looking up from the Database **
...
INFO LogMediator text = Stock price - 153.47886496064808
```

## Sample 363: Reusable Database Connection Pools

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### Introduction

This sample demonstrates how you can setup reusable connection pools for the [DBLookup](#) and [DB Report](#) mediator s.

### Prerequisites

- Setup a Derby database and the Synapse datasources.
  - For instructions on setting up a Derby database, see [Setting up Remote Derby](#).
  - For instructions on setting up a Synapse datasources, see [Setting up Synapse datasources](#).
- For a list of general prerequisites, see [Prerequisites to Start the ESB Samples](#).

### Building the sample

The XML configuration for this sample is as follows:

```
<!-- Reusable database connection pool -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="myFaultHandler">
 <makefault response="true">
```

```

 <code xmlns:tns="http://www.w3.org/2003/05/soap-envelope"
value="tns:Receiver"/>
 <reason expression="get-property('ERROR_MESSAGE')"/>
 </makefault>
 <send/>
 <drop/>
 </sequence>
<sequence name="main" onError="myFaultHandler">
 <in>
 <log level="custom">
 <property name="text" value="** Looking up from the Database **"/>
 </log>
 <dblookup>
 <connection>
 <pool>
 <dsName>lookupdb</dsName>
 </pool>
 </connection>
 <statement>
 <sql>select * from company where name =?</sql>
 <parameter xmlns:m0="http://services.samples"
expression="//m0:getQuote/m0:request/m0:symbol"
type="VARCHAR"/>
 <result name="company_id" column="id"/>
 </statement>
 </dblookup>
 <switch source="get-property('company_id')">
 <case regex="c1">
 <log level="custom">
 <property name="text" expression="fn:concat('Company ID -",
get-property('company_id'))"/>
 </log>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </case>
 <case regex="c2">
 <log level="custom">
 <property name="text" expression="fn:concat('Company ID -",
get-property('company_id'))"/>
 </log>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </case>
 <case regex="c3">
 <log level="custom">
 <property name="text" expression="fn:concat('Company ID -",
get-property('company_id'))"/>
 </log>
 <send>
 <endpoint>
 <address

```

```

uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
</case>
<default>
 <log level="custom">
 <property name="text" value="** Unrecognized Company ID **"/>
 </log>
 <makefault response="true">
 <code xmlns:tns="http://www.w3.org/2003/05/soap-envelope"
value="tns:Receiver" />
 <reason value="** Unrecognized Company ID **"/>
 </makefault>
 <send/>
 <drop/>
 </default>
</switch>
<drop/>
</in>
<out>
 <log level="custom">
 <property name="text" value="** Reporting to the Database **"/>
 </log>
 <dbreport>
 <connection>
 <pool>
 <dsName>reportdb</dsName>
 </pool>
 </connection>
 <statement>
 <sql>update company set price=? where name =?</sql>
 <parameter xmlns:m0="http://services.samples"
xmlns:m1="http://services.samples/xsd"
expression="//m0:return/m1:last/child::text()"
type="DOUBLE" />
 <parameter xmlns:m0="http://services.samples"
xmlns:m1="http://services.samples/xsd"
expression="//m0:return/m1:symbol/child::text()"
type="VARCHAR" />
 </statement>
 </dbreport>
 <log level="custom">
 <property name="text" value="** Looking up from the Database **"/>
 </log>
 <dblookup>
 <connection>
 <pool>
 <dsName>reportdb</dsName>
 </pool>
 </connection>
 <statement>
 <sql>select * from company where name =?</sql>
 <parameter xmlns:m0="http://services.samples"
xmlns:m1="http://services.samples/xsd"
expression="//m0:return/m1:symbol/child::text()"
type="VARCHAR" />
 <result name="stock_price" column="price"/>
 </statement>
 </dblookup>
 </log>

```

```
<log level="custom">
 <property name="text" expression="fn:concat('Stock price -
',get-property('stock_price'))" />
</log>
<send/>
```

```

 </out>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_363.xml` is available in the `<ESB_HOME>/repository/samples` directory.

When you go through this configuration, you will see that there are two instances of the dblookup mediator and a single instance of the dbreport mediator.

### To build the sample

1. Start the ESB with the sample 363 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds*".

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

### Executing the sample

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### To execute the sample client

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/
```

### Analyzing the output

If you look at debug log output on the ESB console, you will see the following:

```

INFO LogMediator text = ** Looking up from the Database **
...
INFO LogMediator text = Company ID - c1
...
INFO LogMediator text = ** Reporting to the Database **
...
INFO LogMediator text = ** Looking up from the Database **
...
INFO LogMediator text = Stock price - 183.3635460215262

```

When you analyze the log, you will understand that the ESB logs the above output as it reads from and writes to the database.

### Sample 364: Using Mediators to Execute Database Stored Procedures

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)

- Executing the sample
- Analyzing the output

## Introduction

This sample demonstrates how you can use the **DBLookup Mediator** and **DBReport Mediator** to execute database stored procedures. Here the dblookup mediator and dbreport mediator are used to access the database with statements that call a stored procedure in MySQL.

## Prerequisites

- MySQL database installed and configured. For instructions on setting up the MySQL database, see [Setting up the MySQL database](#).
- For a list of general prerequisites, see [Prerequisites to Start the ESB Samples](#)

## Building the sample

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </in>
 <out>
 <log level="custom">
 <property name="text" value="*** Reporting to the Database ***"/>
 </log>
 <dbreport>
 <connection>
 <pool>
 <driver>com.mysql.jdbc.Driver</driver>
 <url>jdbc:mysql://localhost:3306/synapsesdb</url>
 <user>user</user>
 <password>password</password>
 </pool>
 </connection>
 <statement>
 <sql>call updateCompany(?,?)</sql>
 <parameter xmlns:m0="http://services.samples"
 xmlns:m1="http://services.samples/xsd"
 expression="//m0:return/m1:last/child::text() "
type="DOUBLE" />
 <parameter xmlns:m0="http://services.samples"
 xmlns:m1="http://services.samples/xsd"
 expression="//m0:return/m1:symbol/child::text() "
type="VARCHAR" />
 </statement>
 </dbreport>
 <log level="custom">
 <property name="text" value="*** Looking up from the Database ***"/>
 </log>
 <dblookup>
 <connection>
```

```
<pool>
 <driver>com.mysql.jdbc.Driver</driver>
 <url>jdbc:mysql://localhost:3306/synapseDB</url>
 <user>user</user>
 <password>password</password>
</pool>
</connection>
<statement>
 <sql>call getCompany(?)</sql>
 <parameter xmlns:m0="http://services.samples"
 xmlns:m1="http://services.samples/xsd"
 expression="//m0:return/m1:symbol/child::text()"
 type="VARCHAR" />
 <result name="stock_prize" column="price"/>
 </statement>
</dblookup>
<log level="custom">
 <property name="text"
 expression="fn:concat('Stock Prize -
',get-property('stock_prize'))" />
 </log>
<send/>
```

```

</out>
</sequence>
</definitions>

```

This configuration file `synapse_sample_364.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### To build the sample

1. Start the ESB with the sample 364 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

### Executing the sample

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

#### To execute the sample client

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```

ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=IBM

```

### Analyzing the output

When the client is run, you will see that the ESB invokes the two stored procedures and that the response is mediated back to the client.

If you analyze the debug log output on the ESB console, you will see the following output:

```

INFO LogMediator text = ** Looking up from the Database ** ...
INFO LogMediator text = Company ID - c1 ...
INFO LogMediator text = Stock price - 183.3635460215262

```

### Throttling Messages (Throttle Mediator)

The following samples demonstrate how to use the **throttle mediator**:

- [Sample 370: Introduction to Throttle Mediator and Concurrency Throttling](#)
- [Sample 371: Restricting Requests Based on Policies](#)
- [Sample 372: Use of Both Concurrency Throttling and Request-Rate-Based Throttling](#)

#### Sample 370: Introduction to Throttle Mediator and Concurrency Throttling

**Objective:** Demonstrate the use of throttle mediator for concurrency throttling.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <throttle id="A">
 <policy>
 <!-- define throttle policy -->
 <wsp:Policy
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:throttle="http://www.wso2.org/products/wso2commons/throttle">
 <throttle:ThrottleAssertion>

<throttle:MaximumConcurrentAccess>10</throttle:MaximumConcurrentAccess>
 </throttle:ThrottleAssertion>
 </wsp:Policy>
 </policy>
 <onAccept>
 <log level="custom">
 <property name="text" value="**Access Accept**"/>
 </log>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </onAccept>
 <onReject>
 <log level="custom">
 <property name="text" value="**Access Denied**"/>
 </log>
 <makefault>
 <code value="tns:Receiver"
 xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason value="**Access Denied**"/>
 </makefault>
 <property name="RESPONSE" value="true"/>
 <header name="To" action="remove"/>
 <send/>
 <drop/>
 </onReject>
 </throttle>
 </in>
 <out>
 <throttle id="A"/>
 <send/>
 </out>
 </sequence>
 </definitions>

```

### Prerequisites:

- Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000.
- Start ESB with the sample configuration 370 (i.e. wso2esb-samples -sn 370).

Above configuration specifies a throttle mediator inside the in mediator. Therefore, all request messages directed to the main sequence will be subjected to throttling. Throttle mediator has 'policy', 'onAccept' and 'onReject' tags at top

level. The 'policy' tag specifies the throttling policy for throttling messages. This sample policy only contains a component called "MaximumConcurrentAccess". This indicates the maximum number of concurrent requests that can pass through Synapse on a single unit of time. To test concurrency throttling, it is required to send concurrent requests to Synapse. If Synapse with above configuration, receives 20 requests concurrently from clients, then approximately half of those will succeed while the others being throttled. The client command to try this is as follows.

```
ant stockquote -Dsymbol=IBM -Dmode=quote -Daddurl=http://localhost:8280/
```

### Sample 371: Restricting Requests Based on Policies

**Objective:** Demonstrate the use of throttle mediator for restricting request counts

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <in>
 <throttle id="A">
 <policy>
 <!-- define throttle policy -->
 <wsp:Policy
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:throttle="http://www.wso2.org/products/wso2commons/throttle">
 <throttle:MediatorThrottleAssertion>
 <wsp:Policy>
 <throttle:ID throttle:type="IP">other</throttle:ID>
 <wsp:Policy>
 <throttle:Control>
 <wsp:Policy>

<throttle:MaximumCount>4</throttle:MaximumCount>

<throttle:UnitTime>800000</throttle:UnitTime>
 <throttle:ProhibitTimePeriod
wsp:Optional="true">1000
 </throttle:ProhibitTimePeriod>
 </wsp:Policy>
 </throttle:Control>
 </wsp:Policy>
 <wsp:Policy>
 <throttle:ID throttle:type="IP">10.100.1.160 -
10.100.1.165</throttle:ID>
 <wsp:Policy>
 <throttle:Control>
 <wsp:Policy>

<throttle:MaximumCount>5</throttle:MaximumCount>

<throttle:UnitTime>800000</throttle:UnitTime>
 <throttle:ProhibitTimePeriod
wsp:Optional="true">100000
 </throttle:ProhibitTimePeriod>
 </wsp:Policy>
 </throttle:Control>
 </wsp:Policy>
 </wsp:Policy>
 </throttle>
 </in>
 </sequence>
</definitions>
```

```
</throttle:MediatorThrottleAssertion>
</wsp:Policy>
</policy>
<onAccept>
 <log level="custom">
 <property name="text" value="**Access Accept**"/>
 </log>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </onAccept>
 <onReject>
 <log level="custom">
 <property name="text" value="**Access Denied**"/>
 </log>
 <makefault>
 <code value="tns:Receiver"
 xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason value="**Access Denied**"/>
 </makefault>
 <property name="RESPONSE" value="true"/>
 <header name="To" action="remove"/>
 <send/>
 <drop/>
 </onReject>
</throttle>
</in>
<out>
 <throttle id="A"/>
 <send/>
```

```

</out>
</sequence>
</definitions>

```

**Prerequisites:**

- Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000.
- Start ESB with the sample configuration 371 (i.e. wso2esb-samples -sn 371).

Above configuration specifies a throttle mediator inside the in mediator. Therefore, all request messages directed to the main sequence will be subjected to throttling. Throttle mediator has policy, onAccept and onReject tags at the top level. Policy tag specifies the throttling policy to be applied for messages. It contains some IP address ranges and the maximum number of messages to be allowed for those ranges within a time period given in "UnitTime" tag. "ProhibitTimePeriod" tag specifies the time period to prohibit further requests after the received request count exceeds the specified time. Now run the client 5 times repetitively using the following command to see how throttling works.

```
ant stockquote -Dsymbol=IBM -Dmode=quote -Daddurl=http://localhost:8280/
```

For the first four requests you will get the quote prices for IBM as follows.

```
[java] Standard :: Stock price = $177.20143371883802
```

You will receive the following response for the fifth request.

```
[java] org.apache.axis2.AxisFault: **Access Denied**
```

Maximum number of requests within 800000 milliseconds is specified as 4 for any server (including localhost) other than the explicitly specified ones. Therefore, our fifth request is denied by the throttle mediator. You can verify this by looking at the ESB console.

```

[HttpServerWorker-1] INFO LogMediator - text = **Access Accept**
[HttpServerWorker-2] INFO LogMediator - text = **Access Accept**
[HttpServerWorker-3] INFO LogMediator - text = **Access Accept**
[HttpServerWorker-4] INFO LogMediator - text = **Access Accept**
[HttpServerWorker-5] INFO LogMediator - text = **Access Denied**

```

**Sample 372: Use of Both Concurrency Throttling and Request-Rate-Based Throttling**

**Objective:** Use of both concurrency throttling and request rate based throttling.

```

<!-- Use of both concurrency throttling and request rate based throttling -->
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <registry provider="org.wso2.carbon.mediation.registry.ESBRegistry">
 <!-- the root property of the simple URL registry helps resolve a resource URL
as root + key -->
 <parameter name="root">file:repository/</parameter>
 <!-- all resources loaded from the URL registry would be cached for this
number of milli seconds -->
 <parameter name="cachableDuration">150000</parameter>
 </registry>

 <sequence name="onAcceptSequence">
 <log level="custom">
 <property name="text" value="**Access Accept**"/>
 </log>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </sequence>
 <sequence name="onRejectSequence" trace="enable">
 <log level="custom">
 <property name="text" value="**Access Denied**"/>
 </log>
 <makefault>
 <code value="tns:Receiver"
 xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason value="**Access Denied**"/>
 </makefault>
 <property name="RESPONSE" value="true"/>
 <header name="To" action="remove"/>
 <send/>
 <drop/>
 </sequence>
 <proxy name="StockQuoteProxy">
 <target>
 <inSequence>
 <throttle onReject="onRejectSequence" onAccept="onAcceptSequence"
id="A">
 <policy
key="repository/samples/resources/policy/throttle_policy.xml"/>
 </throttle>
 </inSequence>
 <outSequence>
 <throttle id="A"/>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 </proxy>
 </definitions>

```

## Prerequisites:

- Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000.
- Start ESB with the sample configuration 372 (i.e. wso2esb-samples -sn 372).

Throttle policy is loaded from the ?throttle\_policy.xml? .That policy contains merging policy from sample 370 and 371. To check the functionality , it is need to run load test. The all-enabled request from the concurrency throttling will be controlled by the access rate base throttling according to the policy.

Run the client as follows:

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
```

You will get results same as sample 371.if you run the load test, results will be different due to affect of concurrency throttling.

## ***Extending the mediation in java (Class Mediator)***

Class mediator can be used to write your own custom mediation in Java and you have access to the SynapseMessageContext and to the full Synapse API in there. This is a useful extension mechanism within ESB to extend its functionality. This class can contain fields for which you can assign values at runtime through the configuration.

## **Extending the Mediation in Java (Class Mediator)**

The following samples demonstrate how to use the [class mediator](#):

- Sample 380: Writing your own Custom Mediation in Java
- Sample 381: Class Mediator to CBR Binary Messages

### **Sample 380: Writing your own Custom Mediation in Java**

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

## **Introduction**

This sample demonstrate how you can write your own custom mediation in java by using the [Class mediator](#) to extend the mediation functionality.

## **Prerequisites**

- For a list of general prerequisites, see [Prerequisites to Start the ESB Samples](#).

## **Building the sample**

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">

 <sequence name="fault">
 <makefault>
 <code value="tns:Receiver"
xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason value="Mediation failed."/>
 </makefault>
 <send/>
 </sequence>

 <sequence name="main" onError="fault">
 <in>
 <send>
 <endpoint name="stockquote">
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </in>
 <out>
 <class name="samples.mediators.DiscountQuoteMediator">
 <property name="discountFactor" value="10"/>
 <property name="bonusFor" value="5"/>
 </class>
 <send/>
 </out>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_380.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 380 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

## Executing the sample

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### To execute the sample client

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory.

```
ant stockquote -Dsymbol=IBM -Dmode=quote -Daddurl=http://localhost:8280
```

## Analyzing the output

When the client is run, you will see the following discounted quote value on the client console.

```
[java] Standard :: Stock price = $138.77458254967408
```

When you have a look at the console running Synapse. You will see the original value and the discounted value for the requested quote as follows.

```
Quote value discounted.
Original price: 162.30945327447262
Discounted price: 138.77458254967408
```

According to `synapse_sample_380.xml`, ESB hands over the request message to the specified endpoint, which sends it to the Axis2 server running on port 9000. But the response message is passed through the class mediator before sending it back to the client. Two parameters named "discountFactor" and "bonusFor" are passed to the instance mediator implementation class (i.e. `samples.mediators.DiscountQuoteMediator`) before each invocation. Code of the mediator implementation class is given below.

```
package samples.mediators;

import javax.xml.namespace.QName;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.synapse.MessageContext;
import org.apache.synapse.mediators.AbstractMediator;

public class DiscountQuoteMediator extends AbstractMediator {

 private static final Log log = LogFactory.getLog(DiscountQuoteMediator.class);
 private String discountFactor = "10";
 private String bonusFor = "10";
 private int bonusCount = 0;
 public DiscountQuoteMediator() {}
 public boolean mediate(MessageContext mc) {

 String price = mc.getEnvelope().getBody().getFirstElement().getFirstElement().
 getFirstChildWithName(new QName("http://services.samples/xsd", "last")).getText();

 //converting String properties into integers
 int discount = Integer.parseInt(discountFactor);
 int bonusNo = Integer.parseInt(bonusFor);
 double currentPrice = Double.parseDouble(price);

 //discounting factor is deducted from current price form every response
 Double lastPrice = new Double(currentPrice - currentPrice * discount / 100);

 //Special discount of 5% offers for the first responses as set in the bonusFor
 //property
 if (bonusCount <= bonusNo) {
 lastPrice = new Double(lastPrice.doubleValue() - lastPrice.doubleValue() * 0.05);
 bonusCount++;
 }
 }
}
```

```
String discountedPrice = lastPrice.toString();

mc.getEnvelope().getBody().getFirstElement().getFirstElement().getFirstChildWithName
 (new QName("http://services.samples/xsd", "last")).setText(discountedPrice);

System.out.println("Quote value discounted.");
System.out.println("Original price: " + price);
System.out.println("Discounted price: " + discountedPrice);

return true;
}

public String getType() {
 return null;
}

public void setTraceState(int traceState) {
 traceState = 0;
}

public int getTraceState() {
 return 0;
}

public void setDiscountFactor(String discount) {
 discountFactor = discount;
}

public String getDiscountFactor() {
 return discountFactor;
}

public void setBonusFor(String bonus) {
 bonusFor = bonus;
}

public String getBonusFor() {
 return bonusFor;
}
```

```
 }
}
```

All classes developed for class mediation should implement the Mediator interface, which contains the mediate(...) method. mediate(...) method of the above class is invoked for each response message mediated through the main sequence, with the message context of the current message as the parameter. All the details of the message including the SOAP headers, SOAP body and properties of the context hierarchy can be accessed from the message context. In this sample, the body of the message is retrieved and the discount percentage is subtracted from the quote price. If the quote request number is less than the number specified in the "bonusFor" property in the configuration, a special discount is given.

### Sample 381: Class Mediator to CBR Binary Messages

**Objective:** Demonstrate on CBR a message with binary payload

```

<definitions xmlns="http://ws.apache.org/ns/synapse"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ws.apache.org/ns/synapse
http://synapse.apache.org/ns/2010/04/configuration/synapse_config.xsd">

 <proxy name="JMSBinaryProxy" transports="jms">
 <target inSequence="BINARY_CBR_SEQ"/>
 </proxy>

 <sequence name="BINARY_CBR_SEQ">
 <in>
 <log level="full"/>
 <property action="set" name="OUT_ONLY" value="true"/>
 <class name="samples.mediators.BinaryExtractMediator">
 <property name="offset" value="11"/>
 <property name="length" value="4"/>
 <property name="variableName" value="symbol"/>
 <property name="binaryEncoding" value="utf-8"/>
 </class>
 <log level="custom">
 <property name="symbol" expression="get-property('symbol')"/>
 </log>
 <switch source="get-property('symbol')">
 <case regex="GOOG">
 <send>
 <endpoint>
 <address
uri="jms:/dynamicTopics/mdd.GOOG?transport.jms.ConnectionFactoryJNDIName=TopicConnecti
onFactory&java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextF
actory&java.naming.provider.url=tcp://localhost:61616&transport.jms.DestinationType=to
pic"/>
 </endpoint>
 </send>
 </case>
 <case regex="MSFT">
 <send>
 <endpoint>
 <address
uri="jms:/dynamicTopics/mdd.MSFT?transport.jms.ConnectionFactoryJNDIName=TopicConnecti
onFactory&java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextF
actory&java.naming.provider.url=tcp://localhost:61616&transport.jms.DestinationType=to
pic"/>
 </endpoint>
 </send>
 </case>
 <default/>
 </switch>
 </in>
 </sequence>
 </definitions>

```

### Prerequisites:

- Make sure the synapse-samples-1.0.jar is in your class path (by default this jar is placed in the lib directory)

- when installing Synapse)
- Configure the JMS transport using ActiveMQ as described in [Setting Up the ESB Samples](#).
  - Start ESB with the sample configuration 381 (i.e. wso2esb-samples -sn 381)
  - Start the sample Axis2 server and deploy the SimpleStockQuoteService.

In this configuration, a proxy has configured to accept incoming JMS messages. JMS messages contains a binary payload. User configure the offset, length, binary encoding of the text literal that it need to use for CBR. And a variable name to set the decoded value as a property. Configuration simply route the messages based on the text to different endpoints.

A JMS producer and two instances of a consumer used to demonstrate the CBR functionality.

Now run the first consumer using the following command.

```
ant mddconsumer -Djms_topic=mdd.MSFT
```

Now run the second consumer using the following command.

```
ant mddconsumer -Djms_topic=mdd.GOOG
```

So, now both consumers are ready to listen the topic. Run the market data producer to generate market data for symbol 'MSFT' using the following command.

```
ant mddproducer -Dsymbol=MSFT
```

Now run the market data producer to generate market data for symbol 'GOOG' using the following command.

```
ant mddproducer -Dsymbol=GOOG
```

You will see the below output in the client console(s) based on the symbol.

```
mddconsumer:
[java] Market data received for symbol : topic://mdd.MSFT
[java] Market data received for symbol : topic://mdd.MSFT
```

## Evaluating XQuery for Mediation (XQuery Mediator)

The following samples demonstrate how to use the [XQuery mediator](#):

- [Sample 390: Introduction to XQuery Mediator](#)
- [Sample 391: Using Data from an External XML Document within XQuery](#)

### Sample 390: Introduction to XQuery Mediator

This example uses the XQuery mediator to perform transformations and behaves the same as [Sample 8](#). The only difference is that this sample uses XQuery instead of XSLT for transformation.

```

<!-- Introduction to the XQuery mediator -->
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <!-- the SimpleURLRegistry allows access to a URL based registry (e.g. file:/// or
 http://) -->
 <registry provider="org.wso2.esb.registry.ESBRegistry">
 <!-- the root property of the simple URL registry helps resolve a resource URL
 as root + key -->
 <parameter name="root">file:repository/samples/resources/</parameter>
 <!-- all resources loaded from the URL registry would be cached for this
 number of milli seconds -->
 <parameter name="cachableDuration">15000</parameter>
 </registry>

 <localEntry key="xquery-key-req"
 src="file:repository/samples/resources/xquery/xquery_req.xq" />
<proxy name="StockQuoteProxy">
 <target>
 <inSequence>
 <xquery key="xquery-key-req">
 <variable name="payload" type="ELEMENT" />
 </xquery>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <out>
 <xquery key="xquery/xquery_res.xq">
 <variable name="payload" type="ELEMENT" />
 <variable name="code" type="STRING"

expression="self::node()//m0:return/m0:symbol/child::text()"
 xmlns:m0="http://services.samples/xsd" />
 <variable name="price" type="DOUBLE"

expression="self::node()//m0:return/m0:last/child::text()"
 xmlns:m0="http://services.samples/xsd" />
 </xquery>
 <send/>
 </out>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl" />
 </proxy>
</definitions>

```

Execute the custom quote client as 'ant stockquote -Dmode=customquote ...'

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
-Dmode=customquote
```

### Sample 391: Using Data from an External XML Document within XQuery

**Objective:** Demonstrate the use of XQuery mediator to import external XML documents to the XQuery engine

```
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <!-- the SimpleURLRegistry allows access to URL based registry (e.g. file:/// or
 http://) -->
 <registry provider="org.wso2.carbon.mediation.registry.ESBRegistry">
 <!-- the root property of the simple URL registry helps resolve a resource URL
 as root + key -->
 <parameter name="root">file:repository/samples/resources/</parameter>
 <!-- all resources loaded from the URL registry would be cached for this
 number of milli seconds -->
 <parameter name="cachableDuration">15000</parameter>
 </registry>

 <proxy name="StockQuoteProxy">
 <target>
 <inSequence>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <out>
 <xquery key="xquery/xquery_commission.xq">
 <variable name="payload" type="ELEMENT"></variable>
 <variable name="commission" type="ELEMENT"
key="misc/commission.xml"></variable>
 </xquery>
 <send/>
 </out>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl" />
 </proxy>
</definitions>
```

#### Prerequisites:

- Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000.
- Start ESB with the sample configuration 391 (i.e. wso2esb-samples -sn 391).

In this sample, data from commission.xml document is used inside XQUERY document. The stock quote price from the response and commission from the commission.xml document will be added and given as a new price .

Invoke the client as follows.

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
```

## Splitting Messages into Parts and Processing in Parallel (Iterate/Aggregate)

The following sample demonstrates how to use the [Iterate](#) and [Aggregate](#) mediators:

- Sample 400: Message Splitting and Aggregating the Responses

### Sample 400: Message Splitting and Aggregating the Responses

**Objective:** Demonstrate the use of Iterate mediator to split the messages in to parts and process them asynchronously and then aggregate the responses coming in to ESB.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <proxy name="SplitAggregateProxy">
 <target>
 <inSequence>
 <iterate expression="//m0:getQuote/m0:request" preservePayload="true"
 attachPath="//m0:getQuote"
 xmlns:m0="http://services.samples">
 <target>
 <sequence>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </sequence>
 </target>
 </iterate>
 </inSequence>
 <outSequence>
 <aggregate>
 <onComplete expression="//m0:getQuoteResponse"
 xmlns:m0="http://services.samples">
 <send/>
 </onComplete>
 </aggregate>
 </outSequence>
 </target>
 </proxy>
</definitions>
```

#### Prerequisites:

- Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000.
- Start ESB with the sample configuration 400 (i.e. wso2esb-samples -sn 400).

In this sample, the message sent to ESB has embedded with a number of elements of the same type in one message. When ESB received this message it will iterate through those elements and then sent to the specified endpoint. When all the responses appear in to ESB then those messages will be aggregated to form the resultant response and sent back to the client.

Invoke the client as follows.

```
ant stockquote -Daddurl=http://localhost:8280/services/SplitAggregateProxy -Ditr=4
```

## Caching Responses Over Requests (Cache Mediator)

The following samples demonstrate how to use the [cache mediator](#):

- [Sample 420: Simple Cache Implemented on ESB for the Actual Service](#)

### Sample 420: Simple Cache Implemented on ESB for the Actual Service

**Objective:** Demonstrate the use of Cache mediator in order to cache the response and use that cached response as the response for an identical xml request.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <in>
 <cache timeout="20" scope="per-host" collector="false"
 hashGenerator="org.wso2.carbon.mediator.cache.digest.DOMHASHGenerator">
 <implementation type="memory" maxSize="100"/>
 </cache>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </in>
 <out>
 <cache collector="true" />
 <send/>
 </out>
</definitions>
```

#### Prerequisites:

- Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000.
- Start ESB with the sample configuration 420 (i.e. wso2esb-samples -sn 420).

In this sample, the message sent to ESB is checked for an existing cached response by calculating the hash value of the request. If there is a cache hit in ESB then this request will not be forwarded to the actual service, rather ESB respond to the client with the cached response. In case of a cache miss that particular message will be forwarded to the actual service and cached that response in the out path for the use of consecutive requests of the same type.

To observe this behavior, invoke the client as follows.

```
ant stockquote -Dtrpurl=http://localhost:8280/
```

You could notice that if you send more than one requests within 20 seconds only the first request is forwarded to the actual service, and the rest of the requests will be served by the cache inside ESB. You could observe this by looking at the printed line of the axis2 server, as well as by observing a constant rate as the response to the client instead of the random rate, which changes by each and every 20 seconds.

### Synchronize web service invocation with Callout mediator

The Callout mediator calls the given service URL with the request message which is given by the source attribute, waits for the response and attaches the received response to the destination which is given by the target attribute. Both the source and the target can be a key or an XPath. In the case of the source, this key refers to either a message context property or to a local entry. For the target, this key refers to a message context property only.

## Mediating JSON Messages

The following samples demonstrate how to mediate JSON messages:

- [Sample 440: Converting JSON to XML Using XSLT](#)
- [Sample 441: Converting JSON to XML Using JavaScript](#)

For more information on using JSON with the ESB, see [JSON Support](#).

### Sample 440: Converting JSON to XML Using XSLT

**Objective:** Demonstrate the ability to expose a SOAP service over JSON by switching between JSON and XML/SOAP message formats using the [XSLT mediator](#) and [Enrich Mediator](#).

```
<definitions xmlns="http://ws.apache.org/ns/synapse"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ws.apache.org/ns/synapse
http://synapse.apache.org/ns/2010/04/configuration/synapse_config.xsd">

 <proxy name="JSONProxy" transports="http https">
 <target>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"
format="soap11"/>
 </endpoint>
 <inSequence>
 <log level="full"/>
 <xslt key="in_transform"/>
 <property name="messageType" scope="axis2" value="text/xml"/>
 <header name="Action" scope="default" value="urn:getQuote"/>
 <enrich>
 <source xmlns:m0="http://services.samples" clone="true"
xpath="//m0:getQuote"/>
 <target type="body"/>
 </enrich>
 </inSequence>
 <outSequence>
 <log level="full"/>
 <xslt key="out_transform"/>
 <property name="messageType" scope="axis2" value="application/json"/>
 <send/>
 </outSequence>
 </target>
 </proxy>

 <localEntry key="in_transform">
 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:fn="http://www.w3.org/2005/02/xpath-functions"
 xmlns:m0="http://services.samples" version="2.0"
exclude-result-prefixes="m0 fn">
 <xsl:output method="xml" omit-xml-declaration="yes" indent="yes"/>
 <xsl:template match="*">
 <xsl:element name="{local-name()}"
namespace="http://services.samples">
 <xsl:copy-of select="attribute::*"/>

```

```
 <xsl:apply-templates/>
 </xsl:element>
</xsl:template>
</xsl:stylesheet>
</localEntry>

<localEntry key="out_transform">
 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
 <xsl:output method="xml" version="1.0" encoding="UTF-8" />
 <xsl:template match="*">
 <xsl:element name="{local-name()}">
 <xsl:apply-templates/>
 </xsl:element>
 </xsl:template>
 </xsl:stylesheet>
```

```

</localEntry>

</definitions>

```

1. Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000.
2. Start Synapse with the sample configuration 440 (i.e. wso2esb-samples -sn 440).
3. Setup Synapse and the sample Axis2 client for JSON (Refer Synapse Samples Setup Guide for details)

Use following command to invoke the proxy service:

```
ant jsonclient -Daddurl=http://localhost:8280/services/JSONProxy
```

jsonclient sends the following JSON request to the above proxy service:

```
{"getQuote": {"request": {"symbol": "WSO2"}}}
```

This request gets the following XML representation when it gets built in the ESB: (Take a look at the [XML representation of JSON payloads](#) within the ESB)

```

<j(jsonObject>
 <getQuote>
 <request>
 <symbol>WSO2</symbol>
 </request>
 </getQuote>
</j(jsonObject>

```

In the `inSequence`, the XSLT transformation `in_transform` transforms the above XML representation to the following XML by applying namespace declarations that are expected by the service.

```

<j(jsonObject xmlns="http://services.samples">
 <getQuote>
 <request>
 <symbol>WSO2</symbol>
 </request>
 </getQuote>
</j(jsonObject>

```

Finally the **Enrich Mediator** extracts the `getQuote` element from the above payload and attaches it as the first child of the current payload so that the final SOAP request that is sent to the service has the following format.

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
 <soapenv:Body>
 <getQuote xmlns="http://services.samples">
 <request>
 <symbol>WSO2</symbol>
 </request>
 </getQuote>
 </soapenv:Body>
</soapenv:Envelope>

```

We need to perform this sequence of transformations to the incoming JSON request to convert it and build the XML SOAP request expected by the service. Note that the XSLT transformation in the outSequence (line 22) can be omitted because the JSON message formatter already knows how to format any XML payload to JSON.

The final response will look like the following:

```
{
 "getQuoteResponse": {
 "return": {
 "change": 3.853593376681722,
 "earnings": 12.802850763714854,
 "high": 67.92488310190126,
 "last": 66.14619264746406,
 "lastTradeTimestamp": "Mon Aug 23 16:48:40 IST 2010",
 "low": -66.04000424423522,
 "marketCap": -9334516.42324327,
 "name": "WSO2 Company",
 "open": -64.61950137150009,
 "peRatio": -19.78600441437058,
 "percentageChange": 5.411779328273005,
 "prevClose": 71.2075112994578,
 "symbol": "WSO2",
 "volume": 16842
 }
 }
}
```

For more information about using JSON with the ESB, see [JSON Support](#). For an example of how to convert JSON to XML using using JavaScript instead of XSLT, see [Sample 441: Converting JSON to XML Using JavaScript](#).

### Sample 441: Converting JSON to XML Using JavaScript

**Objective:** Demonstrate the ability to expose a SOAP service over JSON by switching between JSON and XML/SOAP message formats using JavaScript and the Script mediator.

### Sample 441 configuration

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="JSONProxy" transports="http https">
 <target>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"
format="soap11"/>
 </endpoint>
 <inSequence>
 <!-- transform the custom quote request into a standard quote request
expected by the service -->
 <script language="js"><![CDATA[
 var symbol = mc.getPayloadJSON().symbol.toString();
 mc.setPayloadXML(
 <m:getQuote xmlns:m="http://services.samples">
 <m:request>
 <m:symbol>{symbol}</m:symbol>
 </m:request>
 </m:getQuote>);
]]></script>
 <header name="Action" value="urn:getQuote"/>
 </inSequence>
 <outSequence>
 <script language="js"><![CDATA[
 var symbol = mc.getPayloadXML()...*::symbol.toString();
 var price = parseFloat(mc.getPayloadXML()...*::last);
 mc.setPayloadJSON(
 {
 "Quote" : {
 "Code" : symbol,
 "Price" : price,
 "Current" : {
 "High" : parseFloat(mc.getPayloadXML()...*::high),
 "Last" : parseFloat(mc.getPayloadXML()...*::last)
 }
 },
 "Status" : (price >= 100 ? "OK" : "NOT-OK")
 });
]]></script>
 <property name="messageType" scope="axis2" value="application/json"/>
 <send/>
 </outSequence>
 </target>
 </proxy>
</definitions>

```

The JavaScript specified by the Script mediator found in the inSequence performs the JSON to SOAP (XML) transformation as the SimpleStockQuoteService endpoint accepts SOAP. Similarly, the Script mediator in the outSequence performs the SOAP to JSON transformation.

1. Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000.
2. Start WSO2 ESB with the sample configuration 441 (i.e., `wso2esb-samples -sn 441`).

Invoke the above proxy service with following request:

```
curl -v -X POST \
-H "Content-Type:application/json" \
-d '{"symbol":"WSO2", "ID":"StockQuote"}' \
"http://localhost:8280/services/JSONProxy"
```

If the invocation is successful, you will see a response similar to the following:

```
{
 "Quote": {
 "Current": {
 "High": -61.53602401623976,
 "Last": 62.32682938796667
 },
 "Code": "WSO2",
 "Price": 62.32682938796667
 },
 "Status": "NOT-OK"
}
```

For more information about using JSON with the ESB, see [JSON Support](#). To see an example of converting JSON to XML using XSLT instead of JavaScript, see [Sample 440: Converting JSON to XML Using XSLT](#). For additional examples of the Script mediator, see [Using Scripts in Mediation \(Script Mediator\)](#).

## Rewriting the URL (URL Rewrite Mediator)

The following samples demonstrate how to use the [URLRewrite Mediator](#):

- [Sample 450: Introduction to the URL Rewrite Mediator](#)
- [Sample 451: Conditional URL Rewriting](#)
- [Sample 452: Conditional URL Rewriting with Multiple Rules](#)

### Sample 450: Introduction to the URL Rewrite Mediator

**Objective:** Demonstrate the basic functions of the [URL Rewrite Mediator](#).

```
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <sequence name="main">
 <in>
 <rewrite>
 <rewriterule>
 <action type="replace" regex="soap" value="services"
fragment="path"/>
 </rewriterule>
 </rewrite>
 <send/>
 </in>
 <out>
 <send/>
 </out>
 </sequence>

</definitions>
```

**Prerequisites:**

- Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000.
- Start ESB with the sample configuration 450 (i.e. wso2esb-samples -sn 450).

Invoke the client as follows.

```
ant stockquote -Dtrpurl=http://localhost:8280
-Daddurl=http://localhost:9000/soap/SimpleStockQuoteService
```

Note that the address URL of the client request contains the context 'soap'. But in the Axis2 server all the services are deployed under a context named 'services' by default. ESB will rewrite the To header of the request by replacing the 'soap' context with 'services'. Hence the request will be delivered to the Axis2 server and the Axis2 client will receive a valid response.

**Sample 451: Conditional URL Rewriting**

**Objective:** Demonstrate the ability of the [URL Rewrite mediator](#) to evaluate conditions on messages and perform rewrites based on the results.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <sequence name="main">
 <in>
 <rewrite>
 <rewriterule>
 <condition>
 <and>
 <equal type="url" source="host" value="localhost"/>
 <not>
 <equal type="url" source="protocol" value="https"/>
 </not>
 </and>
 </condition>
 <action fragment="protocol" value="https"/>
 <action fragment="port" value="9002"/>
 </rewriterule>
 </rewrite>
 <send/>
 </in>
 <out>
 <send/>
 </out>
 </sequence>

</definitions>
```

**Prerequisites:**

- Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000.
- Start ESB with the sample configuration 451 (i.e. wso2esb-samples -sn 451).

Invoke the Axis2 client and send some requests to ESB with different address URL values. If the address URL value contains localhost as the hostname and https as the protocol prefix, ESB will route the message as it is. But if the hostname is localhost and the protocol is not https, Synapse will rewrite the URL by setting https as the protocol. The port number will also be set to the HTTPS port of the Axis2 server.

The condition evaluation feature is provided by the Synapse evaluator framework. Currently one can evaluate

expressions on URL values, query parameters, transport headers, properties and SOAP envelope content using this framework. Hence URL rewriting can be done based on any of these aspects.

### Sample 452: Conditional URL Rewriting with Multiple Rules

**Objective:** Demonstrate the ability of the [URL Rewrite mediator](#) to perform rewrites based on multiple rules.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <sequence name="main">
 <in>
 <property name="http.port" value="9000"/>
 <property name="https.port" value="9002"/>
 <rewrite>
 <rewriterule>
 <action fragment="host" value="localhost"/>
 <action fragment="path" type="prepend" value="/services"/>
 </rewriterule>
 <rewriterule>
 <condition>
 <equal type="url" source="protocol" value="http"/>
 </condition>
 <action fragment="port" xpath="get-property('http.port')"/>
 </rewriterule>
 <rewriterule>
 <condition>
 <equal type="url" source="protocol" value="https"/>
 </condition>
 <action fragment="port" xpath="get-property('https.port')"/>
 </rewriterule>
 </rewrite>
 <log level="full"/>
 <send/>
 </in>
 <out>
 <send/>
 </out>
 </sequence>

</definitions>
```

#### Prerequisites:

- Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000.
- Start ESB with the sample configuration 452 (i.e. wso2esb-samples -sn 452).

Invoke the Axis2 client as follows.

```
ant stockquote -Dtrpurl=http://localhost:8280
-Daddurl=http://test.com/SimpleStockQuoteService
```

The provided address URL does not contain a port number and the context. The URL rewrite mediator will replace the hostname to be 'localhost' and add the context '/services' to the path. Then it will add the appropriate port number to the URL by looking at the protocol prefix. Ultimately the service request will be routed to the sample Axis2 server and the client will receive a valid response. Note that the Synapse configuration does not specify any endpoints explicitly. So the messages are sent to the rewritten To header.

Another important aspect shown by this sample is the ability of the URL rewrite mediator to obtain the necessary values by executing XPath expressions. The port numbers are calculated by executing an XPath on the messages.

### **Publishing Events to Topics Using the Event Mediator**

WSO2 ESB can be used as an event broker. It comes with a built-in eventing implementation and a lightweight event broker based on Apache Qpid. You can use the ESB management console to create event topics and clients can subscribe to those topics by sending WS-Eventing subscription requests. The management console also allows creating static subscription.

WSO2 ESB is also equipped with an event mediator which can be used to publish messages to predefined topics. With this mediator it is possible for a sequence or a proxy service to directly publish a received request or a response to a topic as an event.

### **Eventing (Event Mediator)**

The following samples demonstrate how to use the [event mediator](#):

- [Sample 460: Introduction to Eventing and Event Mediator](#)

#### **Sample 460: Introduction to Eventing and Event Mediator**

Objective: Demonstrate the usage of the [Event Mediator](#) to publish messages to [event topics](#).

##### **Prerequisites**

- Deploy the [SimpleStockQuoteService](#) in sample Axis2 server and start it on port 9000.
- Start ESB with the sample configuration: e.g., `wso2esb-samples.sh -sn 460`

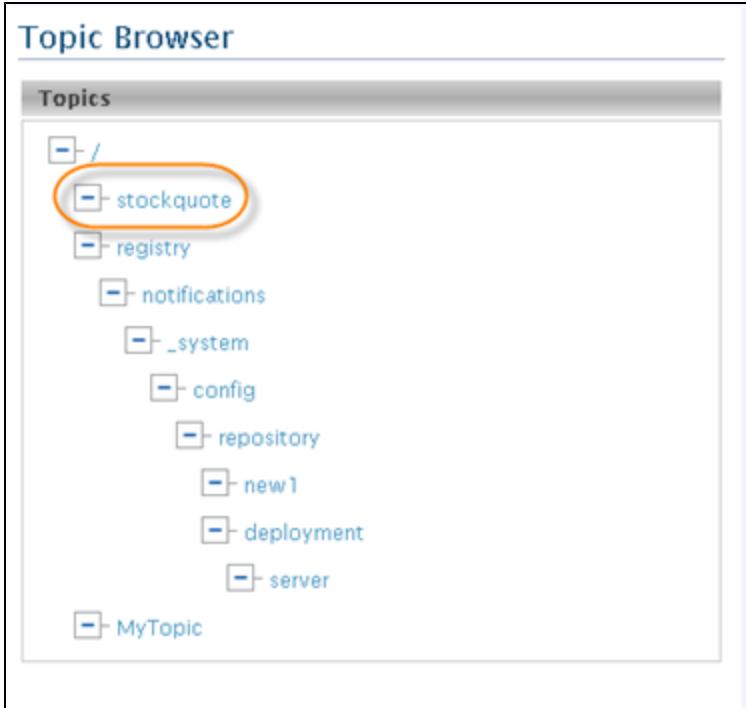
```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="main">
 <log/>
 <event topic="stockquote" />
 </sequence>
</definitions>
```

1) Add an event topic. See [Adding a New Topic](#).

1.1) Enter the name "stockquote" for the topic and click "Add Topic."

Add Topic		
Add Topic		
Topic	stockquote	
Permissions		
Role	Subscribe	Publish
everyone	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<b>Add Topic</b>		

1.2) This will create an event topic named "stockquote" and you will be directed to the "Topic Browser" tree view. The newly created topic will be shown in the tree.



2) Create a subscription. See [Subscribing to a Topic](#).

2.1) Enter the value "http://localhost:9000/services/SimpleStockQuoteService" in the "Event Sink URL" field and click "Subscribe."

The screenshot shows the 'Subscribe' dialog. It has a header 'Enter Subscription Details'. Under 'Topic\*', 'stockquote' is selected. Under 'Subscription Mode :\*', 'Topic Only' is chosen. The 'Event Sink URL\*' field contains 'http://localhost:9000/services/SimpleStockQuoteService' and is circled in orange. Below it, there are fields for 'Date:' and 'Time:', and a 'Expiration Time' section with a date picker and time inputs. At the bottom, there are 'Subscribe' and 'Cancel' buttons, both of which are circled in orange.

3) Now run the sample client as follows to send a request to the main sequence.

```
ant stockquote -Dtrpurl=http://localhost:8280 -Dmode=placeorder
```

The request will be published to the "stockquote" topic by the Event Mediator and as a result the subscriber (Axis2 server in this case) will receive a copy of the message. You will see a log entry in the Axis2 server console indicating the receipt of the place order request.

### Note

The provided ESB configuration does not explicitly specify the endpoint of the Axis2 server. Also we don't set the actual EPR of the service on the request when sending the message from the client either. Therefore the only reason that Axis2 receives the message is because it is subscribed to the "stockquote" event topic.

## Mediating with Spring

The following samples demonstrate how to use a Spring bean as a mediator:

- [Sample 470: How to Initialize and use a Spring Bean as a Mediator](#)

### Sample 470: How to Initialize and use a Spring Bean as a Mediator

**Objective:** Demonstrate how to initialize and use a SpringBean as a mediator.

In your Synapse configuration file, add the Spring bean as shown below:

```
<definitions xmlns="http://ws.apache.org/ns/synapse"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ws.apache.org/ns/synapse
http://synapse.apache.org/ns/2010/04/configuration/synapse_config.xsd">

 <registry provider="org.apache.synapse.registry.url.SimpleURLRegistry">
 <parameter name="root">file:repository/conf/sample/resources/</parameter>
 <parameter name="cachableDuration">15000</parameter>
 </registry>

 <sequence name="main">
 <!--Setting the Spring Mediator and its Spring Beans xml file location -->
 <!--Note that springtest is the bean id used in springCustomLogger.xml -->
 <spring bean="springtest" key="spring/springCustomLogger.xml"/>
 <send/>
 </sequence>

</definitions>
```

In the `springCustomLogger.xml` file, define the Spring bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC " -//SPRING//DTD BEAN//EN "
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

 <bean id="springtest" class="samples.mediators.extensions.SpringCustomLogger"
singleton="false">
 <property name="username"><value>"Synapse User"</value></property>
 <property name="email"><value>"usr@synapse.org"</value></property>
 </bean>

</beans>
```

#### Prerequisites:

- Enable logging for the `samples.mediators` package by adding the following line to the `<ESB_HOME>/repository/conf/log4j.properties` file:  
`log4j.category.samples.mediators=INFO`
- Deploy the `SimpleStockQuoteService` in the sample Axis2 server and start it on port 9000.
- Start the ESB with sample configuration 470 as follows: `wso2esb-samples -sn 470`

In this sample, the Spring Bean `SpringCustomLogger` gets initialized using the `springCustomLogger.xml` file and then logs the message ID. Invoke the client as follows:

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/
```

If you have successfully enabled logging for the `samples.mediators` package, you will see an output similar to the following on the console:

```
2010-09-26 20:46:57,946 [-] [HttpServerWorker-1] INFO SpringCustomLogger Starting
Spring Meditor
2010-09-26 20:46:57,946 [-] [HttpServerWorker-1] INFO SpringCustomLogger Bean in
Initialized with User:["Synapse User"]
2010-09-26 20:46:57,946 [-] [HttpServerWorker-1] INFO SpringCustomLogger
E-MAIL:["usr@synapse.org"]
2010-09-26 20:46:57,946 [-] [HttpServerWorker-1] INFO SpringCustomLogger Massage
Id: urn:uuid:383FA8B27D7CC549D91285514217720
2010-09-26 20:46:57,946 [-] [HttpServerWorker-1] INFO SpringCustomLogger
Logged....
```

## Invoking Web Services

The following samples demonstrate how to invoke web services using the [callout mediator](#) and [call mediator](#):

- [Sample 430: Callout Mediator for Synchronous Service Invocation](#)
- [Sample 500: Call Mediator for Non-Blocking Service Invocation](#)

### Sample 430: Callout Mediator for Synchronous Service Invocation

**Objective:** Demonstrate the use of the [Callout mediator](#) for synchronous web service invocations.

```
<!-- Simple callout mediator -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <callout serviceURL="http://localhost:9000/services/SimpleStockQuoteService"
 action="urn:getQuote">
 <source xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
 xpath="s11:Body/child::*[fn:position()=1] |
s12:Body/child::*[fn:position()=1]"/>
 <target xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
 xpath="s11:Body/child::*[fn:position()=1] |
s12:Body/child::*[fn:position()=1]"/>
 </callout>
 <property name="RESPONSE" value="true"/>
 <header name="To" action="remove"/>
 <send/>
 <drop/>
</definitions>
```

#### Prerequisites:

- Deploy the `SimpleStockQuoteService` in sample Axis2 server and start it on port 9000.
- Start Synapse with the sample configuration 430 (i.e. `wso2esb-samples -sn 430`).

In this sample, the Callout mediator does the direct service invocation to the StockQuoteService using the client request, gets the response, and sets the response as the first child of the SOAP message body. Then, using the [Send mediator](#), the message is sent back to the client.

Invoke the client as follows.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/
```

## Sample 500: Call Mediator for Non-Blocking Service Invocation

**Objective:** Demonstrate the use of the [Call mediator](#) for non-blocking web service invocations. It also demonstrates how we can use the Call mediator to implement service chaining scenarios.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <proxy name="StockQuoteProxy"
 transports="https http"
 startOnLoad="true"
 trace="disable">
 <target>
 <inSequence>
 <call>
 <endpoint key="StockQuoteService"/>
 </call>
 <header name="Action" value="urn:getQuote"/>
 <payloadFactory>
 <format>
 <m0:getName xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>WSO2</m0:symbol>
 </m0:request>
 </m0:getName>
 </format>
 <args/>
 </payloadFactory>
 <call>
 <endpoint key="StockQuoteService"/>
 </call>
 <respond/>
 </inSequence>
 </target>
 </proxy>

 <endpoint name="StockQuoteService">
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>

</definitions>
```

### Prerequisites:

- Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000.
- Start Synapse with the sample configuration 500 (wso2esb-samples -sn 500).

In this sample, Call mediator does a direct service invocation to the StockQuoteService using the client request.

After getting the response, a new payload is constructed using the Payload Factory mediator, and we invoke the StockQuoteService using the Call mediator again. Then, using the [Respond mediator](#), the message is sent back to the client.

Invoke the client as follows.

```
ant stockquote -Dtrpurl=http://localhost:8280/services/StockQuoteProxy
```

## Introduction to Rule Mediator

The following samples are explained:

- [Sample 600 : Simple Message Transformation - Rule Mediator for Message Transformation](#)
- [Sample 601 : Advance Rule Based Routing - Switching Routing Decision According to the Rules - Rule Mediator as Switch mediator](#)

### Sample 600 : Simple Message Transformation - Rule Mediator for Message Transformation

**Objective:** Simple message transformation - Rule mediator for message transformation.

```

<!-- Simple rule based transformation (changing message) -->
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <sequence name="main">
 <in>

 <rule xmlns="http://wso2.org/carbon/rules">
 <source>soapBody</source>
 <target action="replace" xmlns:m0="http://services.samples"
resultXpath="//m0:symbol"
xpath="//m0:getQuote/m0:request/m0:symbol">soapBody</target>
 <ruleSet>
 <properties/>
 <rule resourceType="regular" sourceType="inline">
 <![CDATA[package SimpleRoutingRules;

rule InvokeIBM
when
symbol: String()
eval(symbol.equals("MSFT") || symbol.equals("SUN"))
then
update(drools.getWorkingMemory().getFactHandle(symbol), "IBM");
end
]]
 </rule>
 </ruleSet>
 <input namespace="http://services.samples"
wrapperElementName="getQuote">
 <fact xmlns:m0="http://services.samples" elementName="symbol"
namespace="http://services.samples"
type="java.lang.String"
xpath="//m0:getQuote/m0:request/m0:symbol/child::text()"/>
 </input>
 <output namespace="http://services.samples"
wrapperElementName="getQuote">
 <fact elementName="symbol" namespace="http://services.samples"
type="java.lang.String"/>
 </output>
 </rule>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>

 </in>
 <out>
 <send/>
 </out>
 </sequence>

</definitions>

```

## Prerequisites:

- Start the Synapse configuration numbered 600: (i.e. ./wso2esb-samples.sh -sn 600)

- Start the Axis2 server and deploy the SimpleStockQuoteService if not already deployed
- In this sample , a simple transformation is happened . If the symbol is either SUN or MSFT , then it will changed into IBM in the SOAP envelope and then invoke the external service.

Run the client as

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=MSFT
```

Or as

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=SUN
```

Then check the axis2server log or console

```
Wed July 04 16:33:05 IST 2012 samples.services.SimpleStockQuoteService :: Generating
quote for : IBM
```

## **Sample 601 : Advance Rule Based Routing - Switching Routing Decision According to the Rules - Rule Mediator as Switch mediator**

**Objective:** Rule based routing - Switching routing decision according to the rules.

```
<!-- Simple rule based routing of messages - same as filter mediator -->
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <sequence name="main">

 <in>
 <rule xmlns="http://wso2.org/carbon/rules">
 <source>soapBody</source>
 <target action="replace"
resultXpath="//accept/child::text()">$accept</target>
 <ruleSet>
 <properties/>
 <rule resourceType="regular" sourceType="inline">
 <![CDATA[package SimpleRoutingRules;

 rule "Invoke IBM" no-loop true
 when
 symbol: String()eval(symbol.equals("IBM"))
 then

 update(drools.getWorkingMemory().getFactHandle(symbol), "ibmEndPoint");
 end

 rule "Invoke SUN" no-loop true
 when
 symbol: String()eval(symbol.equals("SUN"))
 then

 update(drools.getWorkingMemory().getFactHandle(symbol), "sunEndPoint");
]]>
 </ruleSet>
 </rule>
 </in>
 </sequence>
</definitions>
```

```

 end

 rule "Invoke MFST" no-loop true
 when
 symbol: String()eval(symbol.equals("MFST"))
 then

update(drools.getWorkingMemory().getFactHandle(symbol), "mfstEndPoint");
 end
]]>
</rule>
</ruleSet>
<input namespace="http://services.samples"
wrapperElementName="getQuote">
 <fact xmlns:m0="http://services.samples" elementName="symbol"
namespace="http://services.samples"
 type="java.lang.String"
xpath="/m0:getQuote/m0:request/m0:symbol/child::text()"/>
</input>
<output namespace="http://services.samples"
wrapperElementName="getQuoteResponse">
 <fact elementName="accept" namespace="" type="java.lang.String"/>
</output>
</rule>

<switch source="get-property('accept')">
 <case regex="ibmEndPoint">
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </case>
 <case regex="sunEndPoint">
 <sequence key="nonExistentService"/>
 </case>
 <case regex="mfstEndPoint">
 <sequence key="nonExistentService"/>
 </case>
 </switch>

 <drop/>

 </in>
 <out>
 <send/>
 </out>

</sequence>

<sequence name="nonExistentService" onError="myFaultHandler">

 <send>
 <endpoint>
 <address
uri="http://localhost:9009/services/NonExistentStockQuoteService"/>
 </endpoint>

```

```
</send>
<drop/>

</sequence>
<sequence name="myFaultHandler">
 <makefault>
 <code value="tns:Receiver"
xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
 <reason expression="get-property('ERROR_MESSAGE')"/>
 </makefault>

 <property name="RESPONSE" value="true"/>
 <header name="To" expression="get-property('ReplyTo')"/>
 <send/>
```

```
</sequence>

</definitions>
```

**Prerequisites:**

- Start the Synapse configuration numbered 601: i.e. (i.e. ./wso2esb-samples.sh -sn 601)
- Start the Axis2 server and deploy the SimpleStockQuoteService if not already deployed
- In rule script , there are three cases each for 'IBM','SUN' and 'MSFT'. When condition is match , then corresponding rule will be got fire.

Invoke IBM rule by running client as,

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/
```

You will get stock quote price successfully

Then invoke SUN (or MSFT) rule by running client as,

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=SUN
```

Then will get,

```
<soapenv:Fault xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<faultcode>soapenv:Server</faultcode>
<faultstring>java.net.ConnectException: Connection refused
</faultstring>
<detail />
</soapenv:Fault>
```

## Miscellaneous Samples

The following miscellaneous samples are available with the WSO2 Enterprise Service Bus (ESB):

- Sample 650: File Hierarchy-Based Configuration Builder
- Sample 651: Using Synapse Observers
- Sample 652: Priority Based Message Mediation
- Sample 653: NHTTP Transport Priority Based Dispatching
- Sample 654: Smooks Mediator
- Sample 655: Message Relay - Basics Sample
- Sample 656: Message Relay - Builder Mediator
- Sample 657: Distributed Transaction Management

- Sample 658: Huge XML Message Processing with Smooks Mediator
- Sample 659: Huge EDI Message Processing with Smooks Mediator

## Sample 650: File Hierarchy-Based Configuration Builder

**Objective:** Demonstrate the ability to construct the Synapse configuration from a file hierarchy.

```

synapse_sample_650.xml
 |-- endpoints
 | '-- foo.xml
 |-- events
 | '-- event1.xml
 |-- local-entries
 | '-- bar.xml
 |-- proxy-services
 | |-- proxy1.xml
 | |-- proxy2.xml
 | '-- proxy3.xml
 |-- registry.xml
 |-- sequences
 | |-- custom-logger.xml
 | |-- fault.xml
 | '-- main.xml
 |-- synapse.xml
 '-- tasks
 '-- task1.xml

```

### Prerequisites:

- Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000.
- Start ESB with the sample configuration 650 (i.e. wso2esb-samples.sh -sn 650).

Go to the repository/samples directory and locate the subdirectory named synapse\_sample\_650.xml within it. When ESB is started with the sample configuration 650, ESB will load the configuration from this directory. You will find a number of subdirectories and a set of XML files in each of those directories. Synapse will parse all the XML files in this file hierarchy and construct the full Synapse configuration at startup. As a result when this sample is executed Synapse will start with four proxy services, several sequences, a task, an event source and some endpoint and local entry definitions.

The names of the subdirectories (eg: proxy-services, sequences, endpoints) are fixed and hence cannot be changed. Also the registry definition should go into a file named registry.xml which resides at the top level of the file hierarchy. It can also be specified in the synapse.xml file at top level. This synapse.xml file can include any item that can be normally defined in a synapse.xml file. The files which define proxy services, sequences, endpoints etc can have any name. These configuration files must have the .xml extension at the end of the name. Synapse will ignore any files which do not have the .xml extension.

None of the directories and files in the sample file hierarchy are mandatory. You can leave entire directories out if you do not need them. For example if your configuration does not contain any proxy services you can leave the subdirectory named proxy-services out.

To use this feature you should simply pass a path to an existing directory when starting the Synapse server. The SynapseServer class which is responsible for starting the server accepts a file path as an argument from where to load the configuration. Generally we pass the path to the synapse.xml file as the value of this argument. If you pass a directory path instead, Synapse configuration will be loaded from the specified directory. Note the following line on the console when Synapse is loading the configuration from a file hierarchy.

```
2009-08-04 14:14:42,489 [-] [main] INFO SynapseConfigurationBuilder Loaded Synapse configuration from the directory hierarchy at : /home/synapse/repository/conf/sample/synapse_sample_650.xml
```

This feature comes in handy when managing large Synapse configurations. It is easier to maintain a well structured file hierarchy than managing one large flat XML file.

## Sample 651: Using Synapse Observers

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### ***Introduction***

This sample demonstrates how you can use the `SynapseObserver` interface to monitor the Synapse configuration at runtime.

### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### ***Building the sample***

- To define the simple logging Synapse observer, open the `<ESB_HOME>/repository/conf/synapse.properties` file and add the following line:  
`synapse.observers=samples.userguide.SimpleLoggingObserver`
- To set the log level of the `samples.userguide` package to `INFO`, open the `<ESB_HOME>/repository/conf/log4j.properties` file and add the following line:  
`log4j.category.samples.userguide=INFO`

### ***Executing the sample***

Start the ESB with any sample configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

### ***Analyzing the output***

You will see that the `SimpleLoggingObserver` captures events that occur while constructing the Synapse configuration and logs them on the console as follows:

```

2009-08-06 14:30:24,578 [-] [main] INFO SimpleLoggingObserver Simple logging observer initialized...Capturing Synapse events...
2009-08-06 14:30:24,604 [-] [main] INFO SimpleLoggingObserver Endpoint : a3 was added to the Synapse configuration successfully
2009-08-06 14:30:24,605 [-] [main] INFO SimpleLoggingObserver Endpoint : a2 was added to the Synapse configuration successfully
2009-08-06 14:30:24,606 [-] [main] INFO SimpleLoggingObserver Endpoint : null was added to the Synapse configuration successfully
2009-08-06 14:30:24,611 [-] [main] INFO SimpleLoggingObserver Local entry : a1 was added to the Synapse configuration successfully
2009-08-06 14:30:24,649 [-] [main] INFO SimpleLoggingObserver Proxy service : StockQuoteProxy2 was added to the Synapse configuration successfully
2009-08-06 14:30:24,661 [-] [main] INFO SimpleLoggingObserver Proxy service : StockQuoteProxy1 was added to the Synapse configuration successfully
2009-08-06 14:30:24,664 [-] [main] INFO SimpleLoggingObserver Sequence : main was added to the Synapse configuration successfully
2009-08-06 14:30:24,701 [-] [main] INFO SimpleLoggingObserver Sequence : fault was added to the Synapse configuration successfully

```

The SimpleLoggingObserver is implemented as follows:

```

package samples.userguide;

import org.apache.synapse.config.AbstractSynapseObserver;

public class SimpleLoggingObserver extends AbstractSynapseObserver {

 public SimpleLoggingObserver() {
 super();
 log.info("Simple logging observer initialized...Capturing Synapse events...");
 }
}

```

It does not override any of the event handler implementations in the `AbstractSynapseObserver` class. The `AbstractSynapseObserver` logs all the received events by default.

### Sample 652: Priority Based Message Mediation

Objective: Demonstrate the priority-based mediation capability of ESB.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <priority-executor name="exec">
 <queues>
 <queue size="100" priority="1"/>
 <queue size="100" priority="10"/>
 </queues>
 </priority-executor>
 <proxy name="StockQuoteProxy">
 <target>
 <inSequence>
 <filter source="$trp:priority" regex="1">
 <then>
 <enqueue priority="1" sequence="priority_sequence"
executor="exec" />
 </then>
 <else>
 <enqueue priority="10" sequence="priority_sequence"
executor="exec" />
 </else>
 </filter>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl" />
 </proxy>
 <sequence name="priority_sequence">
 <log level="full"/>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </sequence>
</definitions>

```

#### Prerequisites

- Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000. Priority is applied only when ESB is loaded with enough messages to consume its core number of threads. So to observe the priority based mediation it is required to use a load testing tool like JMeter, SOAP UI or Apache bench.
- Start ESB with the sample configuration 652: `wso2esb-samples.sh -sn 652`

In this sample, client should send a HTTP header that specifies the priority of the message. This header name is "priority". This header is retrieved in the ESB configuration using the `$trp:priority` XPath expression. Then it is matched against the value 1. If it has the value 1, message is executed with priority 1. Otherwise the message is executed with priority 10.

Here are two sample XML files that can be used to invoke the service using a tool like JMeter or Ab. For SOAP user interface, the user can use the WSDL `repository/samples/resources/proxy/sample_proxy_1.wsdl` to create the request. The only difference between the two requests demonstrated here is the symbol. One has the symbol as IBM and other has MSFT. For one type of request set the priority header to 1 and for the next set the priority header to 10. Then load ESB with high volume of traffic from both types of requests using the load testing

tool. It prints the symbol of the incoming requests in the back end server. User should be able to see more of high priority symbol.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
 <soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
 <wsa:To>http://localhost:8281/services/SimpleStockQuoteService</wsa:To>
 <wsa:MessageID>urn:uuid:1B57D0B0BF770678DE1261165228620</wsa:MessageID>
 <wsa:Action>urn:getQuote</wsa:Action>
 </soapenv:Header>
 <soapenv:Body>
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>IBM</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </soapenv:Body>
</soapenv:Envelope>
```

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
 <soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
 <wsa:To>http://localhost:8281/services/SimpleStockQuoteService</wsa:To>
 <wsa:MessageID>urn:uuid:1B57D0B0BF770678DE1261165228620</wsa:MessageID>
 <wsa:Action>urn:getQuote</wsa:Action>
 </soapenv:Header>
 <soapenv:Body>
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>MSFT</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </soapenv:Body>
</soapenv:Envelope>
```

### Sample 653: NHTTP Transport Priority Based Dispatching

Objective: Demonstrate the priority based dispatching of NHTTP transport.

```

<priorityConfiguration>
 <priority-executor>
 <!-- two priorities specified with priority 10 and 1. Both priority messages
has a queue depth of 100 -->
 <queues isFixedCapacity="true"
nextQueue="org.apache.synapse.commons.executors.PRRNNextQueueAlgorithm">
 <queue size="100" priority="10"/>
 <queue size="100" priority="1"/>
 </queues>
 <!-- these are the default values, values are put here to show their
availability -->
 <threads core="20" max="100" keep-alive="5"/>
 </priority-executor>
 <!-- if a message comes that we cannot determine priority, we set a default
priority of 1 -->
 <conditions defaultPriority="1">
 <condition priority="10">
 <!-- check for the header named priority -->
 <equal type="header" source="priority" value="5"/>
 </condition>
 <condition priority="1">
 <equal type="header" source="priority" value="1"/>
 </condition>
 </conditions>
</priorityConfiguration>

```

#### Prerequisites:

- Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000. Priority is applied only when ESB is loaded with enough messages to consume its core number of threads. So to observe the priority based dispatching it is required to use a load testing tool like JMeter, SOAP UI or Apache bench.
- Open axis2.xml in repository/conf directory and uncomment the following parameter to the configuration priorityConfigFile. Set the value to repository/samples/resources/priority/priority-configuration.xml.
- Start ESB with the sample configuration 150: wso2esb-samples.sh -sn 150

In this sample, client should send a HTTP header that specifies the priority of the message. This header name is "priority". This header is retrieved in the priority configuration. Then it is matched against the value 1 and 10. Depending on this value message is executed with priority 1 or 10. Messages with smaller priority number will get higher priority.

Here are two sample XML files that can be used to invoke the service using a tool like JMeter, or Apache Ab. For SOAP user interface, the user can use the WSDL repository/conf/sample/resources/proxy/sample\_proxy\_1.wsdl to create the request. The only difference between the two requests demonstrated here is the symbol. One has the symbol as IBM and other has MSFT. For one type of request set the priority header to 1 and for the next set the priority header to 10. Then load ESB with high volume of traffic from both types of requests using the load testing tool. It prints the symbol of the incoming requests in the back end server. User should be able to see more of high priority symbol.

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
 <soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
 <wsa:To>http://localhost:8281/services/SimpleStockQuoteService</wsa:To>
 <wsa:MessageID>urn:uuid:1B57D0B0BF770678DE1261165228620</wsa:MessageID>
 <wsa:Action>urn:getQuote</wsa:Action>
 </soapenv:Header>
 <soapenv:Body>
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>IBM</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </soapenv:Body>
</soapenv:Envelope>

```

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
 <soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
 <wsa:To>http://localhost:8281/services/SimpleStockQuoteService</wsa:To>
 <wsa:MessageID>urn:uuid:1B57D0B0BF770678DE1261165228620</wsa:MessageID>
 <wsa:Action>urn:getQuote</wsa:Action>
 </soapenv:Header>
 <soapenv:Body>
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>MSFT</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </soapenv:Body>
</soapenv:Envelope>

```

### Sample 654: Smooks Mediator

**Objective:**Demonstrate the smooks mediator EDI message processing.

Here is the sample with smooks mediator.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <localEntry key="transform-xslt-key"
 src="file:repository/samples/resources/smooks/transform.xslt"/>
 <localEntry key="smooks-key"
src="file:repository/samples/resources/smooks/smooks-config.xml"/>
 <proxy name="StockQuoteProxy" transports="vfs">
 <parameter name="transport.vfs.ContentType">text/plain</parameter>
 <!--CHANGE-->
 <parameter name="transport.vfs.ContentType">text/plain</parameter>
 <parameter
name="transport.vfs.FileURI">file:///home/user/dev/test/smooks/in</parameter>
 <parameter name="transport.vfs.FileNamePattern">.*\.txt</parameter>
 <parameter name="transport.PollInterval">5</parameter>
 <!--CHANGE-->
 <parameter
name="transport.vfs.MoveAfterProcess">file:///home/user/dev/test/smooks/original</para
meter>
 <!--CHANGE-->
 <parameter
name="transport.vfs.MoveAfterFailure">file:///home/user/dev/test/smooks/original</para
meter>
 <parameter name="transport.vfs.ActionAfterProcess">MOVE</parameter>
 <parameter name="transport.vfs.ActionAfterFailure">MOVE</parameter>
 <parameter name="Operation">urn:placeOrder</parameter>
 <target>
 <inSequence>
 <smooks config-key="smooks-key" />
 <xslt key="transform-xslt-key" />
 <iterate expression="//m0:placeOrder/m0:order" preservePayload="true"
attachPath="//m0:placeOrder"
xmlns:m0="http://services.samples">
 <target>
 <sequence>
 <header name="Action" value="urn:placeOrder" />
 <property action="set" name="OUT_ONLY" value="true" />
 <send>
 <endpoint>
 <address format="soap11"
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </send>
 </sequence>
 </target>
 </iterate>
 </inSequence>
 <outSequence/>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/smooks/PlaceStockOrder.wsdl" />
 </proxy>
 </definitions>

```

1. Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000. Then add the plain text builder to the messageBuilders section of the axis2.xml found in the repository/conf directory. Here is the sample configuration.

```
<messageBuilder contentType="text/plain"
class="org.apache.axis2.format.PlainTextBuilder"/>
```

2. Enable the vfs transport in axis2.xml by uncommenting the vfs transport sender and receiver configurations in the axis2.xml.

3. User has to edit the synapse\_sample\_654.xml found in the repository/samples directory. These are the configuration parameters that needs to be edited.

- transport.vfs.FileURI
- transport.vfs.MoveAfterProcess
- transport.vfs.ActionAfterFailure

4. Start ESB with the sample configuration 654 (i.e. wso2esb-samples.sh -sn 654).

5. Drop the edi.txt file found in the repository/samples/resources/smooks directory to the transport.vfs.FileURI parameter specified directory.

### Sample 655: Message Relay - Basics Sample

**Objective:** Demonstrate the Message Relay.

This sample is similar to sample 150 except we have added two log mediators to show the actual message going through the ESB.

```

<?xml version="1.0"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" startOnLoad="true">
 <target>
 <inSequence>
 <log level="full"/>
 <send>
 <endpoint name="epr">
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <log level="full"/>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 </proxy>
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default "fault";
sequence"/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE"
expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
 </sequence>
 <sequence name="main">
 <log/>
 <drop/>
 </sequence>
</definitions>

```

**Prerequisites:** Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000. Then uncomment the Message Relay's Message Builder and Message Formatter in the axis2.xml. These configurations can be found in the messageFormatters and messageBuilders section of the axis2.xml. A message formatter or builder is defined for a content type. So make sure you comment out the normal builders and formatters for a content type when uncommenting the message relay builders and formatters.

Here are the Message Relay Formatter and Builder classes. You need to uncomment the entries containing the following builder and formatter.

```

org.wso2.carbon.relay.ExpandingMessageFormatter

org.wso2.carbon.relay.BinaryRelayBuilder

```

Send a message to the sample using the sample axis2 client.

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
```

You can see the messages going through the proxy by looking at the console because we have two log mediators in this sample. The actual message is not built and printed as a Base64 encoded string.

### Sample 656: Message Relay - Builder Mediator

**Objective:**Demonstrate the Message Relay with Builder mediator.

This sample is similar to [sample 655](#). We have added the builder mediator to build the actual message before logging it.

```
<?xml version="1.0"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="StockQuoteProxy" startOnLoad="true">
 <target>
 <inSequence>
 <builder/>
 <log level="full"/>
 <send>
 <endpoint name="epr">
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <builder/>
 <log level="full"/>
 <send/>
 </outSequence>
 </target>
 <publishWSDL uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
 </proxy>
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default "fault" sequence"/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
 </sequence>
 <sequence name="main">
 <log/>
 <drop/>
 </sequence>
</definitions>
```

**Prerequisites:**Set up is same as sample 655.

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
```

You can see the actual messages going through the proxy by looking at the console because we have two log mediators in this sample. Because we have the builder mediator in place, unlike in sample 655 we have the actual message printed.

### Sample 657: Distributed Transaction Management

- [Introduction](#)
- [Prerequisites](#)

- Building the sample
- Executing the sample
- Analyzing the output

### **Introduction**

This sample demonstrates the functionality of the **Transaction Mediator** using a sample distributed transaction. In this sample, a record is deleted from one database and added to a second database. If either of the operations (deleting from the first database and adding to the other) fails, all operations rollback, and the records do not change.

### **Prerequisites**

- For a list of general prerequisites, see [Setting Up the ESB Samples](#).
- The sample configuration uses two datasources and database instances to point to the sample databases. You have to manually create these in your environment for the sample to work.
  - Setup two distributed Derby databases `esbdb` and `esbdb1`. For instructions on setting up the Derby databases, see [Setting up Remote Derby](#).
  - Create a table in `esbdb` by executing the following statement.

```
CREATE table company(name varchar(10) primary key, id varchar(10), price double);
```

- Create a table in `esbdb1` by executing the following statement.

```
CREATE table company(name varchar(10) primary key, id varchar(10), price double);
```

- Insert records into the two tables that you created by executing the following statements.

#### **To insert records into the table in esbdb**

```
INSERT into company values ('IBM','c1',0.0);
INSERT into company values ('SUN','c2',0.0);
```

#### **To insert records into the table in esbdb1**

```
INSERT into company values ('SUN','c2',0.0);
INSERT into company values ('MSFT','c3',0.0);
```

### **Note**

When inserting records into the tables, the order of the record matters.

- Add the required datasource via the Management Console or create datasource declarations for the distributed databases in the `master-datasources.xml` file. For information on how you can add datasources via the management console, see [Adding Datasources](#). To create datasource declarations in the `master-datasources.xml` file, add the following code segments in the `master`

r-datasources.xml file located in the <ESB\_HOME>/repository/conf/datasources directory, ensuring that the datasource file names are \*-xa-ds.xml :  
**Datasource1: esb-derby-xa-ds.xml**

```
<datasources>
 <xa-datasource>
 <jndi-name>jdbc/XADerbyDS</jndi-name>
 <isSameRM-override-value>false</isSameRM-override-value>

 <xa-datasource-class>org.apache.derby.jdbc.ClientXADataSource</xa-datasource-class>
 <xa-datasource-property name="portNumber">1527</xa-datasource-property>
 <xa-datasource-property name="DatabaseName">esbdb</xa-datasource-property>
 <xa-datasource-property name="user">esb</xa-datasource-property>
 <xa-datasource-property name="password">esb</xa-datasource-property>
 <metadata>
 <type-mapping>Derby</type-mapping>
 </metadata>
 </xa-datasource>
 </datasources>
```

**Datasource2: esb-derby1-xa-ds.xml**

```
<datasources>
 <xa-datasource>
 <jndi-name>jdbc/XADerbyDS1</jndi-name>
 <isSameRM-override-value>false</isSameRM-override-value>

 <xa-datasource-class>org.apache.derby.jdbc.ClientXADataSource</xa-datasource-class>
 <xa-datasource-property name="portNumber">1527</xa-datasource-property>
 <xa-datasource-property name="DatabaseName">esbdb1</xa-datasource-property>
 <xa-datasource-property name="user">esb</xa-datasource-property>
 <xa-datasource-property name="password">esb</xa-datasource-property>
 <metadata>
 <type-mapping>Derby</type-mapping>
 </metadata>
 </xa-datasource>
 </datasources>
```

### **Building the sample**

The XML configuration for this sample is as follows:

```
<definitions xmlns="http://ws.apache.org/ns/synapse"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ws.apache.org/ns/synapse
 http://synapse.apache.org/ns/2010/04/configuration/synapse_config.xsd">
 <sequence name="myFaultHandler">
```

```

<log level="custom">
 <property name="text" value="** Rollback Transaction**"/>
</log>
<transaction action="rollback"/>
<send/>
</sequence>
<sequence name="main" onError="myFaultHandler">
 <in>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </in>
 <out>
 <transaction action="new"/>
 <log level="custom">
 <property name="text" value="** Reporting to the Database esbdb**"/>
 </log>
 <dbreport useTransaction="true" xmlns="http://ws.apache.org/ns/synapse">
 <connection>
 <pool>
 <dsName>jdbc/XADerbyDS</dsName>
 <user>esb</user>
 <password>esb</password>
 </pool>
 </connection>
 <statement>
 <sql>delete from company where name =?</sql>
 <parameter expression="//m0:return/m1:symbol/child::text()" type="VARCHAR"/>
 xmlns:m0="http://services.samples"
 xmlns:m1="http://services.samples/xsd"
 </statement>
 </dbreport>
 <log level="custom">
 <property name="text" value="** Reporting to the Database esbdbl**"/>
 </log>
 <dbreport useTransaction="true" xmlns="http://ws.apache.org/ns/synapse">
 <connection>
 <pool>
 <dsName>jdbc/XADerbyDS1</dsName>
 <user>esb</user>
 <password>esb</password>
 </pool>
 </connection>
 <statement>
 <sql>INSERT into company values ('IBM','c4',12.0)</sql>
 </statement>
 </dbreport>
 <transaction action="commit"/>
 <send/>
 </out>
</sequence>

```

```

</out>
</sequence>
</definitions>

```

This configuration file `synapse_sample_657.xml` is available in the `<ESB_HOME>/repository/samples` directory.

To build the sample

1. Start the ESB with the sample 657 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "WSO2 Carbon started in n seconds."

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

## Infor

WSO2 ESB comes with a default JTA transaction manager (Atomikos), which allows you to run distributed transactions without deploying the ESB on an external application server.

You now have a running ESB instance, databases and a back-end service deployed in your environment.

### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### **To execute the sample client**

- Execute the following command from the `<ESB_HOME>/samples/axis2Client` directory to test the successful scenario.

```

ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=IBM

```

- Execute the following command from the `<ESB_HOME>/samples/axis2Client` directory to test the failure scenario.

```

ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=SUN

```

### **Analyzing the output**

When you run the client to test the successful scenario, you will see that the IBM record is removed from the first database and added to the second database.

When you run the client to test the failure scenario, you will see that a record is neither deleted from the first database nor added into the second database. That is because executing the command attempts to add an already existing record again to the second database, which results in the fault sequence being executed. Therefore, an exception raised for duplicate entries and results in an entire transaction rollback.

## Sample 658: Huge XML Message Processing with Smooks Mediator

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### ***Introduction***

This sample demonstrates how to process huge XML messages via the splitting and routing approach using the [Smooks mediator](#). In this sample, the ESB reads a huge XML input file through the VFS transport, then the Smooks mediator splits the message into parts and routes each split fragment to a specified location.

### ***Prerequisites***

- Enable the VFS transport. For details, see [Enable VFS](#).
- For a list of general prerequisites, see [Setting Up the ESB Samples](#).
- Open the <ESB\_HOME>/repository/sample/synapse\_sample\_658.xml file in a text editor and change the following VFS transport configuration parameters based on the directory locations applicable to you.
  - `transport.vfs.FileURI` - The input file location.
  - `transport.vfs.MoveAfterProcess` - The location to move the input file after it is processed.
  - `transport.vfs.MoveAfterFailure` - The location to move the input file if there happens to be a failure at the time of processing.
- Open the <ESB\_HOME>/repository/samples/resources/smooks/smooks-config-658.xml file in a text editor and replace `/home/user/smooks/orders` in the `<file:destinationDirectoryPattern>/home/user/smooks/orders</file:destinationDirectoryPattern>` configuration line to a directory location applicable to you.

### ***Building the sample***

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <localEntry key="smooks-key"
src="file:repository/samples/resources/smooks/smooks-config-658.xml"/>
 <proxy name="SmooksSample" transports="vfs">
 <!--CHANGE-->
 <parameter
name="transport.vfs.FileURI">file:///home/user/smooks/in</parameter>
 <parameter name="transport.vfs.ContentType">application/xml</parameter>
 <parameter name="transport.vfs.FileNamePattern">.**.xml</parameter>
 <parameter name="transport.PollInterval">5</parameter>
 <!--CHANGE-->
 <parameter
name="transport.vfs.MoveAfterProcess">file:///home/user/smooks/original</parameter>
 <!--CHANGE-->
 <parameter
name="transport.vfs.MoveAfterFailure">file:///home/user/smooks/fail</parameter>
 <parameter name="transport.vfs.ActionAfterProcess">MOVE</parameter>
 <parameter name="transport.vfs.ActionAfterFailure">MOVE</parameter>
 <target>
 <inSequence>
 <smooks config-key="smooks-key">
 <input type="xml"/>
 <output type="xml"/>
 </smooks>
 </inSequence>
 </target>
 </proxy>
 </definitions>

```

This configuration file `synapse_sample_658.xml` is available in the `<ESB_HOME>/repository/samples` directory.

#### To build the sample

- Start the ESB with the sample 658 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

#### ***Executing the sample***

#### **To execute the sample**

Move the `input-message-658.xml` file from the `<ESB_HOME>/repository/samples/resources/smooks` directory to the location specified as `transport.vfs.FileURI` in the configuration.

If you want to try this sample with an input file of your own, be sure that the input file has a XML message similar to the format of `input-message-658.xml`.

#### ***Analyzing the output***

You will see the output of the incoming message that is split and routed via the Smooks mediator in the `destinationDirectoryPattern` location you specified when following the prerequisites.

### **Sample 659: Huge EDI Message Processing with Smooks Mediator**

- Introduction
- Prerequisites

- Building the sample
- Executing the sample
- Analyzing the output

### ***Introduction***

This sample demonstrates how to process huge EDI messages via the splitting and routing approach using the Smooks mediator. In this sample, the ESB reads a huge EDI input file through the VFS transport, then the Smooks mediator splits the message into parts and routes each split fragment to a specified location.

### ***Prerequisites***

- Enable the VFS transport. For details, see [Enable VFS](#).
- Add the following configuration to the **messageBuilders** section of the `<ESB_HOME>/repository/conf/axis2/axis2.xml` file:

```
<messageBuilder contentType="text/plain"
 class="org.apache.axis2.format.PlainTextBuilder"/>
```

- Open the `<ESB_HOME>/repository/sample/synapse_sample_659.xml` file in a text editor and change the following VFS transport configuration parameters based on the directory locations applicable to you.
  - `transport.vfs.FileURI`- The input file location.
  - `transport.vfs.MoveAfterProcess` - The location to move the input file after it is processed.
  - `transport.vfs.MoveAfterFailure` - The location to move the input file if there happens to be a failure at the time of processing.
- Open the `<ESB_HOME>/repository/samples/resources/smooks/smooks-config-659.xml` file in a text editor and replace `/home/user/smooks/orders` in the `<file:destinationDirectoryPattern>/home/user/smooks/orders</file:destinationDirectoryPattern>` configuration line to a directory location applicable to you.
- For a list of general prerequisites, see [Setting Up the ESB Samples](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <localEntry key="smooks-key"
src="file:repository/samples/resources/smooks/smooks-config-659.xml" />
 <proxy name="SmooksSample" transports="vfs" startOnLoad="true" trace="disable">
 <!--CHANGE-->
 <parameter name="transport.vfs.FileURI">file:///home/user/smooks/in</parameter>
 <parameter name="transport.vfs.ContentType">text/plain</parameter>
 <parameter name="transport.vfs.FileNamePattern">.*.txt</parameter>
 <parameter name="transport.PollInterval">5</parameter>
 <!--CHANGE-->
 <parameter
name="transport.vfs.MoveAfterProcess">file:///home/user/smooks/original</parameter>
 <!--CHANGE-->
 <parameter
name="transport.vfs.MoveAfterFailure">file:///home/user/smooks/fail</parameter>
 <parameter name="transport.vfs.ActionAfterProcess">MOVE</parameter>
 <parameter name="transport.vfs.ActionAfterFailure">MOVE</parameter>
 <target>
 <inSequence>
 <smooks config-key="smooks-key">
 <input type="text" />
 <output type="xml" />
 </smooks>
 </inSequence>
 </target>
 </proxy>
</definitions>

```

This configuration file `synapse_sample_659.xml` is available in the `<ESB_HOME>/repository/samples` directory.

To build the sample

- Start the ESB with the sample 659 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds*".

#### ***Executing the sample***

#### **To execute the sample**

Move the `input-message-659.txt` file from the `<ESB_HOME>/repository/samples/resources/smooks` directory to the location specified as `transport.vfs.FileURI` in the configuration.

If you want to try this sample with an input file of your own, be sure that the input file is in a format similar to `input-message-659.txt`.

#### ***Analyzing the output***

You will see the output of the incoming message that is split and routed via the Smooks mediator in the `destinationDirectoryPattern` location you specified when following the prerequisites.

### **Store and Forward Messaging Patterns with Message Stores and Processors**

The following samples are explained:

- [Sample 700: Introduction to Message Store](#)

- Sample 701: Introduction to Message Sampling Processor
- Sample 702: Introduction to Message Forwarding Processor
- Sample 703: Adding Security to Message Forwarding Processor
- Sample 704: RESTful Invocations with Message Forwarding Processor
- Sample 705: Load Balancing with Message Forwarding Processor

## Sample 700: Introduction to Message Store

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

### ***Introduction***

This sample demonstrates the basic functionality of a message store.

### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

```
<!-- Introduction to the Synapse Message Store -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default 'fault' sequence"/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE"
expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
 </sequence>
 <sequence name="onStoreSequence">
 <log>
 <property name="On-Store" value="Storing message"/>
 </log>
 </sequence>
 <sequence name="main">
 <in>
 <log level="full"/>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
 <store messageStore="MyStore" sequence="onStoreSequence"/>
 </in>
 <description>The main sequence for the message mediation</description>
 </sequence>
 <messageStore name="MyStore"/>
</definitions>
```

This configuration file `synapse_sample_700.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### **To build the sample**

1. Start the ESB with the sample 700 configuration. For instructions on starting a sample ESB configuration,

see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "WSO2 Carbon started in n seconds."

2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

#### **To execute the sample client**

- Run the following command from the <ESB\_HOME>/samples/axis2Client directory:

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dmode=placeorder
```

### **Analyzing the output**

When you execute the client you will see that the message is dispatched to the main sequence.

In the main sequence, the store mediator will store the `placeorder` request message in the `MyStore` message store. Before storing the request, the message store mediator will invoke the `onStoreSequence` sequence.

Analyze the output debug messages and you will see the following log:

```
INFO - LogMediator To: http://localhost:9000/services/SimpleStockQuoteService,
WSAction: urn:placeOrder, SOAPAction: urn:placeOrder, ReplyTo:
http://www.w3.org/2005/08/addressing/none, MessageID:
urn:uuid:54f0e7c6-7b43-437c-837e-a825d819688c, Direction: request, On-Store = Storing
message
```

You can then use the JMX view of the Synapse message store by using the jconsole in order to view the stored messages and delete them.

### **Sample 701: Introduction to Message Sampling Processor**

**Objective:**Introduction to Message Sampling Processor.

```

<!-- Introduction to Message Sampling Processor -->

<definitions xmlns="http://ws.apache.org/ns/synapse">

<sequence name="send_seq">
<send>
<endpoint>
<address uri="http://localhost:9000/services/SimpleStockQuoteService">
<suspendOnFailure>
<errorCodes>-1</errorCodes>
<progressionFactor>1.0</progressionFactor>
</suspendOnFailure>
</address>
</endpoint>
</send>
</sequence>
<sequence name="main">
<in>
<log level="full"/>
<property name="FORCE_SC_ACCEPTED" value="true" scope="axis2" />
<property name="OUT_ONLY" value="true" />
<store messageStore="MyStore" />
</in>
<description>The main sequence for the message mediation</description>
</sequence>
<messageStore name="MyStore" />
<messageProcessor
class="org.apache.synapse.message.processors.sampler.SamplingProcessor"
name="SamplingProcessor" messageStore="MyStore">
<parameter name="interval">20000</parameter>
<parameter name="sequence">send_seq</parameter>
</messageProcessor>
</definitions>

```

### Prerequisites:

- Start the configuration numbered 701: i.e. wso2esb-samples -sn 701
- Start the SimpleStockQuoteService if its not already started

To Execute the Client few times:

```

ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dmode=placeorder

```

When you execute the client the message will be dispatched to the main sequence. In the Main sequence store mediator will store the placeOrder request message in the "MyStore" Message Store. Message Processor will consume the messages and forward to the send\_seq sequence in configured rate. You will observe that service invocation rate is not changing when increasing the rate we execute the client.

### Sample 702: Introduction to Message Forwarding Processor

**Objective:**Introduction to Message Forwarding Processor.

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <taskManager provider="org.wso2.carbon.mediation.ntask.NTaskTaskManager"/>
 <endpoint name="StockQuoteServiceEp">
 <address uri="http://localhost:9000/services/SimpleStockQuoteService">
 <suspendOnFailure>
 <errorCodes>-1</errorCodes>
 <progressionFactor>1.0</progressionFactor>
 </suspendOnFailure>
 </address>
 </endpoint>
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default 'fault' sequence"/>
 <property expression="get-property('ERROR_CODE')" name="ERROR_CODE"/>
 <property expression="get-property('ERROR_MESSAGE')" name="ERROR_MESSAGE" />
 </log>
 <drop/>
 </sequence>
 <sequence name="main">
 <in>
 <log level="full"/>
 <property name="FORCE_SC_ACCEPTED" scope="axis2" value="true"/>
 <property name="OUT_ONLY" value="true"/>
 <store messageStore="MyStore"/>
 </in>
 <description>The main sequence for the message mediation</description>
 </sequence>
 <messageStore name="MyStore" />
 <messageProcessor
 class="org.apache.synapse.message.processor.impl.forwarder.ScheduledMessageForwardingProcessor"
 messageStore="MyStore" name="ScheduledProcessor"
 targetEndpoint="StockQuoteServiceEp">
 <parameter name="interval">10000</parameter>
 <parameter name="throttle">false</parameter>
 <parameter name="target.endpoint">StockQuoteServiceEp</parameter>
 </messageProcessor>
</definitions>

```

### Prerequisites:

- Start the configuration numbered 702: i.e. wso2esb-samples -sn 702

### To Execute the Client:

```

ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dmode=placeorder

```

Now Start the SimpleStockQuoteService. When you Start the service you will see message getting delivered to the service. Even though service is down when we invoke it from the client. Here in the Main sequence store mediator

will store the placeOrder request message in the "MyStore" Message Store. Message Processor will send the message to the endpoint configured as a message context property. Message processor will remove the message from the store only if message delivered successfully.

### Sample 703: Adding Security to Message Forwarding Processor

**Objective:** Add security policies to the Message Forwarding Processor.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <registry provider="org.wso2.carbon.mediation.registry.WSO2Registry">
 <parameter name="cachableDuration">15000</parameter>
 </registry>
 <proxy name="StockQuoteProxy" transports="https http" startOnLoad="true"
trace="disable">
 <description />
 <target>
 <inSequence>
 <property name="OUT_ONLY" value="true" />
 <store messageStore="MSG_STORE" />
 </inSequence>
 <outSequence>
 <send />
 </outSequence>
 </target>
 </proxy>
 <localEntry key="sec_policy"
src="file:repository/samples/resources/policy/policy_3.xml" />
 <endpoint name="SecureStockQuoteService">
 <address uri="http://localhost:9000/services/SecureStockQuoteService">
 <enableSec policy="sec_policy" />
 </address>
 </endpoint>
 <messageStore name="MSG_STORE" />
 <messageProcessor
class="org.apache.synapse.message.processor.impl.forwarder.ScheduledMessageForwardingP
rocessor" name="SecureForwardingProcessor" targetEndpoint="SecureStockQuoteService"
messageStore="MSG_STORE">
 <parameter name="client.retry.interval">1000</parameter>
 <parameter name="interval">1000</parameter>
 <parameter name="is.active">true</parameter>
 </messageProcessor>
</definitions>

```

### Prerequisites

- Start the Synapse configuration numbered 703, e.g., wso2esb-samples.sh -sn 703
- Start the Axis2 server and deploy the SecureStockQuoteService if you have not done so already.
- If needed, download and install the unlimited strength policy files for your JDK before using Apache Rampart (see [http://java.sun.com/javase/downloads/index\\_jdk5.jsp](http://java.sun.com/javase/downloads/index_jdk5.jsp))

Use the stockquote client to send a request without WS-Security. ESB is configured to enable WS-Security as per the policy specified by 'policy\_3.xml' for the outgoing messages to the SecureStockQuoteService endpoint hosted on the Axis2 instance. The debug log messages on ESB shows the encrypted message flowing to the service and the encrypted response being received by ESB.

```

ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
-Dmode=placeorder -Dsymbol=WSO2

```

You can see the message sent by the ESB to the secure service using a TCPMon. Upon successful execution, there should be a message on the back end as follows:

```
Sun Aug 18 10:58:00 IST 2013 samples.services.SimpleStockQuoteService :: Accepted
order #5 for : 18851 stocks of WSO2 at $ 61.782478265721714
```

Start the SimpleStockQuoteService. When you start the service, you will see the message getting delivered to the service, even though the service was down when we invoked it from the client. The Main sequence store mediator stores the placeOrder request message in the "MyStore" message store, and the message processor sends the message to the endpoint configured as a message context property. The message processor will remove the message from the store only if the message is delivered successfully.

#### Sample 704: RESTful Invocations with Message Forwarding Processor

**Objective:** Invoke REST messages with a plain-old XML (POX) back end using the message store and Message Forwarding Processor.

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <registry provider="org.wso2.carbon.mediation.registry.WSO2Registry">
 <parameter name="cachableDuration">15000</parameter>
 </registry>
 <proxy name="StockQuoteProxy" transports="https http" startOnLoad="true"
trace="disable">
 <description />
 <target>
 <inSequence>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2" />
 <property name="OUT_ONLY" value="true" />
 <property name="messageType" value="application/xml" scope="axis2" />
 <store messageStore="MSG_STORE" />
 </inSequence>
 <outSequence>
 <send />
 </outSequence>
 </target>
 </proxy>
 <endpoint name="SecureStockQuoteService">
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"
format="rest" />
 </endpoint>
 <sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default 'fault' sequence" />
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop />
 </sequence>
 <sequence name="main">
 <in>
 <log level="full" />
 <filter source="get-property('To')" regex="http://localhost:9000.*">
 <send />
 </filter>
 </in>
 <out>
 <send />
 </out>
 <description>The main sequence for the message mediation</description>
 </sequence>
 <messageStore name="MSG_STORE" />
 <messageProcessor
class="org.apache.synapse.message.processor.impl.forwarder.ScheduledMessageForwardingP
rocessor" name="SecureForwardingProcessor" targetEndpoint="SecureStockQuoteService"
messageStore="MSG_STORE">
 <parameter name="client.retry.interval">1000</parameter>
 <parameter name="interval">1000</parameter>
 <parameter name="is.active">true</parameter>
 </messageProcessor>
</definitions>

```

## Prerequisites

- Start the Synapse configuration numbered 704, e.g., `wso2esb-samples.sh -sn 704`

- Start the Axis2 server and deploy the SecureStockQuoteService if you have not done so already.

This sample demonstrate how to do RESTful invocations with the message store and message forwarding processor. Though in this sample we have used an In memory message store, these instructions are valid for any available message store implementations. Use the stockquote client to send the following placeorder request:

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
-Dmode=placeorder -Dsymbol=WSO2
```

You can see the message sent by the ESB to the secure service using a TCPMon. Upon successful execution, there should be a message on the back end as follows:

```
Sun Aug 18 10:58:00 IST 2013 samples.services.SimpleStockQuoteService :: Accepted
order #5 for : 18851 stocks of WSO2 at $ 61.782478265721714
```

## Sample 705: Load Balancing with Message Forwarding Processor

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

### ***Introduction***

This sample demonstrates how you can load balance messages using the message store and message forwarding processor.

### ***Prerequisites***

- Install [ActiveMQ](#) and copy the ActiveMQ client JARs activemq-core-5.2.0.jar and geronimo-j2ee-management\_1.0\_spec-1.0.jar into the <ESB\_HOME>/repository/components/lib directory to allow the ESB to connect to the JMS provider.
- Start ActiveMQ on its default port.
- For a list of general prerequisites, see [Prerequisites to start the ESB samples](#)

### ***Building the sample***

The XML configuration for this sample is as follows:

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <registry provider="org.wso2.carbon.mediation.registry.WSO2Registry">
 <parameter name="cachableDuration">15000</parameter>
 </registry>
 <taskManager provider="org.wso2.carbon.mediation.ntask.NTaskTaskManager"/>
 <proxy name="StockQuoteProxy">
 <transports>https http</transports>
 <startOnLoad>true</startOnLoad>
 <trace>disable</trace>
 <description/>
 <target>
 <inSequence>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
 <property name="OUT_ONLY" value="true"/>
 <store messageStore="JMSMS"/>
 </inSequence>
 </target>
 </proxy>
</definitions>
```

```

 </target>
 </proxy>
<endpoint name="SimpleStockQuoteService3">
 <address uri="http://localhost:9003/services/SimpleStockQuoteService"/>
</endpoint>
<endpoint name="SimpleStockQuoteService2">
 <address uri="http://localhost:9002/services/SimpleStockQuoteService"/>
</endpoint>
<endpoint name="SimpleStockQuoteService1">
 <address uri="http://localhost:9001/services/SimpleStockQuoteService"/>
</endpoint>
<sequence name="fault">
 <log level="full">
 <property name="MESSAGE" value="Executing default 'fault' sequence"/>
 <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
 <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
 </log>
 <drop/>
</sequence>
<sequence name="main">
 <in>
 <log level="full"/>
 <filter source="get-property('To')" regex="http://localhost:9000.*">
 <send/>
 </filter>
 </in>
 <out>
 <send/>
 </out>
 <description>The main sequence for the message mediation</description>
</sequence>
<messageStore class="org.apache.synapse.message.store.impl.jms.JmsStore"
name="JMSMS">
 <parameter
name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="store.jms.cache.connection">false</parameter>
 <parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
 <parameter name="store.jms.JMSSpecVersion">1.1</parameter>
 <parameter name="store.jms.destination">JMSMS</parameter>
</messageStore>
<messageProcessor
class="org.apache.synapse.message.processor.impl.forwarder.ScheduledMessageForwardingProcessor"
name="Forwarder3"
targetEndpoint="SimpleStockQuoteService3"
messageStore="JMSMS">
 <parameter name="client.retry.interval">1000</parameter>
 <parameter name="interval">1000</parameter>
 <parameter name="is.active">true</parameter>
</messageProcessor>
<messageProcessor
class="org.apache.synapse.message.processor.impl.forwarder.ScheduledMessageForwardingProcessor"
name="Forwarder1"
targetEndpoint="SimpleStockQuoteService1"
messageStore="JMSMS">
 <parameter name="client.retry.interval">1000</parameter>
 <parameter name="interval">1000</parameter>

```

```
<parameter name="is.active">true</parameter>
</messageProcessor>
<messageProcessor
class="org.apache.synapse.message.processor.impl.forwarder.ScheduledMessageForwardingP
rocessor"
 name="Forwarder2"
 targetEndpoint="SimpleStockQuoteService2"
 messageStore="JMSMS">
<parameter name="client.retry.interval">1000</parameter>
<parameter name="interval">1000</parameter>
```

```

<parameter name="is.active">true</parameter>
</messageProcessor>
</definitions>

```

This configuration file `synapse_sample_705.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### To build the sample

1. Start the ESB with the sample 705 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).

The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

2. Start three instances of the sample Axis2 server on HTTP ports 9001, 9002 and 9003 and give unique names to each server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

Now you have a running ESB instance and a back-end service deployed. In the next section, we will send a message to the back-end service through the ESB using a sample client.

### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### To execute the sample client

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory:

```

ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
-Dmode=placeorder -Dsymbol=WSO2

```

### **Analyzing the output**

You can analyze the message sent by the ESB to the secure service using **TCPMon**.

On successful execution of the placeorder request, you will see the following message on the back-end:

```

Sun Aug 18 10:58:00 IST 2013 samples.services.SimpleStockQuoteService :: Accepted
order #5 for : 18851 stocks of WSO2 at $ 61.782478265721714

```

If you use the stockqoute client to send the placeorder request several times and observe the log on the back-end server, you will see that the messages are distributed randomly among the back-end nodes since you send the placeorder request randomly. However, if you send the request message evenly (such as when using soapUI), you will see that the messages are evenly distributed among the back-end nodes.

## Template Samples

- [Sample 750: Stereotyping XSLT Transformations with Templates](#)
- [Sample 751: Message Split Aggregate Using Templates](#)
- [Sample 752: Load Balancing Between 3 Endpoints With Endpoint Templates](#)

### **Sample 750: Stereotyping XSLT Transformations with Templates**

## Objective: Introduction to ESB Sequence Templates.

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <proxy name="StockQuoteProxy">
 <target>
 <inSequence>
 <!--use sequence template to transform incoming request-->
 <call-template target="xslt_func">
 <with-param name="xslt_key" value="xslt-key-req"/>
 </call-template>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </inSequence>

 <outSequence>
 <!--use sequence template to transform incoming response-->
 <call-template target="xslt_func">
 <with-param name="xslt_key" value="xslt-key-back"/>
 </call-template>
 <send/>
 </outSequence>
 </target>
</proxy>

<!--this sequence template will transform requests with the given xslt local
entry key And will log
the message before and after. Takes Iterate local entry key as an argument-->
<template xmlns="http://ws.apache.org/ns/synapse" name="xslt_func">
 <parameter name="xslt_key"/>
 <sequence>
 <log level="full">
 <property name="BEFORE_TRANSFORM" value="true" />
 </log>
 <xslt key="${func:xslt_key}" />
 <log level="full">
 <property name="AFTER_TRANSFORM" value="true" />
 </log>
 </sequence>
</template>

<localEntry key="xslt-key-req"
src="file:repository/samples/resources/transform/transform.xslt"/>
<localEntry key="xslt-key-back"
src="file:repository/samples/resources/transform/transform_back.xslt"/>
</definitions>

```

### Prerequisites:

- Start the Synapse configuration numbered 750: i.e. wso2esb-samples -sn 750
- Start the Axis2 server and deploy the SimpleStockQuoteService if not already done

Execute the stock quote client by requesting for a stock quote on the proxy service as follows:

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
-Dmode=customquote
```

ESB Sequence Template can act a reusable function. Here proxy service reuses template xslt\_func which will transform requests with the given xslt local entry key And will log the message before and after. It takes xslt transformation corresponding to local entry key as an argument (for insequence this key is xslt-key-req and out sequence it is xslt-key-back). We use call-template mediator for passing the xslt key parameter to a sequence template. ESB console will display how the custom stockquote is transformed and the transformed response from the service.

### Sample 751: Message Split Aggregate Using Templates

**Objective:** Demonstrate the use of tempaltes in Split/Aggregate scenario.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">

 <proxy name="SplitAggregateProxy">
 <target>
 <inSequence>
 <!--use iterate sequence template to split incoming request and send-->
 <call-template target="iter_func">
 <with-param xmlns:m0="http://services.samples"
name="iter_expr" value="{{//m0:getQuote/m0:request}}"/>
 <with-param xmlns:m0="http://services.samples" name="attach_path"
value="{{//m0:getQuote}}"/>
 </call-template>
 </inSequence>
 <outSequence>
 <!--use aggregate sequence template to combine the responses and send back-->
 <call-template target="aggr_func">
 <with-param xmlns:m0="http://services.samples"
name="aggr_expr" value="{{//m0:getQuoteResponse}}"/>
 </call-template>
 </outSequence>
 </target>
 </proxy>

 <!--this sequence template will aggregate the responses , merge and send back to
the client. This takes aggregate
expression as an argument-->
<template xmlns="http://ws.apache.org/ns/synapse" name="aggr_func">
 <parameter name="aggr_expr" />
 <sequence>
 <log level="full"/>
 <aggregate>
 <completeCondition>
 <messageCount min="-1" max="-1" />
 </completeCondition>
 <onComplete expression="$func:aggr_expr">
 <log level="full" />
 <send/>
 </onComplete>
 </aggregate>
 </sequence>
</template>

 <!--this sequence template will iterate through stock quote symbols ,split and
```

```
send them to endpoints. Takes Iterate
expression and soap attaach path as arguments -->
<template xmlns="http://ws.apache.org/ns/synapse" name="iter_func">
 <parameter name="iter_expr"/>
 <parameter name="attach_path"/>
 <sequence>

 <iterate xmlns:m0="http://services.samples" preservePayload="true"
attachPath="$func:attach_path" expression="$func:iter_expr">
 <target>
 <sequence>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </sequence>
 </target>
 </iterate>
```

```

 </sequence>
</template>
</definitions>

```

**Prerequisites:**

- Deploy the SimpleStockQuoteService in sample Axis2 server and start it on port 9000.
- Start ESB with the sample configuration 751 (i.e. wso2esb-samples -sn 751).

This will be the same sample demonstrated in advanced mediation with Iterate mediator used to split the messages in to parts and process them asynchronously and then aggregate the responses coming in to ESB. Only addition will be two sequence templates 'iter\_func' and 'aggr\_func' which will split the message and aggregate the responses back , respectively.

Invoke the client as follows.

```
ant stockquote -Daddurl=http://localhost:8280/services/SplitAggregateProxy -Ditr=4
```

Sequence Template 'iter\_func' takes iterate expressions and attach path as arguments. Sequence Template 'aggr\_func' takes aggregate expression as argument.

In this sample we are using special dynamic xpath expressions(in double curly braces {{expr}}) in call-template mediator so that they are evaluated only when iteration and aggregation is done. Note that function scope (ie:-\$func) is used to access template parameters in 'iter\_func' and 'aggr\_func'.

**Sample 752: Load Balancing Between 3 Endpoints With Endpoint Templates**

**Objective:** Demonstrate how to use endpoint templates in a load balance configuration.

```

<definitions xmlns="http://ws.apache.org/ns/synapse>

 <proxy name="LBProxy" transports="https http" startOnLoad="true">
 <target faultSequence="errorHandler">
 <inSequence>
 <send>
 <endpoint>
 <session type="simpleClientSession"/>
 <loadbalance
algorithm="org.apache.synapse.endpoints.algorithms.RoundRobin">
 <endpoint name="templ_ep1" template="endpoint_template"
uri="http://localhost:9001/services/LBService1">
 <parameter name="suspend_duration" value="5"/>
 </endpoint>
 <endpoint name="templ_ep2" template="endpoint_template"
uri="http://localhost:9002/services/LBService1">
 <parameter name="suspend_duration" value="20"/>
 </endpoint>
 <endpoint name="templ_ep3" template="endpoint_template"
uri="http://localhost:9003/services/LBService1">
 <parameter name="suspend_duration" value="200"/>
 </endpoint>
 </loadbalance>
 </endpoint>
 </send>
 <drop/>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </target>
 <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_2.wsdl"/>
 </proxy>

 <sequence name="errorHandler">
 <makefault>
 <code value="tns:Receiver"
xmlns:tns="http://www.w3.org/2003/05/soap-envelope">
 <reason value="COULDN'T SEND THE MESSAGE TO THE SERVER."/>
 </makefault>
 <send/>
 </sequence>

 <template name="endpoint_template">
 <parameter name="suspend_duration"/>
 <endpoint name="annonymous">
<address uri="$uri">
 <enableAddressing/>

<suspendDurationOnFailure>$suspend_duration</suspendDurationOnFailure>
 </address>
 </endpoint>
 </template>
 </definitions>

```

**Prerequisites:**

- Sample setup is same as LoadBalance endpoint scenario (#154).
- Start the Synapse configuration numbered 752: i.e. wso2esb-samples -sn 752
- Start the Axis2 server and deploy the LoadbalanceFailoverService if not already done

Configuration demonstrates a single endpoint template named 'endpoint\_template'. Three endpoint templates are created inside loadbalance endpoint which will point to this template. Note different parameters passed on to the same template ie:- service url and suspend duration as parameters

Invoke the client as follows.

```
ant loadbalancefailover -Dmode=session -Dtrpurl=http://localhost:8280/services/LBProxy
```

As in the previous loadbalance samples client console will show loadbalance requests going towards three endpoints created from the same template. Notice differnt suspension times for three services in a failed endpoint scenario as well. (reflects different parameters passed onto the template).

## REST API Management

The following API Management samples are available with WSO2 Enterprise Service Bus (ESB) :

- [Sample 800: Introduction to REST API](#)

### Sample 800: Introduction to REST API

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <!-- You can add any flat sequences, endpoints, etc.. to this synapse.xml file if
you do *not* want to keep the artifacts in several files -->
 <api name="StockQuoteAPI" context="/stockquote">
 <resource uri-template="/view/{symbol}" methods="GET">
 <inSequence>
 <payloadFactory>
 <format>
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>$1</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </format>
 <args>
 <arg expression="get-property('uri.var.symbol')"/>
 </args>
 </payloadFactory>
 <header name="Action" value="urn:getQuote"/>
 <send>
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"
format="soap11"/>
 </endpoint>
 </send>
 </inSequence>
 <outSequence>
 <send/>
 </outSequence>
 </resource>
 <resource methods="POST" url-mapping="/order/*">
 <inSequence>
 <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
 <property name="OUT_ONLY" value="true"/>
 <header name="Action" value="urn:placeOrder"/>
 <send>
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService" format="soap11"/>
 </endpoint>
 </send>
 </inSequence>
 </resource>
 </api>
</definitions>

```

## Objective:

Introduce the REST API support in WSO2 ESB.

## Prerequisites:

1. You have to start the axis2 server and deploy the SimpleStockQuoteService.
2. Start the Synapse configuration numbered 800: `wso2esb-samples.sh -sn 800`
3. You need a REST client like curl to test this.

```
curl -v http://127.0.0.1:8280/stockquote/view/IBM
```

```
curl -v http://127.0.0.1:8280/stockquote/view/MSFT
```

The GET calls are handled by the first resource in the StockQuoteAPI. These REST calls will get converted into SOAP calls and sent to the Axis2 server. The response will be sent to the client in POX format.

The following command posts a simple XML to the ESB. Save the following sample request as "placeorder.xml" in your local file system and execute the command that is used to invoke a SOAP service. The ESB returns the 202 response back to the client.

```
curl -v -d @placeorder.xml -H "Content-type: application/xml"
http://127.0.0.1:8280/stockquote/order/
```

```
<placeOrder xmlns="http://services.samples">
<order>
 <price>50</price>
 <quantity>10</quantity>
 <symbol>IBM</symbol>
</order>
</placeOrder>
```

## Inbound Endpoint Samples

The following inbound endpoint samples are available with WSO2 Enterprise Service Bus (ESB) :

- Sample 900: Inbound Endpoint File Protocol Sample (VFS)
- Sample 901: Inbound Endpoint JMS Protocol Sample
- Sample 902: HTTP Inbound Endpoint Sample
- Sample 903: HTTPS Inbound Endpoint Sample
- Sample 904: Inbound Endpoint Kafka Protocol Sample
- Sample 905: Inbound HL7 with Automatic Acknowledgement
- Sample 906: Inbound Endpoint MQTT Protocol Sample
- Sample 907: Inbound Endpoint RabbitMQ Protocol Sample

### Sample 900: Inbound Endpoint File Protocol Sample (VFS)

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

#### ***Introduction***

This sample demonstrates how to use the file system as an input medium via the inbound file listener.

#### ***Prerequisites***

- Create 3 new directories named `in`, `out` and `original` in a test directory (e.g., `/home/user/test`) in the local file system. Then, open the `<ESB_HOME>/repository/samples/synapse_sample_900.xml` file in a text editor and change the `transport.vfs.FileURI`, `transport.vfs.MoveAfterProcess`, `transpor`

t.vfs.MoveAfterFailure parameter values to the **in**, **original** and **original** directory locations respectively.

### **Building the sample**

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <registry provider="org.wso2.carbon.mediation.registry.WSO2Registry">
 <parameter name="cachableDuration">15000</parameter>
 </registry>
 <taskManager provider="org.wso2.carbon.mediation.ntask.NTaskTaskManager">
 <parameter name="cachableDuration">15000</parameter>
 </taskManager>
 <inboundEndpoint xmlns="http://ws.apache.org/ns/synapse" name="file_inbound"
sequence="request" onError="fault" protocol="file" suspend="false">
 <parameters>
 <parameter name="interval">1000</parameter>
 <parameter name="sequential">true</parameter>
 <parameter name="transport.vfs.FileURI">file:///home/user/test/in</parameter>
 <!--CHANGE-->
 <parameter name="transport.vfs.ContentType">text/xml</parameter>
 <parameter name="transport.vfs.FileNamePattern">.*\..xml</parameter>
 <parameter
name="transport.vfs.MoveAfterProcess">file:///home/user/test/original</parameter>
 <!--CHANGE-->
 <parameter
name="transport.vfs.MoveAfterFailure">file:///home/user/test/original</parameter>
 <!--CHANGE-->
 <parameter name="transport.vfs.ActionAfterProcess">MOVE</parameter>
 <parameter name="transport.vfs.ActionAfterFailure">MOVE</parameter>
 </parameters>
 </inboundEndpoint>
 <sequence name="request" onError="fault">
 <call>
 <endpoint>
 <address format="soap12"
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </call>
 <drop/>
 </sequence>
</definitions>
```

This configuration file `synapse_sample_900.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### **To build the sample**

1. Start the ESB with the sample 900 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).  
The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds*".
2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

### **Executing the sample**

## To execute the sample client

- Copy the `ESB_HOME/repository/samples/resources/vfs/test.xml` file to the location specified in `transport.vfs.FileURI` in the configuration (i.e., the `in` directory).

The `test.xml` file contains a simple stock quote request and is as follows:

```
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:wsa="http://www.w3.org/2005/08/addressing">
 <soapenv:Body>
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>IBM</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </soapenv:Body>
</soapenv:Envelope>
```

## Analyzing the output

You will see that the inbound polling file listener picks the file from `in` directory and sends it to the Axis2 service, and that the request XML file is moved to `original` directory.

## Sample 901: Inbound Endpoint JMS Protocol Sample

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### Introduction

This sample demonstrates how one way message bridging from JMS to HTTP can be done using the inbound JMS endpoint.

### Prerequisites

- Download and set up Apache ActiveMQ. Instructions can be found in section [Installation Prerequisites](#).
- Copy the following client libraries from the `<AMQ_HOME>/lib` directory to the `<ESB_HOME>/repository/components/lib` directory.

#### ActiveMQ 5.8.0 and above

- `activemq-broker-5.8.0.jar`
- `activemq-client-5.8.0.jar`
- `geronimo-jms_1.1_spec-1.1.1.jar`
- `geronimo-j2ee-management_1.1_spec-1.0.1.jar`
- `hawtbuf-1.9.jar`

#### Earlier version of ActiveMQ

- `activemq-core-5.5.1.jar`
- `geronimo-j2ee-management_1.0_spec-1.0.jar`
- `geronimo-jms_1.1_spec-1.1.1.jar`

### Building the sample

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <registry provider="org.wso2.carbon.mediation.registry.WSO2Registry">
 <parameter name="cachableDuration">15000</parameter>
 </registry>
 <taskManager provider="org.wso2.carbon.mediation.ntask.NTaskTaskManager">
 <parameter name="cachableDuration">15000</parameter>
 </taskManager>
 <inboundEndpoint xmlns="http://ws.apache.org/ns/synapse" name="jms_inbound"
sequence="request" onError="fault" protocol="jms" suspend="false">
 <parameters>
 <parameter name="interval">1000</parameter>
 <parameter name="transport.jms.Destination">ordersQueue</parameter>
 <parameter name="transport.jms.CacheLevel">1</parameter>
 <parameter
name="transport.jms.ConnectionFactoryJNDIName">QueueConnectionFactory</parameter>
 <parameter name="sequential">true</parameter>
 <parameter
name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFact
ory</parameter>
 <parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
 <parameter
name="transport.jms.SessionAcknowledgement">AUTO_ACKNOWLEDGE</parameter>
 <parameter name="transport.jms.SessionTransacted">false</parameter>
 <parameter name="transport.jms.ConnectionFactoryType">queue</parameter>
 </parameters>
 </inboundEndpoint>
 <sequence name="request" onError="fault">
 <call>
 <endpoint>
 <address format="soap12"
uri="http://localhost:9000/services/SimpleStockQuoteService" />
 </endpoint>
 </call>
 <drop/>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_901.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

1. Start the ESB with the sample 901 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).  
The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"
2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

## Executing the sample

### To execute the sample client

- Log on to the ActiveMQ console using the `http://localhost:8161/admin` url.
- Browse the queue `ordersQueue` listening via the above endpoint.
- Add a new message with the following content to the queue:

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:wsa="http://www.w3.org/2005/08/addressing">
 <soapenv:Body>
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>IBM</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </soapenv:Body>
</soapenv:Envelope>

```

### **Analyzing the output**

You will see that the JMS endpoint gets the message from the queue and sends it to the stock quote service.

## **Sample 902: HTTP Inbound Endpoint Sample**

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### ***Introduction***

This sample demonstrates how an HTTP inbound endpoint can act as a dynamic http listener. Many http listeners can be added without restarting the server. When a message arrives at a port it will bypass the inbound side axis2 layer and will be sent directly to the sequence for mediation. The response also behaves in the same way.

### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
 name="HttpListenerEP1"
 sequence="TestIn"
 onError="fault"
 protocol="http"
 suspend="false">
 <parameters>
 <parameter name="inbound.http.port">8085</parameter>
 </parameters>
 </inboundEndpoint>
 <sequence xmlns="http://ws.apache.org/ns/synapse" name="TestIn">
 <send receive="receiveSeq">
 <endpoint>
 <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </sequence>
 <sequence xmlns="http://ws.apache.org/ns/synapse" name="receiveSeq">
 <send/>
 </sequence>
</definitions>

```

This configuration file `synapse_sample_902.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### To build the sample

1. Start the ESB with the sample 902 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).  
The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds*".
2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

### Executing the sample

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### To execute the sample client

- Run the following command from the `<ESB_HOME>/samples/axis2Client` directory, to execute the **Stock Quote Client** in the **Dumb Client Mode**.

```
ant stockquote -Dtrpurl=http://localhost:8085/
```

### Analyzing the output

Analyze the output debug messages for the actions in the dumb client mode.

You will see that the ESB receives a message when the ESB Inbound is set as the ultimate receiver. You will also see the response from the back end in the client.

### Sample 903: HTTPS Inbound Endpoint Sample

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

### ***Introduction***

This sample demonstrates how an HTTPS inbound endpoint can act as a dynamic https listener. Many https listeners can be added without restarting the server. When a message arrives at a port it will bypass the inbound side axis2 layer and will be sent directly to the sequence for mediation. The response also behaves in the same way.

### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
 <inboundEndpoint name="HttpsListenerEP"
 protocol="https"
 suspend="false" sequence="TestIn" onError="fault" >
 <p:parameters xmlns:p="http://ws.apache.org/ns/synapse">
 <p:parameter name="inbound.http.port">8081</p:parameter>
 <p:parameter name="keystore">
 <KeyStore>
 <Location>repository/resources/security/wso2carbon.jks</Location>
 <Type>JKS</Type>
 <Password>wso2carbon</Password>
 <KeyPassword>wso2carbon</KeyPassword>
 </KeyStore>
 </p:parameter>
 <p:parameter name="truststore">
 <TrustStore>
 <Location>repository/resources/security/client-truststore.jks</Location>
 <Type>JKS</Type>
 <Password>wso2carbon</Password>
 </TrustStore>
 </p:parameter>
 </p:parameters>
 </inboundEndpoint>
 <sequence xmlns="http://ws.apache.org/ns/synapse" name="TestIn">
 <send receive="receiveSeq">
 <endpoint>
 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
 </endpoint>
 </send>
 </sequence>
 <sequence xmlns="http://ws.apache.org/ns/synapse" name="receiveSeq">
 <send/>
 </sequence>
</definitions>
```

This configuration file `synapse_sample_903.xml` is available in the `<ESB_HOME>/repository/samples` dire

ctory.

## To build the sample

1. Start the ESB with the sample 902 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).  
The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "WSO2 Carbon started in n seconds."
2. Start the Axis2 server. For instructions on starting the Axis2 server, see [Starting the Axis2 server](#).
3. Deploy the back-end service **SimpleStockQuoteService**. For instructions on deploying sample back-end services, see [Deploying sample back-end services](#).

### **Executing the sample**

The sample client used here is the **Stock Quote Client**, which can operate in several modes. For further details on this sample client and its operation modes, see [Stock Quote Client](#).

### **To execute the sample client**

- Run the following command from the <ESB\_HOME>/samples/axis2Client directory, to execute the **Stock Quote Client** in the **Dumb Client Mode**.

```
ant stockquote -Dtrpurl=https://localhost:8081/
```

### **Analyzing the output**

Analyze the output debug messages for the actions in the Dumb Client Mode.

You will see the ESB receiving a message for which the ESB Inbound is set as the ultimate receiver. You will also see the response from the back end in the Client.

## **Sample 904: Inbound Endpoint Kafka Protocol Sample**

- [Introduction](#)
- [Prerequisites](#)
- [Building the sample](#)
- [Executing the sample](#)
- [Analyzing the output](#)

### **Introduction**

This sample demonstrates how one way message bridging from Kafka to HTTP can be done using the inbound kafka endpoint.

### **Prerequisites**

- Download and install [Apache Kafka](#). For more information, see [Apache Kafka documentation](#).
- Copy the following client libraries from the <KAFKA\_HOME>/lib directory to the <ESB\_HOME>/repository/components/lib directory.
  - kafka\_2.9.2-0.8.1.1.jar
  - scala-library-2.9.2.jar
  - zkclient-0.3.jar
  - zookeeper-3.3.4.jar
  - metrics-core-2.2.0.jar
- Run the following command to start the ZooKeeper server:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

You will see the following log:

```
kathees@kathees-ThinkPad-T540p:~/Documents/Kafka/kafka_2.9.2-0.8.1.1$ bin/zookeeper-server-start.sh config/zookeeper.properties
[2015-05-13 15:32:28,044] INFO Reading configuration from: config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2015-05-13 15:32:28,044] WARN Either no config or no quorum defined in config, running in standalone mode (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,055] INFO Reading configuration from: config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2015-05-13 15:32:28,055] INFO Starting server (org.apache.zookeeper.server.ZooKeeperServerMain)
[2015-05-13 15:32:28,059] INFO Server environment:zookeeper.version=3.3.3-1203054, built on 11/17/2011 05:47 GMT (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,059] INFO Server environment:host.name=kathees-ThinkPad-T540p (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,059] INFO Server environment:java.version=1.7.0_55 (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,059] INFO Server environment:java.vendor=Oracle Corporation (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,059] INFO Server environment:java.home=/home/kathees/Documents/Dev-Tool/jdk1.7.0_55/jre (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,059] INFO Server environment:java.class.path=:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1/bin/..../core/build/de
[2015-05-13 15:32:28,059] INFO Server environment:java.class.path=:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1/bin/..../clients/build/libs//kafka-c
[2015-05-13 15:32:28,059] INFO Server environment:java.class.path=:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1/bin/..../contrib/hadoop-consumer/build/libs//ka
[2015-05-13 15:32:28,059] INFO Server environment:java.class.path=:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1/bin/..../lib
[2015-05-13 15:32:28,059] INFO Server environment:java.class.path=:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1/bin/..../libs/kafka_2.9.2-0.8.1.1-javadoc.jar:ho
[2015-05-13 15:32:28,059] INFO Server environment:java.class.path=:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1/bin/..../libs/metrics-core-2.2.0.jar:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1/bin/..../lib
[2015-05-13 15:32:28,059] INFO Server environment:java.class.path=:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1/bin/..../libs/snappy-java-1.0.5.jar:/home/kathees
[2015-05-13 15:32:28,059] INFO Server environment:java.library.path=/usr/java/packages/lib/amd64:/usr/lib64:/lib64:/lib:/usr/lib (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,059] INFO Server environment:java.io.tmpdir=/tmp (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,059] INFO Server environment:java.compiler=<NA> (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,059] INFO Server environment:os.name=Linux (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,059] INFO Server environment:os.arch=amd64 (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,060] INFO Server environment:os.version=3.13.0-32-generic (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,060] INFO Server environment:user.name=kathees (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,060] INFO Server environment:user.home=/home/kathees (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,060] INFO Server environment:user.dir=/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1 (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,064] INFO ticktime set to 3000 (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,064] INFO minSessionTimeout set to -1 (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,064] INFO maxSessionTimeout set to -1 (org.apache.zookeeper.server.ZooKeeperServer)
[2015-05-13 15:32:28,072] INFO binding to port 0.0.0.0/0.0.0.0:2181 (org.apache.zookeeper.server.NIOServerCnxn)
[2015-05-13 15:32:28,080] INFO Reading snapshot /tmp/zookeeper/version-2/snapshot.0 (org.apache.zookeeper.server.persistence.FileSnap)
[2015-05-13 15:32:28,089] INFO Snapshotting: 13 (org.apache.zookeeper.server.persistence.FileTxnSnapLog)
```

- Run the following command to start the Kafka server

```
bin/kafka-server-start.sh config/server.properties
```

You will see the following log:

```
[kathees@kathees-ThinkPad-T540p:~/Documents/Kafka/kafka_2.9.2-0.8.1.1$ bin/kafka-server-start.sh config/server.properties
[2015-05-13 15:38:14,262] INFO Verifying properties (kafka.utils.VerifiableProperties)
[2015-05-13 15:38:14,287] INFO Property broker.id is overridden to 0 (kafka.utils.VerifiableProperties)
[2015-05-13 15:38:14,287] INFO Property log.cleaner.enable is overridden to false (kafka.utils.VerifiableProperties)
[2015-05-13 15:38:14,287] INFO Property log.dirs is overridden to /tmp/kafka-logs (kafka.utils.VerifiableProperties)
[2015-05-13 15:38:14,288] INFO Property log.retention.check.interval.ms is overridden to 60000 (kafka.utils.VerifiableProperties)
[2015-05-13 15:38:14,288] INFO Property log.retention.hours is overridden to 168 (kafka.utils.VerifiableProperties)
[2015-05-13 15:38:14,288] INFO Property log.segment.bytes is overridden to 536870912 (kafka.utils.VerifiableProperties)
[2015-05-13 15:38:14,288] INFO Property num.io.threads is overridden to 8 (kafka.utils.VerifiableProperties)
[2015-05-13 15:38:14,288] INFO Property num.network.threads is overridden to 2 (kafka.utils.VerifiableProperties)
[2015-05-13 15:38:14,288] INFO Property num.partitions is overridden to 2 (kafka.utils.VerifiableProperties)
[2015-05-13 15:38:14,288] INFO Property port is overridden to 9092 (kafka.utils.VerifiableProperties)
[2015-05-13 15:38:14,289] INFO Property socket.receive.buffer.bytes is overridden to 1048576 (kafka.utils.VerifiableProperties)
[2015-05-13 15:38:14,289] INFO Property socket.request.max.bytes is overridden to 104857600 (kafka.utils.VerifiableProperties)
[2015-05-13 15:38:14,289] INFO Property socket.send.buffer.bytes is overridden to 1048576 (kafka.utils.VerifiableProperties)
[2015-05-13 15:38:14,289] INFO Property zookeeper.connect is overridden to localhost:2181 (kafka.utils.VerifiableProperties)
[2015-05-13 15:38:14,300] INFO [Kafka Server 0], starting (kafka.server.KafkaServer)
[2015-05-13 15:38:14,301] INFO [Kafka Server 0], Connecting to zookeeper on localhost:2181 (kafka.server.KafkaServer)
[2015-05-13 15:38:14,308] INFO Starting ZKClient event thread. (org.I0Itec.zkclient.ZkEventThread)
[2015-05-13 15:38:14,328] INFO Client environment:zookeeper.version=3.3.3-1203054, built on 11/17/2011 05:47 GMT (org.apache.zookeeper.ZooKeeper)
[2015-05-13 15:38:14,312] INFO Client environment:host.name=kathees-ThinkPad-T540p (org.apache.zookeeper.ZooKeeper)
[2015-05-13 15:38:14,312] INFO Client environment:java.version=1.7.0_55 (org.apache.zookeeper.ZooKeeper)
[2015-05-13 15:38:14,312] INFO Client environment:java.vendor=Oracle Corporation (org.apache.zookeeper.ZooKeeper)
[2015-05-13 15:38:14,312] INFO Client environment:java.home=/home/kathees/Documents/Dev-Tool/jdk1.7.0_55/jre (org.apache.zookeeper.ZooKeeper)
[2015-05-13 15:38:14,312] INFO Client environment:java.class.path=:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1/bin/./core/build/dependant-libs-2.8.0/*.jar:/home/kathees/8.1.1/bin/../perf/build/libs/kafka-perf_2.8.0*.jar:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1/bin/..clients/build/libs/kafka-client*.jar:/home/kathees/Documents/Kafka/8.1.1/bin/../contrib/hadoop/consumer/build/libs/kafka-hadoop-consumer*.jar:/home/kathees/1.1/bin/..libs/jopt-simple-3.2.jar:/home/kathees/1.2-0.8.1.1-scaladoc.jar:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1/bin/..libs/kafka_2.9.2-0.8.1.1-javadoc.jar:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1-bin/..libs/metrics-core-2.2.0.jar:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1-bin/..libs/snappy-java-1.0.5.jar:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1-bin/..libs/slf4j-api-1.7.2.jar:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1-bin/..libs/zookeeper-3.3.4.jar:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1-bin/..libs/zookeeper-3.3.4.jar:/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1-bin/..core/build/libs/kafka_2.8.0*.jar (org.apache.kaf
[2015-05-13 15:38:14,312] INFO Client environment:user.home=/home/kathees (org.apache.zookeeper.ZooKeeper)
[2015-05-13 15:38:14,313] INFO Client environment:user.dir=/home/kathees/Documents/Kafka/kafka_2.9.2-0.8.1.1 (org.apache.zookeeper.ZooKeeper)
[2015-05-13 15:38:14,313] INFO Initiating client connection, connectString=localhost:2181 sessionTimeout=6000 watcher=org.I0Itec.zkclient.ZkClient@41a7d388 (org.apache.zookeeper.ClientCnxn)
[2015-05-13 15:38:14,321] INFO Opening socket connection to server localhost/127.0.0.1:2181 (org.apache.zookeeper.ClientCnxn)
[2015-05-13 15:38:14,325] INFO Socket connection established to localhost/127.0.0.1:2181, initiating session (org.apache.zookeeper.ClientCnxn)
[2015-05-13 15:38:14,357] INFO Session establishment complete on server localhost/127.0.0.1:2181, sessionid = 0x14d4cb9b37c0000, negotiated timeout = 6000 (org.apache.zookeeper.ClientCnxn)
[2015-05-13 15:38:14,358] INFO zookeeper state changed (SyncConnected) (org.I0Itec.zkclient.ZkClient)
[2015-05-13 15:38:14,412] INFO Found clean shutdown file. Skipping recovery for all logs in data directory '/tmp/kafka-logs' (kafka.log.LogManager)
[2015-05-13 15:38:14,413] INFO Starting log cleanup with a period of 60000 ms. (kafka.log.LogManager)
[2015-05-13 15:38:14,415] INFO Starting log flusher with a default period of 9223372036854775807 ms. (kafka.log.LogManager)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation.
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
[2015-05-13 15:38:14,436] INFO Awaiting socket connections on 0.0.0.0:9092. (kafka.network.Acceptor)
[2015-05-13 15:38:14,437] INFO [Socket Server on Broker 0], Started (kafka.network.SocketServer)
[2015-05-13 15:38:14,492] INFO Will not load MX4J, mx4j-tools.jar is not in the classpath (kafka.utils.Mx4jLoader$)
[2015-05-13 15:38:14,531] INFO 0 successfully elected as leader (kafka.server.ZookeeperLeaderElector)
[2015-05-13 15:38:14,626] INFO Registered broker 0 at path /brokers/ids/0 with address kathees-ThinkPad-T540p:9092. (kafka.utils.ZkUtils$)
[2015-05-13 15:38:14,633] INFO [Kafka Server 0], started (kafka.server.KafkaServer)
[2015-05-13 15:38:14,669] INFO New leader is 0 (kafka.server.ZookeeperLeaderElector$LeaderChangeListener)
```

### **Building the sample**

The XML configuration for this sample is as follows:

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
 name="KAFKAListenerEP"
 sequence="TestIn"
 onError="fault"
 protocol="kafka"
 suspend="false">
<parameters>
 <parameter name="interval">10</parameter>
 <parameter name="consumer.type">highlevel</parameter>
 <parameter name="content.type">application/xml</parameter>
 <parameter name="coordination">false</parameter>
 <parameter name="sequential">false</parameter>
 <parameter name="topics">test</parameter>
 <parameter name="zookeeper.connect">localhost:2181</parameter>
 <parameter name="group.id">test-1</parameter>
</parameters>
</inboundEndpoint>

<sequence xmlns="http://ws.apache.org/ns/synapse" name="TestIn">
 <log level="full"/>
 <drop/>
</sequence>

</definitions>

```

This configuration file `synapse_sample_904.xml` is available in the `<ESB_HOME>/repository/samples` directory.

## To build the sample

- Start the ESB with the sample 904 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).  
The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"

## **Executing the sample**

- Run the following on the Kafka command line to create a topic named `test` with a single partition and only one replica:

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1
--partitions 1 --topic test
```

- Run the following on the Kafka command line to send a message to the Kafka brokers. You can also use the WSO2 ESB Kafka producer connector to send the message to the Kafka brokers.

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
```

```
kathees@kathees-ThinkPad-T540p:~/Documents/Kafka/kafka_2.9.2-0.8.1.1$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#staticLoggerBinder for further details.
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:wsa="http://www.w3.org/2005/08/addressing">
 <soapenv:Body>
 <m0:getQuote xmlns:m0="http://services.samples">
 <m0:request>
 <m0:symbol>IBM</m0:symbol>
 </m0:request>
 </m0:getQuote>
 </soapenv:Body>
</soapenv:Envelope>
```

### Analyzing the output

You will see the following Message content:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:wsa="http://www.w3.org/2005/08/addressing"><soapenv:Body><m0:getQuote
xmlns:m0="http://services.samples">
<m0:request><m0:symbol>IBM</m0:symbol></m0:request></m0:getQuote></soapenv:Body></soap
env:Envelope>
```

The Kafka inbound gets the messages from the Kafka brokers and logs the messages in the ESB

## Sample 905: Inbound HL7 with Automatic Acknowledgement

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

### ***Introduction***

This sample illustrates how the [HL7 Inbound Protocol](#) can be used to receive a simple HL7 message.

### ***Prerequisites***

For a list of prerequisites, see [Prerequisites to Start the ESB Samples](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

## Inbound HL7 Automatic Acknowledgement

```

<definitions xmlns="http://ws.apache.org/ns/synapse">
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
 name="Sample1"
 sequence="main"
 onError="fault"
 protocol="hl7"
 suspend="false">
 <parameters>
 <parameter name="inbound.hl7.AutoAck">true</parameter>
 <parameter name="inbound.hl7.Port">20000</parameter>
 <parameter name="inbound.hl7.TimeOut">3000</parameter>
 <parameter name="inbound.hl7.CharSet">UTF-8</parameter>
 <parameter name="inbound.hl7.ValidateMessage">false</parameter>
 <parameter name="transport.hl7.BuildInvalidMessages">false</parameter>
 </parameters>
</inboundEndpoint>

<sequence name="main">
 <in>
 <log level="full"/>
 <drop/>
 </in>
 <out>
 <send/>
 </out>
 <description>The main sequence for the message mediation</description>
</sequence>

<sequence name="fault">
 <drop/>
</sequence>
</definitions>

```

This configuration file `synapse_sample_905.xml` is available in the `<ESB_HOME>/repository/samples` directory.

### To build the sample

1. Start the ESB with the sample 905 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).  
The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "*WSO2 Carbon started in n seconds.*"
2. Download and install the [HAPI HL7 TestPanel](#).

### ***Executing the sample***

The sample client used here is the **HAPI HL7 TestPanel**.

### **To execute the sample**

- Connect to the port defined in the inbound endpoint (i.e., 20000, which is the value of `inbound.hl7.Port`) using the HAPI HL7 TestPanel.
- Generate and send an HL7 message using the messages dialog frame.

### ***Analyzing the output***

You will see that the ESB receives the HL7 message and logs a serialisation of this message in a SOAP envelope. You will also see that the HAPI HL7 TestPanel receives an acknowledgement.

## Sample 906: Inbound Endpoint MQTT Protocol Sample

- Introduction
- Prerequisites
- Building the sample
  - For consumer ESB
  - For publisher ESB
- Executing the request
- Analyzing the output

### ***Introduction***

This sample demonstrates how the MQTT connector publishes a message on a particular topic and how a MQTT client that is subscribed to that topic receives it.

### ***Prerequisites***

- You need to have the following downloaded.
  - axis2-transport-mqtt-1.0.0.jar
  - mqtt-client-0.4.0.jar
  - mosquitto-1.3.4 (<http://mosquitto.org/>)

### ***Building the sample***

#### **For consumer ESB**

- Copy the following client libraries to the < ESB\_HOME>/repository/components/lib directory.
  - axis2-transport-mqtt-1.0.0.jar
  - mqtt-client-0.4.0.jar
- Add the XML configuration for the inbound endpoint as follows:

**Insequence**

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse" name="test"
sequence="TestIn" onError="TestIn" protocol="mqtt" suspend="false">
 <parameters>
 <parameter name="sequential">true</parameter>
 <parameter name="mqtt.connection.factory">mqttFactory</parameter>
 <parameter name="mqtt.server.host.name">localhost</parameter>
 <parameter name="mqtt.server.port">1883</parameter>
 <parameter name="mqtt.topic.name">esb.test2</parameter>
 <parameter name="mqtt.subscription.qos">2</parameter>
 <parameter name="content.type">application/xml</parameter>
 <parameter name="mqtt.session.clean">false</parameter>
 <parameter name="mqtt.ssl.enable">false</parameter>
 <parameter name="mqtt.subscription.username">elill1</parameter>
 <parameter name="mqtt.subscription.password">el13</parameter>
 <parameter name="mqtt.temporary.store.directory">my</parameter>
 <parameter name="mqtt.blocking.sender">false</parameter>
 </parameters>
</inboundEndpoint>
```

## TestIn

```
<sequence xmlns="http://ws.apache.org/ns/synapse" name="TestIn">
 <log level="full"/>
 <drop/>
</sequence>
```

## For publisher ESB

Run the following command to use the mosquito publisher:

```
mosquitto_pub -h 127.0.0.1 -t esb.test2 -m "<msg><a>Testing123</msg>"
```

## Executing the request

After you perform the required configuration changes,

- Start the ESB and the mosquito publisher.
- Start the mosquito subscriber by executing the following command:

```
mosquitto_sub -h 127.0.0.1 -t esb.test2
```

## Analyzing the output

On the console you will see the following:

```
mosquitto_sub -h 127.0.0.1 -t esb.test2
11:07:53,693 [INFO] RuleEngineConfigDS Successfully registered the Rule Config service
[2015-10-08 11:07:53,727] [INFO] ServiceBusInitializer Starting ESB...
[2015-10-08 11:07:53,738] [INFO] ServiceBusInitializer Initializing Apache Synapse...
[2015-10-08 11:07:53,741] [INFO] SynapseControllerFactory Using Synapse Home : /Users/ellimathaa/Downloads/WSO2/ESB/wso2esb-4.9.0/.
[2015-10-08 11:07:53,741] [INFO] SynapseControllerFactory Using Synapse Home : /Users/ellimathaa/Downloads/WSO2/ESB/wso2esb-4.9.0./repository/deployment/server/synapse-configs/default
[2015-10-08 11:07:53,742] [INFO] SynapseControllerFactory Using Server name : localhost
[2015-10-08 11:07:53,746] [INFO] SynapseControllerFactory The timeout handler will run every : 15s
[2015-10-08 11:07:53,753] [INFO] Axis2SynapseController Initializing Synapse at : Thu Oct 08 11:07:53 IST 2015
[2015-10-08 11:07:53,753] [INFO] Axis2SynapseController Initializing the Synapse configuration from the file system
[2015-10-08 11:07:53,768] [INFO] MultiXMLConfigurationBuilder Building synapse configuration from the synapse artifact repository at : ./repository/deployment/server/synapse-configs/default
[2015-10-08 11:07:53,763] [INFO] XMLConfigurationBuilder Generating the Synapse configuration model by parsing the XML configuration
[2015-10-08 11:07:53,823] [INFO] SynapseConfigurationBuilder Loaded Synapse configuration from the artifact repository at : ./repository/deployment/server/synapse-configs/default
[2015-10-08 11:07:53,823] [INFO] Axis2SynapseController Deploying Axis2 service: Echo
[2015-10-08 11:07:53,841] [INFO] DeploymentInterceptor Deploying Axis2 service: echo [super-tenant]
[2015-10-08 11:07:53,842] [INFO] DeploymentEngine Deploying Web service: Echo.aar - file:/Users/ellimathaa/Downloads/WSO2/ESB/wso2esb-4.9.0/repository/deployment/server/axis2services/Echo.ear
[2015-10-08 11:07:53,859] [INFO] DeploymentInterceptor Deploying Axis2 Version [super-tenant]
[2015-10-08 11:07:53,851] [INFO] Axis2SynapseController Deploying the Synapse service...
[2015-10-08 11:07:53,853] [INFO] Axis2SynapseController Deploying Proxy services...
[2015-10-08 11:07:53,853] [INFO] Axis2SynapseController Deploying TestSources...
[2015-10-08 11:07:53,853] [INFO] Axis2SynapseController Deploying Test
[2015-10-08 11:07:53,871] [WARN] MqttConnectionFactory Default value is used for the parameter : mqtt.client.id
[2015-10-08 11:07:53,871] [WARN] MqttConnectionFactory Default value is used for the parameter : mqtt.reconnection.interval
[2015-10-08 11:07:53,872] [INFO] MqttListener Mqtt inbound endpoint test initializing ...
[2015-10-08 11:07:53,872] [INFO] MqttConnectionManager Mqtt Connection Manager initialized ...
[2015-10-08 11:07:53,887] [INFO] MqttConnectionFactory Successfully created MQTT client
[2015-10-08 11:07:53,888] [INFO] InboundOneTimeTriggerRequestProcessor Starting the inbound endpoint test, with coordination true. Type : MQTT--SYNAPSE_INBOUND_ENDPOINT
[2015-10-08 11:07:53,894] [INFO] ServerManager Server ready for processing...
[2015-10-08 11:07:53,901] [INFO] MediationStatisticsComponent MediationStatisticsReporter is Disabled
[2015-10-08 11:07:53,946] [INFO] MediationStatisticsComponent Can't register an observer for mediationStatisticsStore. If you have disabled StatisticsReporter, please enable it in the Carbon.xml
[2015-10-08 11:07:54,486] [INFO] PassThroughHttpsSSLListener Starting Pass-through HTTPS Listener...
[2015-10-08 11:07:54,586] [INFO] PassThroughListeningOReactorManager Pass-through HTTPS Listener started on 0.0.0.0:8243
[2015-10-08 11:07:54,586] [INFO] PassThroughHttpsSSLListener Starting Pass-through HTTPS Listener...
[2015-10-08 11:07:54,598] [INFO] PassThroughListeningOReactorManager Pass-through HTTP Listener started on 0.0.0.0:8280
[2015-10-08 11:07:54,515] [INFO] NioSelectorPool Using a shared selector for servlet write/read
[2015-10-08 11:07:54,857] [INFO] NioSelectorPool Using a shared selector for servlet write/read
[2015-10-08 11:07:54,857] [INFO] NioSelectorPool Using a shared selector for servlet write/read
[2015-10-08 11:07:54,948] [INFO] NTasksTasManager Initializing task manager, Tenant [-1:234]
[2015-10-08 11:07:54,969] [INFO] AbstractQuartzTaskManager Task scheduled: [-1:234][ESB_TASK][test-MOTT--SYNAPSE_INBOUND_ENDPOINT]
[2015-10-08 11:07:54,969] [INFO] NTasksTasManager Scheduled task [NTASKS:-1234::test-MOTT--SYNAPSE_INBOUND_ENDPOINT]
[2015-10-08 11:07:55,430] [INFO] JMXServiceManager JMX Service URL : service:jmx:rmi:///localhost:9999/jmxrmi
[2015-10-08 11:07:55,436] [INFO] StartupFinalizerServiceComponent Server : WSO2 Enterprise Service Bus-4.9.0
[2015-10-08 11:07:55,437] [INFO] StartupFinalizerServiceComponent WSO2 Carbon started in 19 sec
[2015-10-08 11:07:55,437] [INFO] StartupFinalizerServiceComponent WSO2 Carbon started in 19 sec
[2015-10-08 11:07:55,988] [INFO] MqttConnectionConsumer Connected to the remote server
[2015-10-08 11:07:59,890] [INFO] MqttSyncCallback Received Message: Topic: esb.test2
[2015-10-08 11:07:59,915] [INFO] LogMediator To : MessageID:urn:uuid:6C3F6697248169A7B1444282679896, Direction: request, Envelope: <?xml version='1.0' encoding='utf-8'?><soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Body><a>Testing123</soapenv:Body></soapenv:Envelope>
```

```
esb:~ elilmatha$ mosquitto_sub -h 127.0.0.1 -t esb.test2
<msg>
<a>Testing 123
</msg>
```

## Sample 907: Inbound Endpoint RabbitMQ Protocol Sample

- Introduction
- Prerequisites
- Building the sample
- Executing the sample
- Analyzing the output

### ***Introduction***

This sample demonstrates how one way message bridging from RabbitMQ to HTTP can be done using the inbound RabbitMQ endpoint.

### ***Prerequisites***

- Download and install RabbitMQ. For more information, see [RabbitMQ documentation](#).

### ***Building the sample***

The XML configuration for this sample is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <taskManager provider="org.wso2.carbon.mediation.ntask.NTaskTaskManager"/>
 <sequence name="TestIn">
 <log level="full"/>
 <drop/>
 </sequence>
 <inboundEndpoint name="test" onError="fault" protocol="rabbitmq"
 sequence="TestIn" suspend="false">
 <parameters>
 <parameter name="sequential">true</parameter>
 <parameter name="coordination">true</parameter>
 <parameter name="rabbitmq.server.host.name">localhost</parameter>
 <parameter name="rabbitmq.server.port">5672</parameter>
 <parameter
 name="rabbitmq.connection.factory">AMQPConnectionFactory</parameter>
 <parameter name="rabbitmq.server.user.name">guest</parameter>
 <parameter name="rabbitmq.server.password">guest</parameter>
 <parameter name="rabbitmq.queue.name">queue</parameter>
 <parameter name="rabbitmq.exchange.name">exchange</parameter>
 </parameters>
 </inboundEndpoint>
 </definitions>
```

This configuration file `synapse_sample_907.xml` is available in the `<ESB_HOME>/repository/samples` dire

ctory.

## To build the sample

- Start the ESB with the sample 907 configuration. For instructions on starting a sample ESB configuration, see [Starting the ESB with a sample configuration](#).  
The operation log keeps running until the server starts, which usually takes several seconds. Wait until the server has fully booted up and displays a message similar to "WSO2 Carbon started in n seconds."

## **Executing the sample**

- Use the following java client to publish a request to the RabbitMQ broker.

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
factory.setUsername("guest");
factory.setPassword("guest");
factory.setPort(5672);
Channel channel = null;
Connection connection = factory.newConnection();
channel = connection.createChannel();
channel.queueDeclare("queue", false, false, false, null);
channel.exchangeDeclare("exchange", "direct", true);
channel.queueBind("queue", "exchange", "route");

// The message to be sent
String message = "<m:placeOrder xmlns:m=\"http://services.samples\">>" +
 "<m:order>" +
 "<m:price>100</m:price>" +
 "<m:quantity>20</m:quantity>" +
 "<m:symbol>RMQ</m:symbol>" +
 "</m:order>" +
 "</m:placeOrder>";

// Populate the AMQP message properties
AMQP.BasicProperties.Builder builder = new AMQP.BasicProperties().builder();
builder.contentType("application/xml");

// Publish the message to exchange
channel.basicPublish("exchange", "queue", builder.build(), message.getBytes());
```

## Analyzing the output

You will see the following Message content:

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Body><m:placeOrder
xmlns:m="http://services.samples"><m:order><m:price>100</m:price><m:quantity>20</m:qua
ntity><m:symbol>RMQ</m:symbol></m:order></m:placeOrder></soapenv:Body></soapenv:Envelo
pe>
```

The RabbitMQ inbound endpoint gets the messages from the RabbitMQ broker and logs the messages in the ESB.

# FAQ

- General ESB Questions
  - What is WSO2 ESB?
  - What makes WSO2 ESB unique?
  - What is the open source license of WSO2 ESB?
  - How do I get WSO2 ESB?
  - Does WSO2 ESB have performance comparisons with other popular ESBs?
  - Is their commercial support available for WSO2 ESB?
  - Can I get involved with the ESB development activities?
- Core Carbon Questions
  - What are the technologies used underneath WSO2 ESB?
  - What are the minimum requirements to run WSO2 ESB?
  - What should I do if I get the following error when I try to start WSO2 ESB? [ERROR] CARBON is supported only on JDK 1.7 and 1.8
  - What makes WSO2 ESB fast and high-performant?
  - Does WSO2 ESB support clustering?
  - In what occasions do I have to deploy a third-party library into WSO2 ESB?
  - How do I deploy a third-party library into the ESB?
  - Can I extend the management console UI to add custom UIs?
  - Can I use a primary user store to manage and authenticate WSO2 ESB users?
  - Does WSO2 ESB interop with a .net client/service?
  - Does WSO2 ESB support hessian binary web service protocol
  - I don't want some of the features that come with WSO2 ESB. Can I get rid of them?
  - Can I add support for other language for the management console ?
- Mediation-Related Questions
  - What is a proxy service?
  - Does WSO2 ESB have support for transactions?
  - Where do I have to put my custom mediators in WSO2 ESB?
  - Where does WSO2 ESB load its configuration from?
  - Can I get the ESB to load mediation configuration from the registry?
  - What is a mediator?
  - What is a sequence?
  - What is an Endpoint?
  - Can Endpoints perform error handling?
  - What are local entries?
  - How can a sequence refer to another sequence?
  - How does dynamic endpoints and sequences work?
  - How can I change the endpoint dynamically?
  - I have heard of dynamic sequences and endpoints. Are there also dynamic proxy services?
  - How can I create load balance endpoint templates?

- How can I implement a persistence mechanism within my mediation chain?
- I have some dynamic sequences which are loaded from the registry. What will happen to my ESB if the registry goes down or registry becomes unusable for a while?
- What is the recommended approach for dealing with \*.back files that are created when there is an error in an ESB artifact?
- How can you preserve a large CDATA block in WSO2 ESB?
- Transports-Related Questions
  - What are the transports supported by the WSO2 ESB?
  - Do I need an external JMS broker for the JMS transport?
  - Does WSO2 ESB support AMQP?
  - How can I change the host name in WSO2 ESB?
  - I have several JMS connection factories defined under the JMS transport receiver in the axis2.xml. How can I get a particular proxy service to receive messages from one of those connection factories?
  - Does WSO2 ESB support two way JMS scenario (request/response) ?
  - What are the JMS brokers that work with WSO2 ESB?
  - What is the NHTTP transport?
  - What is the underlying HTTP library used by the NHTTP transport?
  - How can I tune up the NHTTP transport?
  - Where is the general configuration of NHTTP transport?
  - How do I process files with the ESB?
  - How can I prevent an exception getting logged every 1-2 milliseconds in ESB when the JMS Broker (Active MQ) is down?
  - How can we add JMS correlation IDs to the messages in a request queue and how does it work?
- Production-Related Questions
  - Where do I keep the WSDLs required by my proxy services?
  - What is a graceful shut down?
  - I already have a WSO2 Governance Registry instance that contains my organization's SOA metadata. Can I get WSO2 ESB to use that registry instance as the metadata store?
  - I need to setup a cluster of ESB instances. How can I share the same configuration among all the ESB nodes?
  - How can I get the <ESB\_HOME>/tmp directory cleared?
- Deployment Questions
  - What are the Java versions that support running WSO2 ESB?
  - What are the minimum artifacts required to deploy WSO2 ESB?
  - How can I disable the management console?
  - Does WSO2 ESB support application server deployments?
  - How can I deploy a custom task?
  - How do I embed a third-party registry with WSO2 ESB?
  - What is the database management system used in WSO2 ESB?
  - How can I change the memory allocation for the WSO2 ESB?
- Monitoring Questions

- Does ESB supports JMX monitoring?
- Does ESB allows custom statistics collection?
- What is the logging framework used in WSO2 ESB?
- I have a sequence which does not behave the way I want it to. How can I find out what's wrong with it?
- My WSO2 ESB instance is receiving messages. But why don't I see any statistics on the mediation statistics page?
- My ESB instance is behaving differently for certain input messages. What are the tools provided to debug the issue?
- What triggers the exception : com.ctc.wstx.exc.WstxIOException: Invalid UTF-8 start byte 0x89 (at char #1, byte #-1)?

## General ESB Questions

### **What is WSO2 ESB?**

WSO2 ESB is a fast, lightweight, open source Enterprise Service Bus implementation. It supports message routing, inter-mediation, transformation, logging, load balancing, fail over routing, task scheduling, eventing and much more.

### **What makes WSO2 ESB unique?**

WSO2 ESB is fast and able to handle thousands of concurrent connections with constant memory usage. It comes with a rich set of mediators to support almost any integration scenario out of the box. It is also easily extensible and highly customizable. The feature-rich admin console makes it very easy to configure, control and monitor the ESB runtime.

### **What is the open source license of WSO2 ESB?**

It is released under Apache Software License Version 2.0, one of the most business-friendly licenses available today.

### **How do I get WSO2 ESB?**

Go to <http://wso2.com/products/enterprise-service-bus/> and download the binary distribution, or click the link at the bottom of the page to access the source code. WSO2 ESB is distributed completely free of charge.

### **Does WSO2 ESB have performance comparisons with other popular ESBs?**

Yes, the results of several rounds of performance comparisons are available in the following article: <http://wso2.org/library/articles/2013/01/esb-performance-65>.

### **Is their commercial support available for WSO2 ESB?**

We are committed to ensuring that your enterprise middleware deployment is completely supported **from evaluation to production**. Our unique approach ensures that all support leverages our open development methodology and is provided by the very same engineers who build the technology. For more details and to take advantage of this unique opportunity, see [WSO2 Support](#).

### **Can I get involved with the ESB development activities?**

Not only you are allowed, but it is encouraged. You can start by subscribing to [carbon-dev@wso2.org](mailto:carbon-dev@wso2.org) and [architecte@wso2.org](mailto:architecte@wso2.org) mailing lists. Feel free to provide ideas, feedback and help make our code better. You can also report bugs on <https://wso2.org/jira/browse/ESBJAVA> and submit patches. For more information on our mailing lists, refer to <http://wso2.org/mail>.

## Core Carbon Questions

### ***What are the technologies used underneath WSO2 ESB?***

WSO2 ESB is built on top of WSO2 Carbon, an OSGi based components framework for SOA. It uses Apache Synapse (<http://synapse.apache.org>) as the underlying mediation engine. Java is the primary programming language used to develop WSO2 ESB.

### ***What are the minimum requirements to run WSO2 ESB?***

Refer to [ESB system requirements](#).

### ***What should I do if I get the following error when I try to start WSO2 ESB?***

[ERROR] CARBON is supported only on JDK 1.7 and 1.8

You need to have Oracle JDK 1.7.\*/1.8.\* installed. You cannot start WSO2 ESB with Oracle JDK 1.6.\* or lower. For more information, see [Installation Prerequisites](#).

### ***What makes WSO2 ESB fast and high-performant?***

The mediation core of WSO2 ESB is designed to be completely asynchronous, non-blocking and streaming. It comes with a non-blocking HTTP transport adapter based on Apache HTTP Core-NIO, which is capable of handling a large number of concurrent connections. WSO2 ESB also uses Apache AXIOM, a StAX based XML infoset model, to process XML and SOAP. This enables WSO2 ESB to stream messages through the mediation engine without having to invoke slow XML processing routines for each and every message.

### ***Does WSO2 ESB support clustering?***

Yes, it supports clustered deployment. WSO2 ESB uses the Apache Axis2 clustering framework to support the following two clustering schemes:

- Multicast based clustering
- Well known addressing based clustering

### ***In what occasions do I have to deploy a third-party library into WSO2 ESB?***

When enabling a new transport you will have to deploy the libraries required by the transport implementation (eg: When enabling JMS you need to deploy the client libraries required to connect to your JMS broker)

When adding a custom mediator or a task, you may have to deploy the dependencies required by the custom code.

### ***How do I deploy a third-party library into the ESB?***

Copy any third-party jars into \$ESB\_HOME/repository/components/lib directory and restart the server. WSO2 ESB converts those jars to OSGi bundles and place them in <ESB\_HOME>/repository/components/dropins directory.

### ***Can I extend the management console UI to add custom UIs?***

Yes, you can extend the management console easily by writing a custom UI component and simply deploying the OSGi bundle.

### ***Can I use a primary user store to manage and authenticate WSO2 ESB users?***

Yes you can connect WSO2 ESB with any primary user store implementation. The user store could be LDAP based, JDBC based or a custom developed user store. For more information, refer to [Managing Users, Roles and Permissions](#) in the WSO2 Administration Guide.

### ***Does WSO2 ESB interop with a .net client/service?***

Yes, it does with both. WSO2 ESB can be easily configured to bridge Java services with .NET clients and .NET services with Java clients.

***Does WSO2 ESB support hessian binary web service protocol?***

Yes, it does.

***I don't want some of the features that come with WSO2 ESB. Can I get rid of them?***

Yes, WSO2 ESB is lean and allows you to maintain just the right set of features required at the time. Simply login to the ESB management console and uninstall any unwanted features as described in [Uninstalling Features](#) in the WSO2 Administration Guide.

***Can I add support for other language for the management console ?***

Yes, WSO2 ESB comes with a UI framework which supports i18n (internationalization) which lets you to use a language of your choice for the management console.

---

## Mediation-Related Questions

***What is a proxy service?***

A proxy service is a virtual service hosted on the ESB. It can accept requests from service clients, just like a real Web service. A proxy service can process requests and forward them to an actual Web service (back-end service) to be further processed. The responses coming back from the back-end service can be routed back to the original client. Proxy services are mostly used to expose an existing service over a different transport, format or QoS configuration.

***Does WSO2 ESB have support for transactions?***

Yes, it has support for distributed transaction, local JMS transactions, and distributed JMS transactions.

***Where do I have to put my custom mediators in WSO2 ESB?***

In the current version, you can place any non-OSGi custom mediators and class mediators in <ESB\_HOME>/repository/components/libs and OSGi bundles in <ESB\_HOME>/repository/components/dropins.

***Where does WSO2 ESB load its configuration from?***

WSO2 ESB reads its mediation configuration from a set of XML files located in the <ESB\_HOME>/repository/deployment/server/synapse-configs/<node> directory. XML files are written in the Synapse configuration language. Any changes done to the configuration through the UI are saved back to the configuration files. (Configuration is also written to the registry by default).

***Can I get the ESB to load mediation configuration from the registry?***

Yes you can. The necessary steps are given in section, [Working with the Registry](#).

***What is a mediator?***

A mediator is the basic message processing unit in the ESB. A mediator can take a message, carry out some predefined actions on it and output the modified message. WSO2 ESB ships with a range of mediators capable of carrying out various tasks on input messages.

- Log mediator - Logs the input message.
- Send mediator - Sends the input message to a given endpoint.
- XSLT mediator - Transforms the input message using a given XSLT.

***What is a sequence?***

A sequence is an ordered list of mediators (a mediator chain). When a sequence is given a message, it will go through all the mediators in the sequence. A sequence can also handover messages to other sequences.

### **What is an Endpoint?**

An endpoint is the entry point to a service, process, or a queue or topic destination. It defines the destination for a message.

### **Can Endpoints perform error handling?**

Yes, see [Endpoint Error Handling](#).

### **What are local entries?**

Local entries can be used to hold various configuration elements required by sequences and proxy services. Usually, they are used to hold WSDLs, XSDs, XSLT files etc. A local entry can contain XML content as well as plain text content and configured to load content from a remote file too.

### **How can a sequence refer to another sequence?**

To refer to a sequence named foo use the sequence mediator as follows:

```
<sequence key="foo" />
```

You can also refer to dynamic sequences saved in the registry by specifying the registry key to the sequence resource as follows:

```
<sequence key="gov:/dev/sequences/foo" />
```

The above sequence mediator will load the sequence configuration from the resource at /dev/sequences/foo in the governance registry.

### **How does dynamic endpoints and sequences work?**

Dynamic endpoints and sequences are just XML configuration bits saved in the registry. These configurations are loaded to the mediation configuration at runtime. Once loaded they will be cached for a specified duration in the memory. Once the cache is expired, it will be loaded again from the registry. Therefore, changes done to dynamic sequences and endpoints at runtime will take effect once the ESB reloads them after a cache timeout.

To use dynamic sequences and endpoints, the mediation registry must be included in the mediation configuration as follows:

```
<sequence key="foo" />
<registry xmlns="http://ws.apache.org/ns/synapse"
provider="org.wso2.carbon.mediation.registry.WSO2Registry">
 <parameter name="cachableDuration">15000</parameter>
</registry>
```

This configuration can be found in a file named registry.xml in \$ESB\_HOME/repository/conf/synapse-config directory by default. Note how the cache duration is set to 15000 ms (15s). This duration can be reduced or extended as necessary by editing the registry.xml file or through the configuration source editor in the UI.

### **How can I change the endpoint dynamically?**

If you want to change the endpoint dynamically per each message, use a header mediator to calculate the new 'To' address.

```
<header name="To" expression="XPath to create the address dynamically"/>
```

**I have heard of dynamic sequences and endpoints. Are there also dynamic proxy services?**

WSO2 ESB does not support dynamic proxy services. But a proxy service may use a dynamic endpoint as the target endpoint and dynamic sequences as the in sequence and the out sequence. This effectively makes the entire proxy service dynamic.

**How can I create load balance endpoint templates?**

Load balance endpoint templates cannot be created in the user interface at present. However, you can create them using the source view as follows.

1. Open the ESB management console and click **Templates** under **Service Bus** in the **Main** tab.
2. Click **Add Endpoint Template** in the **Templates** page. Then click one one of the endpoint template types that will be displayed (e.g., **Address Endpoint Template**).
3. Enter a name for the new template in the template page (**Address Endpoint Template** in this example) and click **Switch to source view**.
4. Enter the following configuration in the source view.

```
<template xmlns="http://ws.apache.org/ns/synapse" name="LBEndpointTemplate">
<parameter name="ep1Url"/> <parameter name="ep2Url"/> <parameter name="name"/>
<parameter name="uri"/>
<endpoint name="$name">
<loadbalance algorithm="org.apache.synapse.endpoints.algorithms.RoundRobin">
<endpoint name="ep1"> <address uri="$ep1Url" /> </endpoint>
<endpoint name="ep2"> <address uri="$ep2Url" /> </endpoint>
</loadbalance> </endpoint></template>
```

5. Click **Save & Close**. You will see the load balance endpoint template you created in the **Templates** window. This template can be used to create load balance endpoints as shown in the configuration below.

```
<endpoint xmlns="http://ws.apache.org/ns/synapse" template="LBEndpointTemplate"
name="LBSenderTmplEndpoint" uri="">
<parameter name="ep1Url"
value="http://localhost:9000/services/SimpleStockQuoteService" /> <parameter
name="ep2Url" value="http://localhost:9001/services/SimpleStockQuoteService" />
</endpoint>
```

**How can I implement a persistence mechanism within my mediation chain?**

You have two options.

- Use dbreport and dblookup mediators to persist the message into a database and look it up later from the database.
- Use JMS transport in which you can use a JMS broker to persist the message.

**I have some dynamic sequences which are loaded from the registry. What will happen to my ESB if the registry goes down or registry becomes unusable for a while?**

If the dynamic sequences are loaded to the memory at least once, the ESB will continue to use the cached version of the sequence, as long as the registry is unreachable. A warning will be logged every time ESB attempts to load the sequence from the registry and fails. As soon as the registry comes back on-line, ESB will load the sequence from the registry.

**What is the recommended approach for dealing with \*.back files that are created when there is an error in an ESB artifact?**

You can delete the \*.back file manually from the file system once you fix the error in the artifact. If an artifact fails during the process of creating it via the management console, the artifact can be deleted via the management console and the \*.back file that is created will be deleted automatically.

**Note**

If you upload an erroneous artifact through a CAR file, a \*.back file will not be created even if the deployment fails.

**How can you preserve a large CDATA block in WSO2 ESB?**

In WSO2 ESB, all message that need to be built are parsed using AXIOM as the object model which relies on a standard StAX implementation library named Woodstox. When it comes to parsing CDATA sections there are few ambiguities in the StAX specification which result in multiple interpretations of what to do when non-coalescing mode is activated (`javax.xml.stream.isCoalescing=false`). What Woodstox adheres to when it comes to large CDATA sections is to break them up to several pieces based on the buffer size. This shouldn't be a problem because most clients should be able to handle multiple consecutive CDATA sections.

However when it becomes a problem, Woodstox can be configured to increase the size of the text segment it reads. By setting this value to the maximum Integer value, Woodstox will not break up the CDATA sections until it reaches the configured value.

To configure this, add the following entry to `<ESB_HOME>/XMLInputFactory.properties` file.

```
com.ctc.wstx.minTextSegment=2147483647
```

## Transports-Related Questions

For full details on transports, see [Working with Transports](#).

**What are the transports supported by the WSO2 ESB?**

For a complete list, see [ESB Transports](#).

**Do I need an external JMS broker for the JMS transport?**

Yes, WSO2 ESB requires an external JMS broker like Apache ActiveMQ (<http://activemq.apache.org>).

**Does WSO2 ESB support AMQP?**

WSO2 ESB supports AMQP through its JMS transport. The JMS transport can be configured to connect to an AMQP broker as if it was connecting to a JMS broker. This functionality has been tested with Apache Qpid (<http://qpid.apache.org>) and RabbitMQ.

**How can I change the host name in WSO2 ESB?**

Edit parameter bind-address in http/https transport receiver to change the host name of WSO2 ESB. This hostname will be displayed on service endpoints and generated WSDLs.

**I have several JMS connection factories defined under the JMS transport receiver in the axis2.xml. How can I get a particular proxy service to receive messages from one of those connection factories?**

You can specify the preferred connection factory by adding the following parameter to the proxy service configuration.

```
<parameter name="transport.jms.ConnectionFactory">MyConnectionFactory</parameter>
```

Replace 'MyConnectionFactory' with the name of the appropriate connection factory. If the above parameter is not specified, proxy service will bind to the connection factory named 'default'.

#### **Does WSO2 ESB support two way JMS scenario (request/response) ?**

Yes, refer to [sample 264](#), which demonstrates exactly the JMS request/response scenario.

#### **What are the JMS brokers that work with WSO2 ESB?**

Any JMS broker that provides JNDI support can be integrated with WSO2 ESB. The default configurations are for Apache ActiveMQ. WSO2 ESB has also been tested with IBM Websphere MQ, Swift MQ and JBOSS MQ.

#### **What is the NHTTP transport?**

This is the default HTTP transport used by WSO2 ESB. NHTTP stands for non-blocking HTTP. NHTTP transport uses the Java Non-blocking I/O API. This allows the NHTTP transport to scale into handling hundreds of connections without blocking the threads. The server worker threads used by the NHTTP transport do not get blocked on I/O until the ESB receives responses for the already forwarded requests. Therefore WSO2 ESB can accept more concurrent connections and requests than most HTTP server products.

#### **What is the underlying HTTP library used by the NHTTP transport?**

NHTTP transport uses the Apache Http Core NIO library (<http://hc.apache.org/httpcomponents-core-ga/httpcore-nio>) underneath. This library provides low-level I/O handling and HTTP-level detail handling.

#### **How can I tune up the NHTTP transport?**

There is a file called nhttp.properties in \$ESB\_HOME/repository/conf directory. User can change various configuration parameters like number of threads and maintain alive connections through this file. For more information, refer to [Configuring nhttp.properties](#).

#### **Where is the general configuration of NHTTP transport?**

The NHTTP transport configuration has two parts and they are transportReceiver and transportSender. Both configuration are located in \$ESB\_HOME\repository\conf\axis2\axis2.xml file.

#### **How do I process files with the ESB?**

Use the [VFS transport](#) for file processing. For an example, see [Sample 271: File Processing](#).

#### **How can I prevent an exception getting logged every 1-2 milliseconds in ESB when the JMS Broker (Active MQ) is down?**

You can add failover: `tcp://localhost:61616` to `java.naming.provider.url` of axis.xml JMSListener configuration as follows:

```

<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
 <parameter name="myTopicConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">failover:tcp://localhost:61616</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">TopicConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
 </parameter>

 <parameter name="myQueueConnectionFactory" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">failover:tcp://localhost:61616</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>

 <parameter name="default" locked="false">
 <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
 <parameter name="java.naming.provider.url"
locked="false">failover:tcp://localhost:61616</parameter>
 <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
 <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
 </parameter>
</transportReceiver>

```

When the configuration is done as shown above, the attempted reconnection to Active MQ will not be logged in the log file.

#### **How can we add JMS correlation IDs to the messages in a request queue and how does it work?**

In order to make correlation IDs to the messages in the request queue:

1. Add the following property within the `in sequence` of the proxy service.

```
<property name="JMS_COORELATION_ID" value="1234" scope="axis2"/>
```

2. Make a response with same correlation ID (`message_properties.correlation := '1234'`). Your request and response messages will be matched using this correlation ID.

## Production-Related Questions

### **Where do I keep the WSDLs required by my proxy services?**

You can keep them as local entries. But it is recommended to keep them in the registry instead, for easier and better management.

### **What is a graceful shut down?**

When a graceful shut down is initiated, the ESB will continue to serve the accepted requests but stops processing new ones.

**I already have a WSO2 Governance Registry instance that contains my organization's SOA metadata. Can I get WSO2 ESB to use that registry instance as the metadata store?**

WSO2 ESB integrates with WSO2 Governance Registry out of the box. For steps involved in remote registry configuration, refer to [Working with the Registry](#).

**I need to setup a cluster of ESB instances. How can I share the same configuration among all the ESB nodes?**

Put the configuration into the registry and point all ESB instances to that registry instance.

**How can I get the <ESB\_HOME>/tmp directory cleared?**

Based on the HouseKeeping task for carbon.xml, if the <AutoStart> property is set to true, all files that are more than 30 minutes old will be cleared every 10 minutes.

If you are running the ESB on Linux, you can also write a cron job to handle the clearing of contents within the <ESB\_HOME>/tmp directory.

---

## Deployment Questions

**What are the Java versions that support running WSO2 ESB?**

You can run WSO2 ESB on JDK 1.7.\* and JDK 1.8.\*. For more information, see [Tested JDks](#).

**What are the minimum artifacts required to deploy WSO2 ESB?**

Refer to [Installation Prerequisites](#).

**How can I disable the management console?**

You can uninstall all the UI components using the feature manager, which will disable the management console.

**Does WSO2 ESB support application server deployments?**

Yes it has been tested on JBoss, Weblogic, Websphere and Tomcat.

**How can I deploy a custom task?**

Put your non-OSGi task jar into \$ESB\_HOME/repository/components/lib directory, which will be deployed automatically. For information on writing a custom task, refer to [Writing Tasks](#).

**How do I embed a third-party registry with WSO2 ESB?**

You need to write a class implementing the org.apache.synapse.registry.Registry interface to integrate the third party registry to WSO2 ESB.

**What is the database management system used in WSO2 ESB?**

WSO2 ESB ships with an embedded H2 database. However any database management system can be plugged into the ESB via JDBC. The relevant database configurations are available in repository/conf/registry.xml and repository/cong/user-mgt.xml files. WSO2 ESB has been tested with MySQL, Oracle, MSSQL and PostgreSQL databases. For information, refer to [Setting up the Physical Database](#).

**How can I change the memory allocation for the WSO2 ESB?**

The memory allocation setting are in the wso2server.sh. You can change the memory allocation settings by changing the following configuration.

-Xms256m -Xmx512m -XX:MaxPermSize=128m

---

## Monitoring Questions

### **Does ESB supports JMX monitoring?**

Yes it support JMX monitoring. Users can use the JConsole for simple monitoring and use JMX clients for custom monitoring. For more information, refer to [Monitoring the ESB](#).

### **Does ESB allows custom statistics collection?**

Users can write and plug their own statistics collection mechanisms to the ESB. This allows users to report statistics to their own statistics collection systems.

### **What is the logging framework used in WSO2 ESB?**

WSO2 ESB uses Apache Log4J over Apache Commons Logging as the logging framework. Logging configuration is loaded from the log4j.properties file in \$ESB\_HOME/repository/conf directory. The UI also allows configuring logging at runtime. For instructions, refer to [Logging](#). The generated server logs can be found in the \$ESB\_HOME/repository/logs directory.

### **I have a sequence which does not behave the way I want it to. How can I find out what's wrong with it?**

Login to the ESB management console and go to the 'sequence management' page. Enable tracing for the sequence you want to debug. Send a few messages to the sequence and go to the system logs viewer in the UI (or open the wso2-esb.log file in \$ESB\_HOME/repository/logs). Go through the generated trace logs and locate the problem.

### **My WSO2 ESB instance is receiving messages. But why don't I see any statistics on the mediation statistics page?**

You need to enable statistics on sequences, endpoints, proxy services for the ESB to collect statistics on them. By default, WSO2 ESB does not collect statistics on anything to maintain minimum overhead. Simply enable statistics for required items using the management console UI and the mediation statistics page will start to get updated.

### **My ESB instance is behaving differently for certain input messages. What are the tools provided to debug the issue?**

You can use the SOAP tracer in the UI to capture and monitor actual content of the incoming messages. Enable tracing on the appropriate sequences to trace the flow of messages through the ESB. ESB also comes with Apache TCPMon which is a simple but extremely useful tool for monitoring message flows.

### **What triggers the exception : com.ctc.wstx.exc.WstxIOException: Invalid UTF-8 start byte 0x89 (at char #1, byte #-1)"?**

This exception occurs when the <ESB\_HOME>/repository/conf/axis2/axis2.xml file does not define any builders and/or formatters that can read the content type of the request received by the ESB. Update the axis2.xml file with a message builder and formatter that can be used to read the content type of the relevant request in order to overcome this issue. See [Configuring Message Relay](#) for instructions on how to configure a message builder and a formatter as required. See [Working with Message Builders and Formatters](#) for further information on how to use builders and formatters.