



Red Hat Fuse 7.5

Apache CXF Development Guide

Develop applications with Apache CXF Web services

Red Hat Fuse 7.5 Apache CXF Development Guide

Develop applications with Apache CXF Web services

Legal Notice

Copyright © 2020 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

Guide to developing Web services using Apache CXF.

Table of Contents

PART I. WRITING WSDL CONTRACTS	30
CHAPTER 1. INTRODUCING WSDL CONTRACTS	31
1.1. STRUCTURE OF A WSDL DOCUMENT	31
Overview	31
The logical part	31
The concrete part	31
1.2. WSDL ELEMENTS	31
1.3. DESIGNING A CONTRACT	32
CHAPTER 2. DEFINING LOGICAL DATA UNITS	33
2.1. INTRODUCTION TO LOGICAL DATA UNITS	33
2.2. MAPPING DATA INTO LOGICAL DATA UNITS	33
Overview	33
Available type systems for defining service data units	33
XML Schema as a type system	33
Considerations for creating your data units	34
2.3. ADDING DATA UNITS TO A CONTRACT	34
Overview	34
Procedure	34
2.4. XML SCHEMA SIMPLE TYPES	35
Overview	35
Entering simple types	35
Supported XSD simple types	35
2.5. DEFINING COMPLEX DATA TYPES	36
2.5.1. Defining data structures	37
Overview	37
Complex type varieties	38
Defining the parts of a structure	38
Defining attributes	39
2.5.2. Defining arrays	40
Overview	40
Complex type arrays	40
SOAP arrays	40
2.5.3. Defining types by extension	41
2.5.4. Defining types by restriction	42
Overview	42
Specifying the base type	42
Defining the restrictions	43
Example	43
2.5.5. Defining enumerated types	43
Overview	43
Defining an enumeration in XML Schema	44
Example	44
2.6. DEFINING ELEMENTS	44
CHAPTER 3. DEFINING LOGICAL MESSAGES USED BY A SERVICE	46
OVERVIEW	46
MESSAGES AND PARAMETER LISTS	46
MESSAGE DESIGN FOR INTEGRATING WITH LEGACY SYSTEMS	46
MESSAGE DESIGN FOR SOAP SERVICES	46
MESSAGE NAMING	47

MESSAGE PARTS	47
EXAMPLE	48
CHAPTER 4. DEFINING YOUR LOGICAL INTERFACES	50
OVERVIEW	50
PROCESS	50
PORT TYPES	50
OPERATIONS	50
OPERATION MESSAGES	50
RETURN VALUES	51
EXAMPLE	52
PART II. WEB SERVICES BINDINGS	53
CHAPTER 5. UNDERSTANDING BINDINGS IN WSDL	54
OVERVIEW	54
PORT TYPES AND BINDINGS	54
THE WSDL ELEMENTS	54
ADDING TO A CONTRACT	54
SUPPORTED BINDINGS	55
CHAPTER 6. USING SOAP 1.1 MESSAGES	56
6.1. ADDING A SOAP 1.1 BINDING	56
Using wsdl2soap	56
Example	57
6.2. ADDING SOAP HEADERS TO A SOAP 1.1 BINDING	58
Overview	58
Syntax	58
Splitting messages between body and header	59
Example	59
CHAPTER 7. USING SOAP 1.2 MESSAGES	61
7.1. ADDING A SOAP 1.2 BINDING TO A WSDL DOCUMENT	61
Using wsdl2soap	61
Example	62
7.2. ADDING HEADERS TO A SOAP 1.2 MESSAGE	63
Overview	63
Syntax	63
Splitting messages between body and header	64
Example	65
CHAPTER 8. SENDING BINARY DATA USING SOAP WITH ATTACHMENTS	68
OVERVIEW	68
NAMESPACE	68
CHANGING THE MESSAGE BINDING	68
DESCRIBING A MIME MULTIPART MESSAGE	68
EXAMPLE	70
CHAPTER 9. SENDING BINARY DATA WITH SOAP MTOM	72
9.1. OVERVIEW OF MTOM	72
9.2. ANNOTATING DATA TYPES TO USE MTOM	72
Overview	72
WSDL first	72
Java first	74
9.3. ENABLING MTOM	75

9.3.1. Using JAX-WS APIs	75
Overview	75
Service provider	75
Consumer	76
9.3.2. Using configuration	76
Overview	76
Procedure	76
Example	76
CHAPTER 10. USING XML DOCUMENTS	78
XML BINDING NAMESPACE	78
HAND EDITING	78
XML MESSAGES ON THE WIRE	78
OVERRIDING THE BINDING'S ROOTNODE ATTRIBUTE SETTING	80
PART III. WEB SERVICES TRANSPORTS	82
CHAPTER 11. UNDERSTANDING HOW ENDPOINTS ARE DEFINED IN WSDL	83
OVERVIEW	83
ENDPOINTS AND SERVICES	83
THE WSDL ELEMENTS	83
ADDING ENDPOINTS TO A CONTRACT	83
SUPPORTED TRANSPORTS	84
CHAPTER 12. USING HTTP	85
12.1. ADDING A BASIC HTTP ENDPOINT	85
Alternative HTTP runtimes	85
Netty HTTP URL	85
Payload types	85
SOAP 1.1	85
SOAP 1.2	86
Other messages types	86
12.2. CONFIGURING A CONSUMER	87
12.2.1. Mechanisms for HTTP Consumer Endpoints	87
12.2.2. Using Configuration	87
Namespace	87
Undertow runtime or Netty runtime	87
The conduit element	87
The client element	88
Example	92
More information	92
12.2.3. Using WSDL	93
Namespace	93
Undertow runtime or Netty runtime	93
The client element	93
Example	93
12.2.4. Consumer Cache Control Directives	93
12.3. CONFIGURING A SERVICE PROVIDER	94
12.3.1. Mechanisms for a HTTP Service Provider	94
12.3.2. Using Configuration	95
Namespace	95
Undertow runtime or Netty runtime	95
The destination element	95
The server element	96

Example	97
12.3.3. Using WSDL	98
Namespace	98
Undertow runtime or Netty runtime	98
The server element	98
Example	98
12.3.4. Service Provider Cache Control Directives	99
12.4. CONFIGURING THE UNDERTOW RUNTIME	100
Overview	100
Maven dependency	100
Namespace	100
The engine-factory element	101
The engine element	101
Configuring the thread pool	102
Example	103
12.5. CONFIGURING THE NETTY RUNTIME	104
Overview	104
Maven dependencies	104
Namespace	104
The engine-factory element	104
The engine element	105
Configuring the thread pool	107
Example	107
12.6. USING THE HTTP TRANSPORT IN DECOUPLED MODE	108
Overview	108
Configuring decoupled interactions	109
Configuring an endpoint to use WS-Addressing	109
Configuring the consumer	110
How messages are processed	110
CHAPTER 13. USING SOAP OVER JMS	113
13.1. BASIC CONFIGURATION	113
Overview	113
Specifying the JMS transport type	113
Specifying the target destination	113
Configuring JNDI and the JMS transport	114
13.2. JMS URIS	115
Overview	115
Syntax	115
JMS properties	115
JNDI properties	118
Additional JNDI properties	118
Example	119
Publishing a service	119
Consuming a service	120
13.3. WSDL EXTENSIONS	120
Overview	120
SOAP/JMS namespace	120
WSDL extension elements	121
Configuration scopes	122
Example	123
CHAPTER 14. USING GENERIC JMS	125

14.1. APPROACHES TO CONFIGURING JMS	125
14.2. USING THE JMS CONFIGURATION BEAN	125
Overview	125
Configuration namespace	125
Specifying the configuration	125
Applying the configuration to an endpoint	133
Applying the configuration to the transport	134
14.3. OPTIMIZING CLIENT-SIDE JMS PERFORMANCE	134
Overview	134
Pooling	134
Avoiding synchronous receives	135
14.4. CONFIGURING JMS TRANSACTIONS	135
Overview	135
Local transactions	135
JTA transactions	136
14.5. USING WSDL TO CONFIGURE JMS	137
14.5.1. JMS WSDL Extension Namespace	137
14.5.2. Basic JMS configuration	137
Overview	137
Specifying the JMS address	137
Specifying JNDI properties	138
Example	139
14.5.3. JMS client configuration	139
Overview	139
Specifying the message type	140
Example	140
14.5.4. JMS provider configuration	140
Overview	140
Specifying the configuration	141
Example	141
14.6. USING A NAMED REPLY DESTINATION	142
Overview	142
Setting the reply destination name	142
Example	142
CHAPTER 15. INTEGRATING WITH APACHE ACTIVEMQ	143
OVERVIEW	143
THE INITIAL CONTEXT FACTORY	143
LOOKING UP THE CONNECTION FACTORY	143
SYNTAX FOR DYNAMIC DESTINATIONS	143
CHAPTER 16. CONDUITS	145
OVERVIEW	145
CONDUIT LIFE-CYCLE	145
CONDUIT WEIGHT	145
PART IV. CONFIGURING WEB SERVICE ENDPOINTS	146
CHAPTER 17. CONFIGURING JAX-WS ENDPOINTS	147
17.1. CONFIGURING SERVICE PROVIDERS	147
17.1.1. Elements for Configuring Service Providers	147
17.1.2. Using the <code>jaxws:endpoint</code> Element	147
Overview	147
Identifying the endpoint being configured	147

Attributes	148
Example	150
17.1.3. Using the jaxws:server Element	151
Overview	151
Identifying the endpoint being configured	151
Attributes	152
Example	153
17.1.4. Adding Functionality to Service Providers	154
Overview	154
Elements	154
17.1.5. Enable Schema Validation on a JAX-WS Endpoint	155
Overview	155
Example	156
17.2. CONFIGURING CONSUMER ENDPOINTS	156
Overview	156
Basic Configuration Properties	156
Adding functionality	158
Example	159
Enable schema validation on a JAX-WS consumer	160
CHAPTER 18. CONFIGURING JAX-RS ENDPOINTS	161
18.1. CONFIGURING JAX-RS SERVER ENDPOINTS	161
18.1.1. Defining a JAX-RS Server Endpoint	161
Basic server endpoint definition	161
Blueprint example	161
Blueprint XML namespaces	162
Spring example	162
Spring XML namespaces	163
Auto-discovery in Spring XML	163
Lifecycle management in Spring XML	163
Attaching a WADL document	165
Schema validation	165
Specifying the data binding	166
Using the JMS transport	166
Extension mappings and language mappings	167
18.1.2. jaxrs:server Attributes	168
Attributes	168
18.1.3. jaxrs:server Child Elements	170
Child elements	170
18.2. CONFIGURING JAX-RS CLIENT ENDPOINTS	173
18.2.1. Defining a JAX-RS Client Endpoint	173
Injecting client proxies	173
Namespaces	173
Basic client endpoint definition	173
Specifying headers	173
18.2.2. jaxrs:client Attributes	174
Attributes	174
18.2.3. jaxrs:client Child Elements	175
Child elements	175
18.3. DEFINING REST SERVICES WITH THE MODEL SCHEMA	177
RESTful services without annotations	177
Example model schema	177
Namespaces	178

How to attach a model schema to an endpoint	178
Configuration of model schema referencing a class	178
Configuration of model schema referencing an interface	178
Model Schema Reference	179
CHAPTER 19. APACHE CXF LOGGING	182
19.1. OVERVIEW OF APACHE CXF LOGGING	182
Overview	182
Default properties file	182
Logging feature	182
Where to begin?	182
More information on java.util.logging	182
19.2. SIMPLE EXAMPLE OF USING LOGGING	183
Changing the log levels and output destination	183
19.3. DEFAULT LOGGING CONFIGURATION FILE	184
19.3.1. Overview of Logging Configuration	184
19.3.2. Configuring Logging Output	184
Overview	184
Configuring the console handler	185
Configuring the file handler	185
Configuring both the console handler and the file handler	186
19.3.3. Configuring Logging Levels	186
Logging levels	186
Configuring the global logging level	187
Configuring logging at an individual package	187
19.4. ENABLING LOGGING AT THE COMMAND LINE	187
Overview	187
Specifying the log configuration file on application	187
19.5. LOGGING FOR SUBSYSTEMS AND SERVICES	187
Overview	187
Apache CXF logging subsystems	188
Example	189
19.6. LOGGING MESSAGE CONTENT	189
Overview	189
Configuring message content logging	189
Adding the logging feature to an endpoint	190
Adding the logging feature to a consumer	190
Set logging to log INFO level messages	190
Logging SOAP messages	190
CHAPTER 20. DEPLOYING WS-ADDRESSING	193
20.1. INTRODUCTION TO WS-ADDRESSING	193
Overview	193
Supported specifications	193
Further information	193
20.2. WS-ADDRESSING INTERCEPTORS	193
Overview	193
WS-Addressing Interceptors	193
20.3. ENABLING WS-ADDRESSING	194
Overview	194
Adding WS-Addressing as a Feature	194
20.4. CONFIGURING WS-ADDRESSING ATTRIBUTES	195
Overview	195

Configuring WS-Addressing attributes	195
Using a WS-Policy assertion embedded in a feature	195
CHAPTER 21. ENABLING RELIABLE MESSAGING	197
21.1. INTRODUCTION TO WS-RM	197
Overview	197
How WS-RM works	197
WS-RM delivery assurances	198
Supported specifications	198
Selecting the WS-RM version	199
21.2. WS-RM INTERCEPTORS	199
Overview	199
Apache CXF WS-RM Interceptors	199
Enabling WS-RM	200
Configuring WS-RM Attributes	200
21.3. ENABLING WS-RM	200
Overview	200
Spring beans: explicitly adding interceptors	201
WS-Policy framework: implicitly adding interceptors	202
21.4. RUNTIME CONTROL	203
Overview	203
Runtime control options	203
Controlling WS-RM through JMX	204
Example of JMX control	204
21.5. CONFIGURING WS-RM	205
21.5.1. Configuring Apache CXF-Specific WS-RM Attributes	205
Overview	205
Children of the rmManager Spring bean	205
Example	206
21.5.2. Configuring Standard WS-RM Policy Attributes	206
Overview	206
WS-Policy RMAssertion Children	206
More detailed reference information	207
RMAssertion in rmManager Spring bean	207
Policy within a feature	208
WSDL file	208
External attachment	209
21.5.3. WS-RM Configuration Use Cases	209
Overview	209
Base retransmission interval	210
Exponential backoff for retransmission	210
Acknowledgement interval	210
Maximum unacknowledged messages threshold	211
Maximum length of an RM sequence	211
Message delivery assurance policies	212
21.6. CONFIGURING WS-RM PERSISTENCE	212
Overview	212
How it works	213
Enabling WS-persistence	213
Configuring WS-persistence	213
CHAPTER 22. ENABLING HIGH AVAILABILITY	215
22.1. INTRODUCTION TO HIGH AVAILABILITY	215

Overview	215
HA with static failover	215
22.2. ENABLING HA WITH STATIC FAILOVER	215
Overview	215
Encode replica details in your service WSDL file	215
Add the clustering feature to your client configuration	216
22.3. CONFIGURING HA WITH STATIC FAILOVER	217
Overview	217
Configuring a random strategy	217
CHAPTER 23. APACHE CXF BINDING IDS	218
TABLE OF BINDING IDS	218
APPENDIX A. USING THE MAVEN OSGI TOOLING	219
A.1. THE MAVEN BUNDLE PLUG-IN	219
A.2. SETTING UP A RED HAT FUSE OSGI PROJECT	219
Overview	219
Directory structure	219
Adding a bundle plug-in	220
Activating a bundle plug-in	221
Useful Maven archetypes	221
Spring OSGi archetype	221
Apache CXF code-first archetype	221
Apache CXF wsdl-first archetype	222
Apache Camel archetype	222
A.3. CONFIGURING THE BUNDLE PLUG-IN	222
Overview	222
Configuration properties	222
Setting a bundle's symbolic name	222
Setting a bundle's name	223
Setting a bundle's version	224
Specifying exported packages	224
Specifying private packages	225
Specifying imported packages	225
More information	226
PART V. DEVELOPING APPLICATIONS USING JAX-WS	227
CHAPTER 24. BOTTOM-UP SERVICE DEVELOPMENT	228
24.1. INTRODUCTION TO JAX-WS SERVICE DEVELOPMENT	228
24.2. CREATING THE SEI	228
Overview	228
Writing the interface	229
Implementing the interface	229
24.3. ANNOTATING THE CODE	230
24.3.1. Overview of JAX-WS Annotations	230
24.3.2. Required Annotations	230
Overview	231
The @WebService annotation	231
Annotating the SEI	232
Annotating the service implementation	232
24.3.3. Optional Annotations	233
Overview	233
The @SOAPBinding annotation	233

Document bare style parameters	234
Document wrapped parameters	235
Example	235
Overview	235
The @WebMethod annotation	236
The @RequestWrapper annotation	236
The @ResponseWrapper annotation	237
The @WebFault annotation	238
The @Oneway annotation	238
Example	239
Overview	239
The @WebParam annotation	239
The @WebResult annotation	240
Example	241
24.3.4. Apache CXF Annotations	242
24.3.4.1. WSDL Documentation	242
@WSDLDocumentation annotation	242
24.3.4.2. @WSDLDocumentation properties	242
@WSDLDocumentationCollection annotation	242
Placement in the WSDL contract	242
Example of @WSDLDocumentation	243
24.3.4.3. Using @WSDLDocumentation	243
24.3.4.4. WSDL generated with documentation	244
Example of @WSDLDocumentationCollection	244
24.3.4.5. Using @WSDLDocumentationCollection	244
24.3.4.6. Schema Validation of Messages	244
@SchemaValidation annotation	244
Schema validation type	244
24.3.4.7. Schema Validation Type Values	245
Example	245
24.3.4.8. Specifying the Data Binding	246
@DataBinding annotation	246
Supported data bindings	246
Example	246
24.3.4.9. Setting the data binding	246
24.3.4.10. Compressing Messages	247
@GZIP annotation	247
24.3.4.11. @GZIP Properties	247
@FastInfoSet	247
24.3.4.12. @FastInfoSet Properties	247
Example of @GZIP	247
24.3.4.13. Enabling GZIP	247
Exampe of @FastInfoSet	248
24.3.4.14. Enabling FastInfoSet	248
24.3.4.15. Enable Logging on an Endpoint	248
@Logging annotation	248
24.3.4.16. @Logging Properties	248
Example	248
24.3.4.17. Logging configuration using annotations	249
24.3.4.18. Adding Properties and Policies to an Endpoint	249
24.3.4.19. Adding properties	249
@EndpointProperty annotation	249
24.3.4.20. Configuring WS-Security Using @EndpointProperty Annotations	250

@EndpointProperties annotation	250
24.3.4.21. Configuring WS-Security Using an @EndpointProperties Annotation	250
24.3.4.22. Adding policies	250
@Policy annotation	250
24.3.4.23. @Policy Properties	251
@Policies annotation	251
Placement in the WSDL contract	251
Example of @Policy	252
Example of @Policies	252
24.4. GENERATING WSDL	252
Using Maven	252
Example	253
CHAPTER 25. DEVELOPING A CONSUMER WITHOUT A WSDL CONTRACT	255
25.1. JAVA-FIRST CONSUMER DEVELOPMENT	255
25.2. CREATING A SERVICE OBJECT	255
Overview	255
The create() methods	255
Example	256
25.3. ADDING A PORT TO A SERVICE	257
Overview	257
The addPort() method	257
Example	258
25.4. GETTING A PROXY FOR AN ENDPOINT	258
Overview	258
The getPort() method	259
Example	259
25.5. IMPLEMENTING THE CONSUMER'S BUSINESS LOGIC	259
Overview	259
Example	260
CHAPTER 26. A STARTING POINT WSDL CONTRACT	261
26.1. SAMPLE WSDL CONTRACT	261
CHAPTER 27. TOP-DOWN SERVICE DEVELOPMENT	264
27.1. OVERVIEW OF JAX-WS SERVICE PROVIDER DEVELOPMENT	264
27.2. GENERATING THE STARTING POINT CODE	264
Overview	264
Running the code generator	264
Generated code	265
Generated packages	266
27.3. IMPLEMENTING THE SERVICE PROVIDER	266
Generating the implementation code	266
Generated code	266
Implement the operation's logic	266
Example	267
CHAPTER 28. DEVELOPING A CONSUMER FROM A WSDL CONTRACT	268
28.1. GENERATING THE STUB CODE	268
Overview	268
Generating the consumer code	268
Generated code	269
28.2. IMPLEMENTING A CONSUMER	269
Overview	269

Generated service class	270
Service endpoint interface	271
Consumer main function	271
Client proxy generated with -fe cxf option	273
CHAPTER 29. FINDING WSDL AT RUNTIME	274
29.1. MECHANISMS FOR LOCATING THE WSDL DOCUMENT	274
29.2. INSTANTIATING A PROXY BY INJECTION	274
Overview	274
Procedure	274
Configuring the proxy	275
Coding the provider implementation	276
29.3. USING A JAX-WS CATALOG	276
Overview	276
Writing the catalog	276
Packaging the catalog	277
29.4. USING A CONTRACT RESOLVER	277
Overview	277
Implementing the contract resolver	277
Registering the contract resolver programmatically	278
Registering a contract resolver using configuration	278
Contract resolution order	280
CHAPTER 30. GENERIC FAULT HANDLING	281
30.1. RUNTIME FAULTS	281
Overview	281
APIs that throw WebServiceException	281
30.2. PROTOCOL FAULTS	282
Overview	282
Types of protocol exceptions	282
Using the SOAP protocol exception	282
CHAPTER 31. PUBLISHING A SERVICE	284
31.1. WHEN TO PUBLISH A SERVICE	284
31.2. APIS USED TO PUBLISH A SERVICE	284
Overview	284
Instantiating an service provider	284
Publishing a service provider	285
Stopping a published service provider	285
31.3. PUBLISHING A SERVICE IN A PLAIN JAVA APPLICATION	285
Overview	285
Generating a Server Mainline	286
Writing a Server Mainline	286
31.4. PUBLISHING A SERVICE IN AN OSGI CONTAINER	288
Overview	288
The bundle activator interface	288
Implementing the start method	288
Implementing the stop method	289
Informing the container	290
CHAPTER 32. BASIC DATA BINDING CONCEPTS	291
32.1. INCLUDING AND IMPORTING SCHEMA DEFINITIONS	291
Overview	291
xsd:include syntax	291

xsd:import syntax	292
Using non-referenced schema documents	293
32.2. XML NAMESPACE MAPPING	293
Overview	293
Package naming	293
Package contents	294
32.3. THE OBJECT FACTORY	295
Overview	295
Complex type factory methods	295
Element factory methods	295
32.4. ADDING CLASSES TO THE RUNTIME MARSHALLER	296
Overview	296
Using the @XmlSeeAlso annotation	296
Example	296
CHAPTER 33. USING XML ELEMENTS	298
OVERVIEW	298
XML SCHEMA MAPPING	298
JAVA MAPPING OF ELEMENTS WITH A NAMED TYPE	300
USING ELEMENTS WITH NAMED TYPES IN WSDL	301
JAVA MAPPING OF ELEMENTS WITH AN IN-LINE TYPE	301
JAVA MAPPING OF ABSTRACT ELEMENTS	302
JAVA MAPPING OF ELEMENTS WITH A DEFAULT VALUE	302
CHAPTER 34. USING SIMPLE TYPES	304
34.1. PRIMITIVE TYPES	304
Overview	304
Mappings	304
Wrapper classes	305
34.2. SIMPLE TYPES DEFINED BY RESTRICTION	306
Overview	306
Procedure	307
Defining a simple type in XML Schema	307
Mapping to Java	308
Enforcing facets	308
34.3. ENUMERATIONS	309
Overview	309
Defining an enumerated type in XML Schema	309
Mapping to Java	309
34.4. LISTS	310
Overview	310
Defining list types in XML Schema	311
Mapping list type elements to Java	312
Mapping list type parameters to Java	312
34.5. UNIONS	313
Overview	313
Defining in XML Schema	314
Mapping to Java	314
34.6. SIMPLE TYPE SUBSTITUTION	314
Overview	314
Default mapping and marshaling	314
Supporting lossless type substitution	315
CHAPTER 35. USING COMPLEX TYPES	316

35.1. BASIC COMPLEX TYPE MAPPING	316
Overview	316
Defining in XML Schema	316
Mapping to Java	317
35.2. ATTRIBUTES	320
Overview	320
Defining an attribute in XML Schema	321
Using an attribute group in XML Schema	322
Mapping attributes to Java	323
Mapping attribute groups to Java	324
35.3. DERIVING COMPLEX TYPES FROM SIMPLE TYPES	325
Overview	325
Derivation by extension	325
Derivation by restriction	326
Mapping to Java	326
35.4. DERIVING COMPLEX TYPES FROM COMPLEX TYPES	327
Overview	327
Schema syntax	327
Extending a complex type	327
Restricting a complex type	328
Mapping to Java	329
35.5. OCCURRENCE CONSTRAINTS	330
35.5.1. Schema Elements Supporting Occurrence Constraints	330
35.5.2. Occurrence Constraints on the All Element	330
XML Schema	330
Mapping to Java	330
35.5.3. Occurrence Constraints on the Choice Element	331
Overview	331
Using in XML Schema	331
Mapping to Java	331
minOccurs set to 0	332
35.5.4. Occurrence Constraints on Elements	333
Overview	333
minOccurs set to 0	333
minOccurs set to a value greater than 1	333
Elements with maxOccurs set	333
35.5.5. Occurrence Constraints on Sequences	333
Overview	333
Using XML Schema	333
Mapping to Java	334
minOccurs set to 0	335
35.6. USING MODEL GROUPS	335
Overview	335
Defining a model group in XML Schema	335
Using a model group in a type definition	336
Mapping to Java	337
Multiple occurrences	338
CHAPTER 36. USING WILD CARD TYPES	339
36.1. USING ANY ELEMENTS	339
Overview	339
Specifying in XML Schema	339
Mapping to Java	341

Marshalling	342
Unmarshalling	343
36.2. USING THE XML SCHEMA ANYTYPE TYPE	343
Overview	343
Using in XML Schema	343
Mapping to Java	343
Marshalling	344
Unmarshalling	344
36.3. USING UNBOUND ATTRIBUTES	345
Overview	345
Defining in XML Schema	345
Mapping to Java	345
Working with undeclared attributes	346
CHAPTER 37. ELEMENT SUBSTITUTION	348
37.1. SUBSTITUTION GROUPS IN XML SCHEMA	348
Overview	348
Syntax	348
Type restrictions	348
Abstract head elements	350
37.2. SUBSTITUTION GROUPS IN JAVA	350
Overview	350
Generated object factory methods	350
Substitution groups in interfaces	352
Substitution groups in complex types	353
Setting a substitution group property	354
Getting the value of a substitution group property	355
37.3. WIDGET VENDOR EXAMPLE	356
37.3.1. Widget Ordering Interface	356
37.3.2. The checkWidgets Operation	357
Overview	357
Consumer implementation	357
Service implementation	358
37.3.3. The placeWidgetOrder Operation	359
Overview	359
Consumer implementation	359
Service implementation	360
CHAPTER 38. CUSTOMIZING HOW TYPES ARE GENERATED	363
38.1. BASICS OF CUSTOMIZING TYPE MAPPINGS	363
Overview	363
Namespace	363
Version declaration	363
Using in-line customization	363
Using an external binding declaration	364
38.2. SPECIFYING THE JAVA CLASS OF AN XML SCHEMA PRIMITIVE	365
Overview	365
Syntax	365
Specifying the converters	368
What is generated	369
Implementing converters	370
Default primitive type converters	371
38.3. GENERATING JAVA CLASSES FOR SIMPLE TYPES	371

Overview	371
Adding the customization	371
Generated classes	372
38.4. CUSTOMIZING ENUMERATION MAPPING	373
Overview	373
Member name customizer	373
Class customizer	374
Member customizer	374
Examples	374
38.5. CUSTOMIZING FIXED VALUE ATTRIBUTE MAPPING	376
Overview	376
Global customization	377
Local mapping	377
Java mapping	378
38.6. SPECIFYING THE BASE TYPE OF AN ELEMENT OR AN ATTRIBUTE	379
Overview	379
Customization usage	379
Specializing or generalizing the default mapping	380
Usage with javaType	381
CHAPTER 39. USING A JAXBCONTEXT OBJECT	382
OVERVIEW	382
BEST PRACTICES	382
GETTING A JAXBCONTEXT OBJECT USING AN OBJECT FACTORY	382
GETTING A JAXBCONTEXT OBJECT USING PACKAGE NAMES	383
CHAPTER 40. DEVELOPING ASYNCHRONOUS APPLICATIONS	384
40.1. TYPES OF ASYNCHRONOUS INVOCATION	384
40.2. WSDL FOR ASYNCHRONOUS EXAMPLES	384
40.3. GENERATING THE STUB CODE	385
Overview	385
Using an external binding declaration	386
Using an embedded binding declaration	387
Generated interface	388
40.4. IMPLEMENTING AN ASYNCHRONOUS CLIENT WITH THE POLLING APPROACH	388
Overview	389
Using the non-blocking pattern	389
Using the blocking pattern	390
40.5. IMPLEMENTING AN ASYNCHRONOUS CLIENT WITH THE CALLBACK APPROACH	391
Overview	391
Implementing the callback	391
Implementing the consumer	392
40.6. CATCHING EXCEPTIONS RETURNED FROM A REMOTE SERVICE	394
Overview	394
Catching the exception	394
Getting the exception details	394
Example	394
CHAPTER 41. USING RAW XML MESSAGES	396
41.1. USING XML IN A CONSUMER	396
41.1.1. Usage Modes	396
Overview	396
Message mode	396
Payload mode	396

41.1.2. Data Types	397
Overview	397
Using Source objects	397
Using SOAPMessage objects	398
Using DataSource objects	398
Using JAXB objects	398
41.1.3. Working with Dispatch Objects	398
Procedure	398
Creating a Dispatch object	398
Constructing request messages	400
Synchronous invocation	401
Asynchronous invocation	401
Oneway invocation	402
41.2. USING XML IN A SERVICE PROVIDER	403
41.2.1. Messaging Modes	403
Overview	403
Message mode	403
Payload mode	403
41.2.2. Data Types	404
Overview	404
Using Source objects	404
Using SOAPMessage objects	405
Using DataSource objects	405
41.2.3. Implementing a Provider Object	405
Overview	405
Working with messages	406
The @WebService annotation	406
Implementing the invoke() method	407
Examples	407
CHAPTER 42. WORKING WITH CONTEXTS	410
42.1. UNDERSTANDING CONTEXTS	410
Overview	410
How properties are stored in a context	411
Property scopes	411
Overview of contexts in handlers	412
Overview of contexts in service implementations	412
Overview of contexts in consumer implementations	412
42.2. WORKING WITH CONTEXTS IN A SERVICE IMPLEMENTATION	413
Overview	413
Obtaining a context	413
Reading a property from a context	414
Setting properties in a context	414
Supported contexts	415
42.3. WORKING WITH CONTEXTS IN A CONSUMER IMPLEMENTATION	419
Overview	419
Obtaining a context	419
Reading a property from a context	420
Setting properties in a context	420
Supported contexts	421
42.4. WORKING WITH JMS MESSAGE PROPERTIES	422
42.4.1. Inspecting JMS Message Headers	422
Getting the JMS Message Headers in a Service	422

Getting JMS Message Header Properties in a Consumer	423
42.4.2. Inspecting the Message Header Properties	424
Standard JMS Header Properties	424
Optional Header Properties	424
Example	424
42.4.3. Setting JMS Properties	425
JMS Header Properties	425
Optional JMS Header Properties	426
Client Receive Timeout	426
Example	426
CHAPTER 43. WRITING HANDLERS	428
43.1. HANDLERS: AN INTRODUCTION	428
Overview	428
Handler types	429
Implementation of handlers	430
Adding handlers to an application	430
43.2. IMPLEMENTING A LOGICAL HANDLER	430
Overview	430
Procedure	431
43.3. HANDLING MESSAGES IN A LOGICAL HANDLER	431
Overview	431
Getting the message data	431
Working with the message body as an XML object	432
Working with the message body as a JAXB object	432
Working with context properties	433
Determining the direction of the message	433
Determining the return value	433
Example	434
43.4. IMPLEMENTING A PROTOCOL HANDLER	436
Overview	436
Procedure	437
Implementing the getHeaders() method	437
43.5. HANDLING MESSAGES IN A SOAP HANDLER	437
Overview	437
Working with the message body	437
Getting the SOAP headers	438
Working with context properties	438
Determining the direction of the message	438
Determining the return value	439
Example	440
43.6. INITIALIZING A HANDLER	441
Overview	441
Order of initialization	441
43.7. HANDLING FAULT MESSAGES	441
Overview	441
Getting the message payload	442
Determining the return value	442
Example	442
43.8. CLOSING A HANDLER	442
43.9. RELEASING A HANDLER	443
Overview	443
Order of release	443

43.10. CONFIGURING ENDPOINTS TO USE HANDLERS	443
43.10.1. Programmatic Configuration	443
43.10.1.1. Adding a Handler Chain to a Consumer	443
Overview	443
Procedure	443
Example	444
43.10.1.2. Adding a Handler Chain to a Service Provider	444
Overview	444
Procedure	444
The @HandlerChain annotation	445
Handler configuration file	445
43.10.2. Spring Configuration	447
Overview	447
Procedure	447
The handlers element	447
Example	448
CHAPTER 44. MAVEN TOOLING REFERENCE	449
44.1. PLUG-IN SETUP	449
Dependencies	449
44.2. CXF-CODEGEN-PLUGIN	449
Overview	449
Basic example	449
Basic configuration settings	450
Description	450
WSDL options	450
Default options	450
Specifying code generation options	451
Specifying binding files	451
Generating code for a specific WSDL service	452
Generating code for multiple WSDL files	452
Downloading WSDL from a Maven repository	453
Encoding	453
Forking a separate process	453
Options reference	454
44.3. JAVA2WS	458
Synopsis	458
Description	458
Required configuration	458
Optional configuration	458
PART VI. DEVELOPING RESTFUL WEB SERVICES	460
CHAPTER 45. INTRODUCTION TO RESTFUL WEB SERVICES	461
OVERVIEW	461
BASIC REST PRINCIPLES	461
RESOURCES	462
REST BEST PRACTICES	462
DESIGNING A RESTFUL WEB SERVICE	462
IMPLEMENTING REST WITH APACHE CXF	463
DATA BINDINGS	463
CHAPTER 46. CREATING RESOURCES	464
46.1. INTRODUCTION	464

Overview	464
Types of resources	464
Example	464
46.2. BASIC JAX-RS ANNOTATIONS	465
Overview	465
Setting the path	465
Specifying HTTP verbs	466
46.3. ROOT RESOURCE CLASSES	466
Overview	466
Requirements	466
Example	466
46.4. WORKING WITH RESOURCE METHODS	468
Overview	468
General constraints	468
Parameters	468
Return values	469
46.5. WORKING WITH SUB-RESOURCES	469
Overview	469
Specifying a sub-resource	470
Sub-resource methods	470
Sub-resource locators	471
46.6. RESOURCE SELECTION METHOD	472
Overview	472
The basic selection algorithm	473
Selecting from multiple resource classes	473
Selecting from multiple resource methods	474
Customizing the selection process	475
CHAPTER 47. PASSING INFORMATION INTO RESOURCE CLASSES AND METHODS	476
47.1. BASICS OF INJECTING DATA	476
Overview	476
When data is injected	476
Supported data types	476
47.2. USING JAX-RS APIs	476
47.2.1. JAX-RS Annotation Types	476
47.2.2. Injecting data from a request URI	477
Overview	477
Getting data from the URI's path	477
Using query parameters	478
Using matrix parameters	479
Disabling URI decoding	480
Error handling	481
47.2.3. Injecting data from the HTTP message header	481
Overview	481
Injecting information from the HTTP headers	481
Injecting information from a cookie	481
Error handling	482
47.2.4. Injecting data from HTML forms	482
Overview	482
Using the @FormParam annotation to inject form data	482
Example	483
47.2.5. Specifying a default value to inject	483
Overview	483

Syntax	484
Dealing with lists and sets	484
Example	484
47.2.6. Injecting Parameters into a Java Bean	485
Overview	485
Injection target	485
Example without BeanParam annotation	485
Example with BeanParam annotation	486
47.3. PARAMETER CONVERTERS	487
Overview	487
Automatic conversions	487
Parameter converters	487
Factory pattern	487
ParamConverter interface	487
ParamConverterProvider interface	488
Binding the parameter converter provider	488
Example	489
Using the parameter converter	489
Lazy conversion of default value	490
47.4. USING APACHE CXF EXTENSIONS	490
Overview	490
Supported injection annotations	491
Syntax	491
Example	491
CHAPTER 48. RETURNING INFORMATION TO THE CONSUMER	492
48.1. RETURN TYPES	492
48.2. RETURNING PLAIN JAVA CONSTRUCTS	492
Overview	492
Returnable types	492
MIME types	493
Response codes	493
48.3. FINE TUNING AN APPLICATION'S RESPONSES	493
48.3.1. Basics of building responses	493
Overview	493
Relationship between a response and a response builder	494
Getting a response builder	494
More information	495
48.3.2. Creating responses for common use cases	495
Overview	495
Creating responses for successful requests	495
Creating responses for redirection	496
Creating responses to signal errors	496
48.3.3. Handling more advanced responses	497
Overview	497
Adding custom headers	497
Adding a cookie	497
Setting the response status	498
Setting cache control directives	498
48.4. RETURNING ENTITIES WITH GENERIC TYPE INFORMATION	499
Overview	499
Using a GenericEntity<T> object	499
Creating a GenericEntity<T> object	499

48.5. ASYNCHRONOUS RESPONSE	500
48.5.1. Asynchronous Processing on the Server	500
Overview	500
Basic model for asynchronous processing	500
Thread pool implementation with Java executor	501
Defining an asynchronous resource method	502
AsyncResponse class	502
Encapsulating a suspended request as a Runnable	502
Example of asynchronous processing	503
48.5.2. Timeouts and Timeout Handlers	504
Overview	504
Example of setting a timeout without a handler	504
Default timeout behaviour	505
TimeoutHandler interface	505
Example of setting a timeout with a handler	505
Using a timeout handler to cancel the response	506
Dealing with a cancelled response in the Runnable instance	506
48.5.3. Handling Dropped Connections	507
Overview	507
ConnectionCallback interface	507
Registering a connection callback	507
Typical scenario for connection callback	507
48.5.4. Registering Callbacks	508
Overview	508
CompletionCallback interface	508
Registering a completion callback	508
CHAPTER 49. JAX-RS 2.0 CLIENT API	509
49.1. INTRODUCTION TO THE JAX-RS 2.0 CLIENT API	509
Overview	509
Dependencies	509
Client API package	509
Example of a simple client request	510
Fluent API	510
Steps to make a REST invocation	510
Bootstrap the client	510
Configure the target	510
Build and make the invocation	511
Parse the response	511
49.2. BUILDING THE CLIENT TARGET	511
Overview	511
WebTarget builder class	511
Create the client target	512
Base path and path segments	512
URI template parameters	512
Define query parameters	512
Define matrix parameters	513
49.3. BUILDING THE CLIENT INVOCATION	513
Overview	513
Invocation.Builder class	513
Create the invocation builder	513
Define HTTP headers	513
Define cookies	513

Define properties	514
Define accepted media types, languages, or encodings	514
Invoke HTTP method	514
Typed responses	514
Specifying the outgoing message in post or put	514
Delayed invocation	515
Asynchronous invocation	515
49.4. PARSING REQUESTS AND RESPONSES	516
Overview	516
Entities	516
Variants	516
Entity providers	516
Standard entity providers	516
Response object	517
Accessing the response status	517
Accessing the returned headers	517
Accessing the returned cookies	518
Accessing the returned message content	518
Collection return value	518
49.5. CONFIGURING THE CLIENT ENDPOINT	518
Overview	518
Example	519
Configurable API for registering objects	519
What can you configure on the client?	519
Features	519
Providers	519
Filters	520
Interceptors	520
Properties	520
Other configurable types	520
49.6. ASYNCHRONOUS PROCESSING ON THE CLIENT	520
Overview	520
Asynchronous invocation with Future<V> return value	520
Asynchronous invocation with invocation callback	521
CHAPTER 50. HANDLING EXCEPTIONS	523
50.1. OVERVIEW OF JAX-RS EXCEPTION CLASSES	523
Overview	523
JAX-RS runtime level exceptions	523
JAX-RS application level exceptions	523
50.2. USING WEBAPPLICATIONEXCEPTION EXCEPTIONS TO REPORT	524
Overview	524
Creating a simple exception	524
Setting the status code returned to the client	524
Providing an entity body	524
Extending the generic exception	525
50.3. JAX-RS 2.0 EXCEPTION TYPES	525
Overview	525
Exception hierarchy	526
WebApplicationException class	526
ClientErrorException class	526
ServerErrorException class	526
RedirectionException class	526

Client exception subclasses	526
Server exception subclasses	527
50.4. MAPPING EXCEPTIONS TO RESPONSES	527
Overview	527
How exception mappers are selected	527
Implementing an exception mapper	528
Registering an exception mapper	529
Registering an exception mapper for WebApplicationException	529
CHAPTER 51. ENTITY SUPPORT	531
OVERVIEW	531
NATIVELY SUPPORTED TYPES	531
CUSTOM READERS	532
CUSTOM WRITERS	536
REGISTERING READERS AND WRITERS	540
CHAPTER 52. GETTING AND USING CONTEXT INFORMATION	542
52.1. INTRODUCTION TO CONTEXTS	542
Context annotation	542
Types of contexts	542
Where context information can be used	542
Scope	542
Adding contexts	543
52.2. WORKING WITH THE FULL REQUEST URI	543
52.2.1. Injecting the URI information	543
Overview	543
Example	543
52.2.2. Working with the URI	543
Overview	544
Getting the Base URI	544
Getting the path	544
Getting the full request URI	546
52.2.3. Getting the value of URI template variables	547
Overview	547
Methods for getting the path parameters	547
Example	547
CHAPTER 53. ANNOTATION INHERITANCE	549
OVERVIEW	549
INHERITANCE RULES	549
OVERRIDING INHERITED ANNOTATIONS	549
CHAPTER 54. EXTENDING JAX-RS ENDPOINTS WITH SWAGGER SUPPORT	551
54.1. SWAGGER2FEATURE OPTIONS	551
54.2. KARAF IMPLEMENTATIONS	552
54.2.1. Quickstart example	552
54.2.2. Enabling Swagger	552
54.3. SPRING BOOT IMPLEMENTATIONS	556
54.3.1. Quickstart example	556
54.3.2. Enabling Swagger	556
54.4. ACCESSING SWAGGER DOCUMENTS	558
54.5. ACCESSING SWAGGER THROUGH A REVERSE PROXY	559
PART VII. DEVELOPING APACHE CXF INTERCEPTORS	560

CHAPTER 55. INTERCEPTORS IN THE APACHE CXF RUNTIME	561
OVERVIEW	561
MESSAGE PROCESSING IN APACHE CXF	562
INTERCEPTORS	563
PHASES	564
INTERCEPTOR CHAINS	564
DEVELOPING INTERCEPTORS	564
CHAPTER 56. THE INTERCEPTOR APIs	565
INTERFACES	565
ABSTRACT INTERCEPTOR CLASS	566
CHAPTER 57. DETERMINING WHEN THE INTERCEPTOR IS INVOKED	567
57.1. SPECIFYING THE INTERCEPTOR LOCATION	567
57.2. SPECIFYING AN INTERCEPTOR'S PHASE	567
Overview	567
Phase	567
Specifying a phase	568
Setting the phase	568
57.3. CONSTRAINING AN INTERCEPTORS PLACEMENT IN A PHASE	569
Overview	569
Add to the chain before	569
Add to the chain after	570
CHAPTER 58. IMPLEMENTING THE INTERCEPTORS PROCESSING LOGIC	571
58.1. INTERCEPTOR FLOW	571
58.2. PROCESSING MESSAGES	571
Overview	571
Getting the message contents	571
Determining the message's direction	572
Example	572
58.3. UNWINDING AFTER AN ERROR	573
Overview	573
Getting the message payload	574
Example	574
CHAPTER 59. CONFIGURING ENDPOINTS TO USE INTERCEPTORS	575
59.1. DECIDING WHERE TO ATTACH INTERCEPTORS	575
Overview	575
Endpoints and proxies	575
Factories	575
Bindings	575
Buses	576
Combining attachment points	576
59.2. ADDING INTERCEPTORS USING CONFIGURATION	576
Overview	576
Configuration elements	576
Examples	577
More information	578
59.3. ADDING INTERCEPTORS PROGRAMMATICALLY	578
59.3.1. Approaches to Adding Interceptors	578
59.3.2. Using the interceptor provider API	578
Overview	578
Procedure	579

Attaching an interceptor to a consumer	579
Attaching an interceptor to a service provider	580
Attaching an interceptor to a bus	581
59.3.3. Using Java annotations	581
Overview	581
Where to place the annotations	581
The annotations	582
Listing the interceptors	582
Example	582
CHAPTER 60. MANIPULATING INTERCEPTOR CHAINS ON THE FLY	583
OVERVIEW	583
CHAIN LIFE-CYCLE	583
GETTING THE INTERCEPTOR CHAIN	583
ADDING INTERCEPTORS	583
REMOVING INTERCEPTORS	584
CHAPTER 61. JAX-RS 2.0 FILTERS AND INTERCEPTORS	586
61.1. INTRODUCTION TO JAX-RS FILTERS AND INTERCEPTORS	586
Overview	586
Filters	586
Interceptors	586
Server processing pipeline	586
Server extension points	587
Client processing pipeline	587
Client extension points	587
Filter and interceptor order	587
Filter classes	588
Interceptor classes	588
61.2. CONTAINER REQUEST FILTER	588
Overview	588
ContainerRequestFilter interface	588
ContainerRequestContext interface	589
Sample implementation for PreMatchContainerRequest filter	590
Sample implementation for ContainerRequest filter	591
Injecting ResourceInfo	592
Aborting the invocation	592
Binding the server request filter	593
61.3. CONTAINER RESPONSE FILTER	594
Overview	594
ContainerResponseFilter interface	594
ContainerResponseContext interface	595
Sample implementation	597
Binding the server response filter	597
61.4. CLIENT REQUEST FILTER	598
Overview	598
ClientRequestFilter interface	599
ClientRequestContext interface	599
Sample implementation	600
Aborting the invocation	601
Registering the client request filter	602
61.5. CLIENT RESPONSE FILTER	602
Overview	602

ClientResponseFilter interface	602
ClientResponseContext interface	603
Sample implementation	604
Registering the client response filter	605
61.6. ENTITY READER INTERCEPTOR	605
Overview	605
ReaderInterceptor interface	605
ReaderInterceptorContext interface	606
InterceptorContext interface	606
Sample implementation on the client side	607
Sample implementation on the server side	608
Binding a reader interceptor on the client side	608
Binding a reader interceptor on the server side	609
61.7. ENTITY WRITER INTERCEPTOR	610
Overview	610
WriterInterceptor interface	610
WriterInterceptorContext interface	610
InterceptorContext interface	611
Sample implementation on the client side	611
Sample implementation on the server side	612
Binding a writer interceptor on the client side	612
Binding a writer interceptor on the server side	613
61.8. DYNAMIC BINDING	614
Overview	614
DynamicFeature interface	614
Implementing a dynamic feature	615
Example dynamic feature	615
Dynamic binding process	615
FeatureContext interface	615
CHAPTER 62. APACHE CXF MESSAGE PROCESSING PHASES	617
INBOUND PHASES	617
OUTBOUND PHASES	618
CHAPTER 63. APACHE CXF PROVIDED INTERCEPTORS	620
63.1. CORE APACHE CXF INTERCEPTORS	620
Inbound	620
Outbound	620
63.2. FRONT-ENDS	620
JAX-WS	620
JAX-RS	622
63.3. MESSAGE BINDINGS	622
SOAP	622
XML	624
CORBA	625
63.4. OTHER FEATURES	626
Logging	626
WS-Addressing	627
WS-RM	627
CHAPTER 64. INTERCEPTOR PROVIDERS	629
OVERVIEW	629
LIST OF PROVIDERS	629

PART VIII. APACHE CXF FEATURES	631
CHAPTER 65. BEAN VALIDATION	632
65.1. INTRODUCTION	632
Overview	632
Example of annotated class	632
Bean validation or schema validation?	632
Dependencies	633
Core dependencies	633
Hibernate Validator dependencies	633
Resolving the validation provider in an OSGi environment	634
Configuring the validation provider explicitly in OSGi	634
Example HibernateValidationProviderResolver class	634
65.2. DEVELOPING SERVICES WITH BEAN VALIDATION	635
65.2.1. Annotating a Service Bean	635
Overview	635
Validating simple input parameters	635
Validating complex input parameters	635
Validating return values (non-Response)	636
Validating return values (Response)	636
65.2.2. Standard Annotations	636
Bean validation constraints	636
65.2.3. Custom Annotations	638
Defining custom constraints in Hibernate	638
65.3. CONFIGURING BEAN VALIDATION	638
65.3.1. JAX-WS Configuration	638
Overview	638
Namespaces	638
Bean validation feature	639
Sample JAX-WS configuration with bean validation feature	639
Common bean validation 1.1 interceptors	640
Sample JAX-WS configuration with bean validation interceptors	640
Configuring a BeanValidationProvider	641
65.3.2. JAX-RS Configuration	641
Overview	641
Namespaces	642
Bean validation feature	642
Validation exception mapper	642
Sample JAX-RS configuration	642
Common bean validation 1.1 interceptors	643
Sample JAX-RS configuration with bean validation interceptors	643
Configuring a BeanValidationProvider	644
65.3.3. JAX-RS 2.0 Configuration	644
Overview	644
Bean validation feature	644
Validation exception mapper	644
Bean validation invoker	645
Sample JAX-RS 2.0 configuration with bean validation feature	645
Common bean validation 1.1 interceptors	645
Sample JAX-RS 2.0 configuration with bean validation interceptors	646
Configuring a BeanValidationProvider	647
Configuring a JAXRSParameterNameProvider	647

PART I. WRITING WSDL CONTRACTS

This part describes how to define a Web service interface using WSDL.

CHAPTER 1. INTRODUCING WSDL CONTRACTS

Abstract

WSDL documents define services using Web Service Description Language and a number of possible extensions. The documents have a logical part and a concrete part. The abstract part of the contract defines the service in terms of implementation neutral data types and messages. The concrete part of the document defines how an endpoint implementing a service will interact with the outside world.

The recommended approach to design services is to define your services in WSDL and XML Schema before writing any code. When hand-editing WSDL documents you must make sure that the document is valid, as well as correct. To do this you must have some familiarity with WSDL. You can find the standard on the W3C web site, www.w3.org.

1.1. STRUCTURE OF A WSDL DOCUMENT

Overview

A WSDL document is, at its simplest, a collection of elements contained within a root **definition** element. These elements describe a service and how an endpoint implementing that service is accessed.

A WSDL document has two distinct parts:

- A logical part that defines the service in implementation neutral terms
- A concrete part that defines how an endpoint implementing the service is exposed on a network

The logical part

The logical part of a WSDL document contains the **types**, the **message**, and the **portType** elements. It describes the service's interface and the messages exchanged by the service. Within the **types** element, XML Schema is used to define the structure of the data that makes up the messages. A number of **message** elements are used to define the structure of the messages used by the service. The **portType** element contains one or more **operation** elements that define the messages sent by the operations exposed by the service.

The concrete part

The concrete part of a WSDL document contains the **binding** and the **service** elements. It describes how an endpoint that implements the service connects to the outside world. The **binding** elements describe how the data units described by the **message** elements are mapped into a concrete, on-the-wire data format, such as SOAP. The **service** elements contain one or more **port** elements which define the endpoints implementing the service.

1.2. WSDL ELEMENTS

A WSDL document is made up of the following elements:

- **definitions** – The root element of a WSDL document. The attributes of this element specify the name of the WSDL document, the document's target namespace, and the shorthand definitions for the namespaces referenced in the WSDL document.

- **types** – The XML Schema definitions for the data units that form the building blocks of the messages used by a service. For information about defining data types see [Chapter 2, Defining Logical Data Units](#).
- **message** – The description of the messages exchanged during invocation of a services operations. These elements define the arguments of the operations making up your service. For information on defining messages see [Chapter 3, Defining Logical Messages Used by a Service](#).
- **portType** – A collection of **operation** elements describing the logical interface of a service. For information about defining port types see [Chapter 4, Defining Your Logical Interfaces](#).
- **operation** – The description of an action performed by a service. Operations are defined by the messages passed between two endpoints when the operation is invoked. For information on defining operations see [the section called "Operations"](#).
- **binding** – The concrete data format specification for an endpoint. A **binding** element defines how the abstract messages are mapped into the concrete data format used by an endpoint. This element is where specifics such as parameter order and return values are specified.
- **service** – A collection of related **port** elements. These elements are repositories for organizing endpoint definitions.
- **port** – The endpoint defined by a binding and a physical address. These elements bring all of the abstract definitions together, combined with the definition of transport details, and they define the physical endpoint on which a service is exposed.

1.3. DESIGNING A CONTRACT

To design a WSDL contract for your services you must perform the following steps:

1. Define the data types used by your services.
2. Define the messages used by your services.
3. Define the interfaces for your services.
4. Define the bindings between the messages used by each interface and the concrete representation of the data on the wire.
5. Define the transport details for each of the services.

CHAPTER 2. DEFINING LOGICAL DATA UNITS

Abstract

When describing a service in a WSDL contract complex data types are defined as logical units using XML Schema.

2.1. INTRODUCTION TO LOGICAL DATA UNITS

When defining a service, the first thing you must consider is how the data used as parameters for the exposed operations is going to be represented. Unlike applications that are written in a programming language that uses fixed data structures, services must define their data in logical units that can be consumed by any number of applications. This involves two steps:

1. Breaking the data into logical units that can be mapped into the data types used by the physical implementations of the service
2. Combining the logical units into messages that are passed between endpoints to carry out the operations

This chapter discusses the first step. [Chapter 3, Defining Logical Messages Used by a Service](#) discusses the second step.

2.2. MAPPING DATA INTO LOGICAL DATA UNITS

Overview

The interfaces used to implement a service define the data representing operation parameters as XML documents. If you are defining an interface for a service that is already implemented, you must translate the data types of the implemented operations into discreet XML elements that can be assembled into messages. If you are starting from scratch, you must determine the building blocks from which your messages are built, so that they make sense from an implementation standpoint.

Available type systems for defining service data units

According to the WSDL specification, you can use any type system you choose to define data types in a WSDL contract. However, the W3C specification states that XML Schema is the preferred canonical type system for a WSDL document. Therefore, XML Schema is the intrinsic type system in Apache CXF.

XML Schema as a type system

XML Schema is used to define how an XML document is structured. This is done by defining the elements that make up the document. These elements can use native XML Schema types, like `xsd:int`, or they can use types that are defined by the user. User defined types are either built up using combinations of XML elements or they are defined by restricting existing types. By combining type definitions and element definitions you can create intricate XML documents that can contain complex data.

When used in WSDL XML Schema defines the structure of the XML document that holds the data used to interact with a service. When defining the data units used by your service, you can define them as types that specify the structure of the message parts. You can also define your data units as elements that make up the message parts.

Considerations for creating your data units

You might consider simply creating logical data units that map directly to the types you envision using when implementing the service. While this approach works, and closely follows the model of building RPC-style applications, it is not necessarily ideal for building a piece of a service-oriented architecture.

The Web Services Interoperability Organization's WS-I basic profile provides a number of guidelines for defining data units and can be accessed at <http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html#WSDLTYPES>. In addition, the W3C also provides the following guidelines for using XML Schema to represent data types in WSDL documents:

- Use elements, not attributes.
- Do not use protocol-specific types as base types.

2.3. ADDING DATA UNITS TO A CONTRACT

Overview

Depending on how you choose to create your WSDL contract, creating new data definitions requires varying amounts of knowledge. The Apache CXF GUI tools provide a number of aids for describing data types using XML Schema. Other XML editors offer different levels of assistance. Regardless of the editor you choose, it is a good idea to have some knowledge about what the resulting contract should look like.

Procedure

Defining the data used in a WSDL contract involves the following steps:

1. Determine all the data units used in the interface described by the contract.
2. Create a **types** element in your contract.
3. Create a **schema** element, shown in [Example 2.1, "Schema entry for a WSDL contract"](#), as a child of the **type** element.

The **targetNamespace** attribute specifies the namespace under which new data types are defined. Best practice is to also define the namespace that provides access to the target namespace. The remaining entries should not be changed.

Example 2.1. Schema entry for a WSDL contract

```
<schema targetNamespace="http://schemas.iona.com/bank.idl"
       xmlns="http://www.w3.org/2001/XMLSchema"
       xmlns:xsd1="http://schemas.iona.com/bank.idl"
       xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
```

4. For each complex type that is a collection of elements, define the data type using a **complexType** element. See [Section 2.5.1, "Defining data structures"](#).
5. For each array, define the data type using a **complexType** element. See [Section 2.5.2, "Defining arrays"](#).

6. For each complex type that is derived from a simple type, define the data type using a **simpleType** element. See [Section 2.5.4, "Defining types by restriction"](#).
7. For each enumerated type, define the data type using a **simpleType** element. See [Section 2.5.5, "Defining enumerated types"](#).
8. For each element, define it using an **element** element. See [Section 2.6, "Defining elements"](#).

2.4. XML SCHEMA SIMPLE TYPES

Overview

If a message part is going to be of a simple type it is not necessary to create a type definition for it. However, the complex types used by the interfaces defined in the contract are defined using simple types.

Entering simple types

XML Schema simple types are mainly placed in the **element** elements used in the types section of your contract. They are also used in the **base** attribute of **restriction** elements and **extension** elements.

Simple types are always entered using the **xsd** prefix. For example, to specify that an element is of type **int**, you would enter **xsd:int** in its **type** attribute as shown in [Example 2.2, "Defining an element with a simple type"](#).

Example 2.2. Defining an element with a simple type

```
<element name="simpleInt" type="xsd:int" />
```

Supported XSD simple types

Apache CXF supports the following XML Schema simple types:

- **xsd:string**
- **xsd:normalizedString**
- **xsd:int**
- **xsd:unsignedInt**
- **xsd:long**
- **xsd:unsignedLong**
- **xsd:short**
- **xsd:unsignedShort**
- **xsd:float**
- **xsd:double**

- **xsd:boolean**
- **xsd:byte**
- **xsd:unsignedByte**
- **xsd:integer**
- **xsd:positiveInteger**
- **xsd:negativeInteger**
- **xsd:nonPositiveInteger**
- **xsd:nonNegativeInteger**
- **xsd:decimal**
- **xsd:dateTime**
- **xsd:time**
- **xsd:date**
- **xsd:QName**
- **xsd:base64Binary**
- **xsd:hexBinary**
- **xsd:ID**
- **xsd:token**
- **xsd:language**
- **xsd:Name**
- **xsd:NCName**
- **xsd:NMTOKEN**
- **xsd:anySimpleType**
- **xsd:anyURI**
- **xsd:gYear**
- **xsd:gMonth**
- **xsd:gDay**
- **xsd:gYearMonth**
- **xsd:gMonthDay**

2.5. DEFINING COMPLEX DATA TYPES

Abstract

XML Schema provides a flexible and powerful mechanism for building complex data structures from its simple data types. You can create data structures by creating a sequence of elements and attributes. You can also extend your defined types to create even more complex types.

In addition to building complex data structures, you can also describe specialized types such as enumerated types, data types that have a specific range of values, or data types that need to follow certain patterns by either extending or restricting the primitive types.

2.5.1. Defining data structures

Overview

In XML Schema, data units that are a collection of data fields are defined using **complexType** elements. Specifying a complex type requires three pieces of information:

1. The name of the defined type is specified in the **name** attribute of the **complexType** element.
2. The first child element of the **complexType** describes the behavior of the structure's fields when it is put on the wire. See [the section called "Complex type varieties"](#).
3. Each of the fields of the defined structure are defined in **element** elements that are grandchildren of the **complexType** element. See [the section called "Defining the parts of a structure"](#).

For example, the structure shown in [Example 2.3, "Simple Structure"](#) is defined in XML Schema as a complex type with two elements.

Example 2.3. Simple Structure

```
struct personallInfo
{
    string name;
    int age;
};
```

[Example 2.4, "A complex type"](#) shows one possible XML Schema mapping for the structure shown in [Example 2.3, "Simple Structure"](#). The structure defined in [Example 2.4, "A complex type"](#) generates a message containing two elements: **name** and **age**.

Example 2.4. A complex type

```
<complexType name="personallInfo">
<sequence>
    <element name="name" type="xsd:string" />
    <element name="age" type="xsd:int" />
</sequence>
</complexType>
```

Complex type varieties

XML Schema has three ways of describing how the fields of a complex type are organized when represented as an XML document and passed on the wire. The first child element of the **complexType** element determines which variety of complex type is being used. [Table 2.1, "Complex type descriptor elements"](#) shows the elements used to define complex type behavior.

Table 2.1. Complex type descriptor elements

Element	Complex Type Behavior
sequence	All of a complex type's fields can be present and they must be in the order in which they are specified in the type definition.
all	All of the complex type's fields can be present but they can be in any order.
choice	Only one of the elements in the structure can be placed in the message.

If the structure is defined using a **choice** element, as shown in [Example 2.5, "Simple complex choice type"](#), it generates a message with either a **name** element or an **age** element.

Example 2.5. Simple complex choice type

```
<complexType name="personalInfo">
  <choice>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int"/>
  </choice>
</complexType>
```

Defining the parts of a structure

You define the data fields that make up a structure using **element** elements. Every **complexType** element should contain at least one **element** element. Each **element** element in the **complexType** element represents a field in the defined data structure.

To fully describe a field in a data structure, **element** elements have two required attributes:

- The **name** attribute specifies the name of the data field and it must be unique within the defined complex type.
- The **type** attribute specifies the type of the data stored in the field. The type can be either one of the XML Schema simple types, or any named complex type that is defined in the contract.

In addition to **name** and **type**, **element** elements have two other commonly used optional attributes: **minOccurs** and **maxOccurs**. These attributes place bounds on the number of times the field occurs in the structure. By default, each field occurs only once in a complex type. Using these attributes, you can

change how many times a field must or can appear in a structure. For example, you can define a field, **previousJobs**, that must occur at least three times, and no more than seven times, as shown in [Example 2.6, "Simple complex type with occurrence constraints"](#).

Example 2.6. Simple complex type with occurrence constraints

```
<complexType name="personalInfo">
  <all>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int"/>
    <element name="previousJobs" type="xsd:string">
      minOccurs="3" maxOccurs="7"/>
    </element>
  </all>
</complexType>
```

You can also use the **minOccurs** to make the **age** field optional by setting the **minOccurs** to zero as shown in [Example 2.7, "Simple complex type with minOccurs set to zero"](#). In this case **age** can be omitted and the data will still be valid.

Example 2.7. Simple complex type with minOccurs set to zero

```
<complexType name="personalInfo">
  <choice>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int" minOccurs="0"/>
  </choice>
</complexType>
```

Defining attributes

In XML documents, attributes are contained in the element's tag. For example, in the **complexType** element in the code below, **name** is an attribute. To specify an attribute for a complex type, you define an **attribute** element in the **complexType** element definition. An **attribute** element can appear only after the **all**, **sequence**, or **choice** element. Specify one **attribute** element for each of the complex type's attributes. Any **attribute** elements must be direct children of the **complexType** element.

Example 2.8. Complex type with an attribute

```
<complexType name="personalInfo">
  <all>
    <element name="name" type="xsd:string"/>
    <element name="previousJobs" type="xsd:string">
      minOccurs="3" maxOccurs="7"/>
    </element>
  </all>
  <attribute name="age" type="xsd:int" use="required" />
</complexType>
```

In the previous code, the **attribute** element specifies that the **personalInfo** complex type has an **age** attribute. The **attribute** element has these attributes:

- **name** – A required attribute that specifies the string that identifies the attribute.
- **type** – Specifies the type of the data stored in the field. The type can be one of the XML Schema simple types.
- **use** – An optional attribute that specifies whether the complex type is required to have this attribute. Valid values are **required** or **optional**. The default is that the attribute is optional.

In an **attribute** element, you can specify the optional **default** attribute, which lets you specify a default value for the attribute.

2.5.2. Defining arrays

Overview

Apache CXF supports two methods for defining arrays in a contract. The first is to define a complex type with a single element whose **maxOccurs** attribute has a value greater than one. The second is to use **SOAP arrays**. **SOAP arrays** provide added functionality such as the ability to easily define multi-dimensional arrays and to transmit sparsely populated arrays.

Complex type arrays

Complex type arrays are a special case of a sequence complex type. You simply define a complex type with a single element and specify a value for the **maxOccurs** attribute. For example, to define an array of twenty floating point numbers you use a complex type similar to the one shown in [Example 2.9, "Complex type array"](#).

Example 2.9. Complex type array

```
<complexType name="personalInfo">
  <element name="averages" type="xsd:float" maxOccurs="20"/>
</complexType>
```

You can also specify a value for the **minOccurs** attribute.

SOAP arrays

SOAP arrays are defined by deriving from the **SOAP-ENC:Array** base type using the **wsdl:arrayType** element. The syntax for this is shown in [Example 2.10, "Syntax for a SOAP array derived using wsdl:arrayType"](#). Ensure that the **definitions** element declares **xmIIns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"**.

Example 2.10. Syntax for a SOAP array derived using wsdl:arrayType

```
<complexType name="TypeName">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
        wsdl:arrayType="ElementType<ArrayBounds>" />
    </restriction>
  </complexContent>
</complexType>
```

Using this syntax, *TypeName* specifies the name of the newly-defined array type. *ElementType* specifies the type of the elements in the array. *ArrayBounds* specifies the number of dimensions in the array. To specify a single dimension array use `[]`; to specify a two-dimensional array use either `[][]` or `[,]`.

For example, the SOAP Array, `SOAPStrings`, shown in [Example 2.11, “Definition of a SOAP array”](#), defines a one-dimensional array of strings. The **wsdl:arrayType** attribute specifies the type of the array elements, **xsd:string**, and the number of dimensions, with `[]` implying one dimension.

Example 2.11. Definition of a SOAP array

```
<complexType name="SOAPStrings">
<complexContent>
  <restriction base="SOAP-ENC:Array">
    <attribute ref="SOAP-ENC:arrayType"
      wsdl:arrayType="xsd:string[]"/>
  </restriction>
</complexContent>
</complexType>
```

You can also describe a SOAP Array using a simple element as described in the SOAP 1.1 specification. The syntax for this is shown in [Example 2.12, “Syntax for a SOAP array derived using an element”](#).

Example 2.12. Syntax for a SOAP array derived using an element

```
<complexType name="TypeName">
<complexContent>
  <restriction base="SOAP-ENC:Array">
    <sequence>
      <element name="ElementName" type="ElementType"
        maxOccurs="unbounded"/>
    </sequence>
  </restriction>
</complexContent>
</complexType>
```

When using this syntax, the element's **maxOccurs** attribute must always be set to **unbounded**.

2.5.3. Defining types by extension

Like most major coding languages, XML Schema allows you to create data types that inherit some of their elements from other data types. This is called defining a type by extension. For example, you could create a new type called **alienInfo**, that extends the **personallInfo** structure defined in [Example 2.4, “A complex type”](#) by adding a new element called **planet**.

Types defined by extension have four parts:

1. The name of the type is defined by the **name** attribute of the **complexType** element.
2. The **complexContent** element specifies that the new type will have more than one element.

**NOTE**

If you are only adding new attributes to the complex type, you can use a **simpleContent** element.

3. The type from which the new type is derived, called the **base type**, is specified in the **base** attribute of the **extension** element.
4. The new type's elements and attributes are defined in the **extension** element, the same as they are for a regular complex type.

For example, **alienInfo** is defined as shown in [Example 2.13, "Type defined by extension"](#).

Example 2.13. Type defined by extension

```
<complexType name="alienInfo">
  <complexContent>
    <extension base="xsd1:personallInfo">
      <sequence>
        <element name="planet" type="xsd:string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

2.5.4. Defining types by restriction**Overview**

XML Schema allows you to create new types by restricting the possible values of an XML Schema simple type. For example, you can define a simple type, **SSN**, which is a string of exactly nine characters. New types defined by restricting simple types are defined using a **simpleType** element.

The definition of a type by restriction requires three things:

1. The name of the new type is specified by the **name** attribute of the **simpleType** element.
2. The simple type from which the new type is derived, called the **base type**, is specified in the **restriction** element. See [the section called "Specifying the base type"](#).
3. The rules, called **facets**, defining the restrictions placed on the base type are defined as children of the **restriction** element. See [the section called "Defining the restrictions"](#).

Specifying the base type

The base type is the type that is being restricted to define the new type. It is specified using a **restriction** element. The **restriction** element is the only child of a **simpleType** element and has one attribute, **base**, that specifies the base type. The base type can be any of the XML Schema simple types.

For example, to define a new type by restricting the values of an **xsd:int** you use a definition like the one shown in [Example 2.14, "Using int as the base type"](#).

Example 2.14. Using int as the base type

```
<simpleType name="restrictedInt">
  <restriction base="xsd:int">
    ...
  </restriction>
</simpleType>
```

Defining the restrictions

The rules defining the restrictions placed on the base type are called **facets**. Facets are elements with one attribute, **value**, that defines how the facet is enforced. The available facets and their valid **value** settings depend on the base type. For example, **xsd:string** supports six facets, including:

- **length**
- **minLength**
- **maxLength**
- **pattern**
- **whitespace**
- **enumeration**

Each facet element is a child of the **restriction** element.

Example

[Example 2.15, “SSN simple type description”](#) shows an example of a simple type, **SSN**, which represents a social security number. The resulting type is a string of the form **XXX-XX-XXXX**. <SSN>032-43-9876</SSN> is a valid value for an element of this type, but <SSN>032439876</SSN> is not.

Example 2.15. SSN simple type description

```
<simpleType name="SSN">
  <restriction base="xsd:string">
    <pattern value="\d{3}-\d{2}-\d{4}"/>
  </restriction>
</simpleType>
```

2.5.5. Defining enumerated types

Overview

Enumerated types in XML Schema are a special case of definition by restriction. They are described by using the **enumeration** facet which is supported by all XML Schema primitive types. As with enumerated types in most modern programming languages, a variable of this type can only have one of the specified values.

Defining an enumeration in XML Schema

The syntax for defining an enumeration is shown in [Example 2.16, “Syntax for an enumeration”](#).

Example 2.16. Syntax for an enumeration

```
<simpleType name="EnumName">
  <restriction base="EnumType">
    <enumeration value="Case1Value"/>
    <enumeration value="Case2Value"/>
    ...
    <enumeration value="CaseNValue"/>
  </restriction>
</simpleType>
```

EnumName specifies the name of the enumeration type. *EnumType* specifies the type of the case values. *CaseNValue*, where *N* is any number one or greater, specifies the value for each specific case of the enumeration. An enumerated type can have any number of case values, but because it is derived from a simple type, only one of the case values is valid at a time.

Example

For example, an XML document with an element defined by the enumeration **widgetSize**, shown in [Example 2.17, “widgetSize enumeration”](#), would be valid if it contained <widgetSize>big</widgetSize>, but it would not be valid if it contained <widgetSize>big,mungo</widgetSize>.

Example 2.17. widgetSize enumeration

```
<simpleType name="widgetSize">
  <restriction base="xsd:string">
    <enumeration value="big"/>
    <enumeration value="large"/>
    <enumeration value="mungo"/>
  </restriction>
</simpleType>
```

2.6. DEFINING ELEMENTS

Elements in XML Schema represent an instance of an element in an XML document generated from the schema. The most basic element consists of a single **element** element. Like the **element** element used to define the members of a complex type, they have three attributes:

- **name** – A required attribute that specifies the name of the element as it appears in an XML document.
- **type** – Specifies the type of the element. The type can be any XML Schema primitive type or any named complex type defined in the contract. This attribute can be omitted if the type has an in-line definition.
- **nillable** – Specifies whether an element can be omitted from a document entirely. If **nillable** is set to **true**, the element can be omitted from any document generated using the schema.

An element can also have an **in-line** type definition. In-line types are specified using either a **complexType** element or a **simpleType** element. Once you specify if the type of data is complex or simple, you can define any type of data needed using the tools available for each type of data. In-line type definitions are discouraged because they are not reusable.

CHAPTER 3. DEFINING LOGICAL MESSAGES USED BY A SERVICE

Abstract

A service is defined by the messages exchanged when its operations are invoked. In a WSDL contract these messages are defined using **message** element. The messages are made up of one or more parts that are defined using **part** elements.

OVERVIEW

A service's operations are defined by specifying the logical messages that are exchanged when an operation is invoked. These logical messages define the data that is passed over a network as an XML document. They contain all of the parameters that are a part of a method invocation. Logical messages are defined using the **message** element in your contracts. Each logical message consists of one or more parts, defined in **part** elements.

While your messages can list each parameter as a separate part, the recommended practice is to use only a single part that encapsulates the data needed for the operation.

MESSAGES AND PARAMETER LISTS

Each operation exposed by a service can have only one input message and one output message. The input message defines all of the information the service receives when the operation is invoked. The output message defines all of the data that the service returns when the operation is completed. Fault messages define the data that the service returns when an error occurs.

In addition, each operation can have any number of fault messages. The fault messages define the data that is returned when the service encounters an error. These messages usually have only one part that provides enough information for the consumer to understand the error.

MESSAGE DESIGN FOR INTEGRATING WITH LEGACY SYSTEMS

If you are defining an existing application as a service, you must ensure that each parameter used by the method implementing the operation is represented in a message. You must also ensure that the return value is included in the operation's output message.

One approach to defining your messages is RPC style. When using RPC style, you define the messages using one part for each parameter in the method's parameter list. Each message part is based on a type defined in the **types** element of the contract. Your input message contains one part for each input parameter in the method. Your output message contains one part for each output parameter, plus a part to represent the return value, if needed. If a parameter is both an input and an output parameter, it is listed as a part for both the input message and the output message.

RPC style message definition is useful when service enabling legacy systems that use transports such as Tibco or CORBA. These systems are designed around procedures and methods. As such, they are easiest to model using messages that resemble the parameter lists for the operation being invoked. RPC style also makes a cleaner mapping between the service and the application it is exposing.

MESSAGE DESIGN FOR SOAP SERVICES

While RPC style is useful for modeling existing systems, the service's community strongly favors the wrapped document style. In wrapped document style, each message has a single part. The message's

part references a wrapper element defined in the **types** element of the contract. The wrapper element has the following characteristics:

- It is a complex type containing a sequence of elements. For more information see [Section 2.5, "Defining complex data types"](#).
- If it is a wrapper for an input message:
 - It has one element for each of the method's input parameters.
 - Its name is the same as the name of the operation with which it is associated.
- If it is a wrapper for an output message:
 - It has one element for each of the method's output parameters and one element for each of the method's inout parameters.
 - Its first element represents the method's return parameter.
 - Its name would be generated by appending **Response** to the name of the operation with which the wrapper is associated.

MESSAGE NAMING

Each message in a contract must have a unique name within its namespace. It is recommended that you use the following naming conventions:

- Messages should only be used by a single operation.
- Input message names are formed by appending **Request** to the name of the operation.
- Output message names are formed by appending **Response** to the name of the operation.
- Fault message names should represent the reason for the fault.

MESSAGE PARTS

Message parts are the formal data units of the logical message. Each part is defined using a **part** element, and is identified by a **name** attribute and either a **type** attribute or an **element** attribute that specifies its data type. The data type attributes are listed in [Table 3.1, "Part data type attributes"](#).

Table 3.1. Part data type attributes

Attribute	Description
element =" <i>elem_name</i> "	The data type of the part is defined by an element called <i>elem_name</i> .
type =" <i>type_name</i> "	The data type of the part is defined by a type called <i>type_name</i> .

Messages are allowed to reuse part names. For instance, if a method has a parameter, **foo**, that is passed by reference or is an in/out, it can be a part in both the request message and the response message, as shown in [Example 3.1, "Reused part"](#).

Example 3.1. Reused part

```
<message name="fooRequest">
  <part name="foo" type="xsd:int"/>
<message>
<message name="fooReply">
  <part name="foo" type="xsd:int"/>
<message>
```

EXAMPLE

For example, imagine you had a server that stored personal information and provided a method that returned an employee's data based on the employee's ID number. The method signature for looking up the data is similar to [Example 3.2, "personallInfo lookup method"](#).

Example 3.2. personallInfo lookup method

```
personallInfo lookup(long empld)
```

This method signature can be mapped to the RPC style WSDL fragment shown in [Example 3.3, "RPC WSDL message definitions"](#).

Example 3.3. RPC WSDL message definitions

```
<message name="personalLookupRequest">
  <part name="empld" type="xsd:int"/>
<message/>
<message name="personalLookupResponse">
  <part name="return" element="xsd1:personallInfo"/>
<message/>
```

It can also be mapped to the wrapped document style WSDL fragment shown in [Example 3.4, "Wrapped document WSDL message definitions"](#).

Example 3.4. Wrapped document WSDL message definitions

```
<wsdl:types>
  <xsd:schema ... >
  ...
  <element name="personalLookup">
    <complexType>
      <sequence>
        <element name="emplID" type="xsd:int" />
      </sequence>
    </complexType>
  </element>
  <element name="personalLookupResponse">
    <complexType>
      <sequence>
```

```
<element name="return" type="personalInfo" />
</sequence>
</complexType>
</element>
</schema>
</types>
<wsdl:message name="personalLookupRequest">
  <wsdl:part name="emplId" element="xsd1:personalLookup"/>
<message/>
<wsdl:message name="personalLookupResponse">
  <wsdl:part name="return" element="xsd1:personalLookupResponse"/>
<message/>
```

CHAPTER 4. DEFINING YOUR LOGICAL INTERFACES

Abstract

Logical service interfaces are defined using the **portType** element.

OVERVIEW

Logical service interfaces are defined using the WSDL **portType** element. The **portType** element is a collection of abstract operation definitions. Each operation is defined by the input, output, and fault messages used to complete the transaction the operation represents. When code is generated to implement the service interface defined by a **portType** element, each operation is converted into a method containing the parameters defined by the input, output, and fault messages specified in the contract.

PROCESS

To define a logical interface in a **WSDL** contract you must do the following:

1. Create a **portType** element to contain the interface definition and give it a unique name. See [the section called “Port types”](#).
2. Create an **operation** element for each operation defined in the interface. See [the section called “Operations”](#).
3. For each operation, specify the messages used to represent the operation’s parameter list, return type, and exceptions. See [the section called “Operation messages”](#).

POR T TYPES

A WSDL **portType** element is the root element in a logical interface definition. While many Web service implementations map **portType** elements directly to generated implementation objects, a logical interface definition does not specify the exact functionality provided by the the implemented service. For example, a logical interface named ticketSystem can result in an implementation that either sells concert tickets or issues parking tickets.

The **portType** element is the unit of a WSDL document that is mapped into a binding to define the physical data used by an endpoint exposing the defined service.

Each **portType** element in a WSDL document must have a unique name, which is specified using the **name** attribute, and is made up of a collection of operations, which are described in **operation** elements. A WSDL document can describe any number of port types.

OPERATIONS

Logical operations, defined using WSDL **operation** elements, define the interaction between two endpoints. For example, a request for a checking account balance and an order for a gross of widgets can both be defined as operations.

Each operation defined within a **portType** element must have a unique name, specified using the **name** attribute. The **name** attribute is required to define an operation.

OPERATION MESSAGES

Logical operations are made up of a set of elements representing the logical messages communicated between the endpoints to execute the operation. The elements that can describe an operation are listed in [Table 4.1, "Operation message elements"](#).

Table 4.1. Operation message elements

Element	Description
input	Specifies the message the client endpoint sends to the service provider when a request is made. The parts of this message correspond to the input parameters of the operation.
output	Specifies the message that the service provider sends to the client endpoint in response to a request. The parts of this message correspond to any operation parameters that can be changed by the service provider, such as values passed by reference. This includes the return value of the operation.
fault	Specifies a message used to communicate an error condition between the endpoints.

An operation is required to have at least one **input** or one **output** element. An operation can have both **input** and **output** elements, but it can only have one of each. Operations are not required to have any **fault** elements, but can, if required, have any number of **fault** elements.

The elements have the two attributes listed in [Table 4.2, "Attributes of the input and output elements"](#).

Table 4.2. Attributes of the input and output elements

Attribute	Description
name	Identifies the message so it can be referenced when mapping the operation to a concrete data format. The name must be unique within the enclosing port type.
message	Specifies the abstract message that describes the data being sent or received. The value of the message attribute must correspond to the name attribute of one of the abstract messages defined in the WSDL document.

It is not necessary to specify the **name** attribute for all **input** and **output** elements; WSDL provides a default naming scheme based on the enclosing operation's name. If only one element is used in the operation, the element name defaults to the name of the operation. If both an **input** and an **output** element are used, the element name defaults to the name of the operation with either **Request** or **Response** respectively appended to the name.

RETURN VALUES

Because the **operation** element is an abstract definition of the data passed during an operation, WSDL does not provide for return values to be specified for an operation. If a method returns a value it will be mapped into the **output** element as the last part of that message.

EXAMPLE

For example, you might have an interface similar to the one shown in [Example 4.1, “personalInfo lookup interface”](#).

Example 4.1. personalInfo lookup interface

```
interface personalInfoLookup
{
    personalInfo lookup(in int empID)
    raises(idNotFound);
}
```

This interface can be mapped to the port type in [Example 4.2, “personalInfo lookup port type”](#).

Example 4.2. personalInfo lookup port type

```
<message name="personalLookupRequest">
    <part name="empId" element="xsd1:personalLookup"/>
<message/>
<message name="personalLookupResponse">
    <part name="return" element="xsd1:personalLookupResponse"/>
<message/>
<message name="idNotFoundException">
    <part name="exception" element="xsd1:idNotFound"/>
<message/>
<portType name="personalInfoLookup">
    <operation name="lookup">
        <input name="empID" message="tns:personalLookupRequest"/>
        <output name="return" message="tns:personalLookupResponse"/>
        <fault name="exception" message="tns:idNotFoundException"/>
    </operation>
</portType>
```

PART II. WEB SERVICES BINDINGS

This part describes how to add Apache CXF bindings to a WSDL document.

CHAPTER 5. UNDERSTANDING BINDINGS IN WSDL

Abstract

Bindings map the logical messages used to define a service into a concrete payload format that can be transmitted and received by an endpoint.

OVERVIEW

Bindings provide a bridge between the logical messages used by a service to a concrete data format that an endpoint uses in the physical world. They describe how the logical messages are mapped into a payload format that is used on the wire by an endpoint. It is within the bindings that details such as parameter order, concrete data types, and return values are specified. For example, the parts of a message can be reordered in a binding to reflect the order required by an RPC call. Depending on the binding type, you can also identify which of the message parts, if any, represent the return type of a method.

PORT TYPES AND BINDINGS

Port types and bindings are directly related. A port type is an abstract definition of a set of interactions between two logical services. A binding is a concrete definition of how the messages used to implement the logical services will be instantiated in the physical world. Each binding is then associated with a set of network details that finish the definition of one endpoint that exposes the logical service defined by the port type.

To ensure that an endpoint defines only a single service, WSDL requires that a binding can only represent a single port type. For example, if you had a contract with two port types, you could not write a single binding that mapped both of them into a concrete data format. You would need two bindings.

However, WSDL allows for a port type to be mapped to several bindings. For example, if your contract had a single port type, you could map it into two or more bindings. Each binding could alter how the parts of the message are mapped or they could specify entirely different payload formats for the message.

THE WSDL ELEMENTS

Bindings are defined in a contract using the WSDL **binding** element. The binding element consists of attributes like, **name**, that specifies a unique name for the binding and **type** that provides reference to PortType. The value of this attribute is used to associate the binding with an endpoint as discussed in [Chapter 4, Defining Your Logical Interfaces](#).

The actual mappings are defined in the children of the **binding** element. These elements vary depending on the type of payload format you decide to use. The different payload formats and the elements used to specify their mappings are discussed in the following chapters.

ADDING TO A CONTRACT

Apache CXF provides command line tools that can generate bindings for predefined service interfaces.

The tools will add the proper elements to your contract for you. However, it is recommended that you have some knowledge of how the different types of bindings work.

You can also add a binding to a contract using any text editor. When hand editing a contract, you are responsible for ensuring that the contract is valid.

SUPPORTED BINDINGS

Apache CXF supports the following bindings:

- SOAP 1.1
- SOAP 1.2
- CORBA
- Pure XML

CHAPTER 6. USING SOAP 1.1 MESSAGES

Abstract

Apache CXF provides a tool to generate a SOAP 1.1 binding which does not use any SOAP headers. However, you can add SOAP headers to your binding using any text or XML editor.

6.1. ADDING A SOAP 1.1 BINDING

Using wsdl2soap

To generate a SOAP 1.1 binding using **wsdl2soap** use the following command: **wsdl2soap -i port-type-name -b binding-name -d output-directory -o output-file -n soap-body-namespace -s style** (**document/rpc**) -use (**literal/encoded**) -v verbose -quiet **wsdlurl**



NOTE

To use **wsdl2soap** you will need to download the Apache CXF distribution.

The command has the following options:

Option	Interpretation
-i <i>port-type-name</i>	Specifies the portType element for which a binding is generated.
wsdlurl	The path and name of the WSDL file containing the portType element definition.

The tool has the following optional arguments:

Option	Interpretation
-b <i>binding-name</i>	Specifies the name of the generated SOAP binding.
-d <i>output-directory</i>	Specifies the directory to place the generated WSDL file.
-o <i>output-file</i>	Specifies the name of the generated WSDL file.
-n <i>soap-body-namespace</i>	Specifies the SOAP body namespace when the style is RPC.
-style (document/rpc)	Specifies the encoding style (document or RPC) to use in the SOAP binding. The default is document.
-use (literal/encoded)	Specifies the binding use (encoded or literal) to use in the SOAP binding. The default is literal.

Option	Interpretation
-v	Displays the version number for the tool.
-verbose	Displays comments during the code generation process.
-quiet	Suppresses comments during the code generation process.

The **-iport-type-name** and **wsdlurl** arguments are required. If the **-style rpc** argument is specified, the **-nsoap-body-namespace** argument is also required. All other arguments are optional and may be listed in any order.



IMPORTANT

wsdl2soap does not support the generation of **document/encoded** SOAP bindings.

Example

If your system has an interface that takes orders and offers a single operation to process the orders it is defined in a **WSDL** fragment similar to the one shown in [Example 6.1, "Ordering System Interface"](#).

Example 6.1. Ordering System Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
  targetNamespace="http://widgetVendor.com/widgetOrderForm"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://widgetVendor.com/widgetOrderForm"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding">

  <message name="widgetOrder">
    <part name="numOrdered" type="xsd:int"/>
  </message>
  <message name="widgetOrderBill">
    <part name="price" type="xsd:float"/>
  </message>
  <message name="badSize">
    <part name="numInventory" type="xsd:int"/>
  </message>

  <portType name="orderWidgets">
    <operation name="placeWidgetOrder">
      <input message="tns:widgetOrder" name="order"/>
      <output message="tns:widgetOrderBill" name="bill"/>
      <fault message="tns:badSize" name="sizeFault"/>
    </operation>
  </portType>
</definitions>
```

```

</portType>
...
</definitions>
```

The SOAP binding generated for **orderWidgets** is shown in [Example 6.2, "SOAP 1.1 Binding for orderWidgets"](#).

Example 6.2. SOAP 1.1 Binding for orderWidgets

```

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="order">
        <soap:body use="literal"/>
      </input>
      <output name="bill">
        <soap:body use="literal"/>
      </output>
      <fault name="sizeFault">
        <soap:body use="literal"/>
      </fault>
    </operation>
  </binding>
```

This binding specifies that messages are sent using the **document/literal** message style.

6.2. ADDING SOAP HEADERS TO A SOAP 1.1 BINDING

Overview

SOAP headers are defined by adding **soap:header** elements to your default SOAP 1.1 binding. The **soap:header** element is an optional child of the **input**, **output**, and **fault** elements of the binding. The SOAP header becomes part of the parent message. A SOAP header is defined by specifying a message and a message part. Each SOAP header can only contain one message part, but you can insert as many SOAP headers as needed.

Syntax

The syntax for defining a SOAP header is shown in [Example 6.3, "SOAP Header Syntax"](#). The **message** attribute of **soap:header** is the qualified name of the message from which the part being inserted into the header is taken. The **part** attribute is the name of the message part inserted into the SOAP header. Because SOAP headers are always document style, the WSDL message part inserted into the SOAP header must be defined using an element. Together the **message** and the **part** attributes fully describe the data to insert into the SOAP header.

Example 6.3. SOAP Header Syntax

```

<binding name="headwig">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
```

```

<operation name="weave">
  <soap:operation soapAction="" style="document"/>
  <input name="grain">
    <soap:body ... />
    <soap:header message="QName" part="partName"/>
  </input>
  ...
</binding>

```

As well as the mandatory **message** and **part** attributes, **soap:header** also supports the **namespace**, the **use**, and the **encodingStyle** attributes. These attributes function the same for **soap:header** as they do for **soap:body**.

Splitting messages between body and header

The message part inserted into the SOAP header can be any valid message part from the contract. It can even be a part from the parent message which is being used as the SOAP body. Because it is unlikely that you would want to send information twice in the same message, the SOAP binding provides a means for specifying the message parts that are inserted into the SOAP body.

The **soap:body** element has an optional attribute, **parts**, that takes a space delimited list of part names. When **parts** is defined, only the message parts listed are inserted into the SOAP body. You can then insert the remaining parts into the SOAP header.



NOTE

When you define a SOAP header using parts of the parent message, Apache CXF automatically fills in the SOAP headers for you.

Example

[Example 6.4, "SOAP 1.1 Binding with a SOAP Header"](#) shows a modified version of the **orderWidgets** service shown in [Example 6.1, "Ordering System Interface"](#). This version has been modified so that each order has an **xsd:base64binary** value placed in the SOAP header of the request and response. The SOAP header is defined as being the **keyVal** part from the **widgetKey** message. In this case you are responsible for adding the SOAP header to your application logic because it is not part of the input or output message.

Example 6.4. SOAP 1.1 Binding with a SOAP Header

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
  targetNamespace="http://widgetVendor.com/widgetOrderForm"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://widgetVendor.com/widgetOrderForm"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
    xmlns="http://www.w3.org/2001/XMLSchema"

```

```
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
</schema>
</types>

<message name="widgetOrder">
    <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
    <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
    <part name="numInventory" type="xsd:int"/>
</message>
<message name="widgetKey">
    <part name="keyVal" element="xsd1:keyElem"/>
</message>

<portType name="orderWidgets">
    <operation name="placeWidgetOrder">
        <input message="tns:widgetOrder" name="order"/>
        <output message="tns:widgetOrderBill" name="bill"/>
        <fault message="tns:badSize" name="sizeFault"/>
    </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
        <soap:operation soapAction="" style="document"/>
        <input name="order">
            <soap:body use="literal"/>
            <soap:header message="tns:widgetKey" part="keyVal"/>
        </input>
        <output name="bill">
            <soap:body use="literal"/>
            <soap:header message="tns:widgetKey" part="keyVal"/>
        </output>
        <fault name="sizeFault">
            <soap:body use="literal"/>
        </fault>
    </operation>
</binding>
...
</definitions>
```

You can also modify [Example 6.4, “SOAP 1.1 Binding with a SOAP Header”](#) so that the header value is a part of the input and output messages.

CHAPTER 7. USING SOAP 1.2 MESSAGES

Abstract

Apache CXF provides tools to generate a SOAP 1.2 binding which does not use any SOAP headers. You can add SOAP headers to your binding using any text or XML editor.

7.1. ADDING A SOAP 1.2 BINDING TO A WSDL DOCUMENT

Using wsdl2soap



NOTE

To use wsdl2soap you will need to download the Apache CXF distribution.

To generate a SOAP 1.2 binding using **wsdl2soap** use the following command: **wsdl2soap -i port-type-name -b binding-name -s soap12 -d output-directory -o output-file -n soap-body-namespace -s style (document/rpc) -use (literal/encoded) -v verbose -quiet wsdlurl** The tool has the following required arguments:

Option	Interpretation
-i <i>port-type-name</i>	Specifies the portType element for which a binding is generated.
-soap12	Specifies that the generated binding uses SOAP 1.2.
<i>wsdlurl</i>	The path and name of the WSDL file containing the portType element definition.

The tool has the following optional arguments:

Option	Interpretation
-b <i>binding-name</i>	Specifies the name of the generated SOAP binding.
-soap12	Specifies that the generated binding will use SOAP 1.2.
-d <i>output-directory</i>	Specifies the directory to place the generated WSDL file.
-o <i>output-file</i>	Specifies the name of the generated WSDL file.
-n <i>soap-body-namespace</i>	Specifies the SOAP body namespace when the style is RPC.

Option	Interpretation
-style (document/rpc)	Specifies the encoding style (document or RPC) to use in the SOAP binding. The default is document.
-use (literal/encoded)	Specifies the binding use (encoded or literal) to use in the SOAP binding. The default is literal.
-v	Displays the version number for the tool.
-verbose	Displays comments during the code generation process.
-quiet	Suppresses comments during the code generation process.

The **-i port-type-name** and **wsdl:url** arguments are required. If the **-style rpc** argument is specified, the **-n soap-body-namespace** argument is also required. All other arguments are optional and can be listed in any order.



IMPORTANT

wsdl2soap does not support the generation of **document/encoded** SOAP 1.2 bindings.

Example

If your system has an interface that takes orders and offers a single operation to process the orders it is defined in a **WSDL** fragment similar to the one shown in [Example 7.1, "Ordering System Interface"](#).

Example 7.1. Ordering System Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
  targetNamespace="http://widgetVendor.com/widgetOrderForm"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:tns="http://widgetVendor.com/widgetOrderForm"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

  <message name="widgetOrder">
    <part name="numOrdered" type="xsd:int"/>
  </message>
  <message name="widgetOrderBill">
    <part name="price" type="xsd:float"/>
  </message>
  <message name="badSize">
    <part name="numInventory" type="xsd:int"/>
  </message>

  <portType name="orderWidgets">
```

```

<operation name="placeWidgetOrder">
  <input message="tns:widgetOrder" name="order"/>
  <output message="tns:widgetOrderBill" name="bill"/>
  <fault message="tns:badSize" name="sizeFault"/>
</operation>
</portType>
...
</definitions>

```

The SOAP binding generated for `orderWidgets` is shown in [Example 7.2, "SOAP 1.2 Binding for `orderWidgets`".](#)

Example 7.2. SOAP 1.2 Binding for `orderWidgets`

```

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap12:operation soapAction="" style="document"/>
      <input name="order">
        <soap12:body use="literal"/>
      </input>
      <output name="bill">
        <wsoap12:body use="literal"/>
      </output>
      <fault name="sizeFault">
        <soap12:body use="literal"/>
      </fault>
    </operation>
  </binding>

```

This binding specifies that messages are sent using the **document/literal** message style.

7.2. ADDING HEADERS TO A SOAP 1.2 MESSAGE

Overview

SOAP message headers are defined by adding **soap12:header** elements to your SOAP 1.2 message. The **soap12:header** element is an optional child of the **input**, **output**, and **fault** elements of the binding. The SOAP header becomes part of the parent message. A SOAP header is defined by specifying a message and a message part. Each SOAP header can only contain one message part, but you can insert as many headers as needed.

Syntax

The syntax for defining a SOAP header is shown in [Example 7.3, "SOAP Header Syntax".](#)

Example 7.3. SOAP Header Syntax

```

<binding name="headwig">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="weave">

```

```

<soap12:operation soapAction="" style="document"/>
<input name="grain">
  <soap12:body ... />
  <soap12:header message="QName" part="partName"
    use="literal|encoded"
    encodingStyle="encodingURI"
    namespace="namespaceURI" />
</input>
...
</binding>

```

The **soap12:header** element's attributes are described in [Table 7.1, “soap12:header Attributes”](#).

Table 7.1. soap12:header Attributes

Attribute	Description
message	A required attribute specifying the qualified name of the message from which the part being inserted into the header is taken.
part	A required attribute specifying the name of the message part inserted into the SOAP header.
use	Specifies if the message parts are to be encoded using encoding rules. If set to encoded the message parts are encoded using the encoding rules specified by the value of the encodingStyle attribute. If set to literal , the message parts are defined by the schema types referenced.
encodingStyle	Specifies the encoding rules used to construct the message.
namespace	Defines the namespace to be assigned to the header element serialized with use="encoded" .

Splitting messages between body and header

The message part inserted into the SOAP header can be any valid message part from the contract. It can even be a part from the parent message which is being used as the SOAP body. Because it is unlikely that you would send information twice in the same message, the SOAP 1.2 binding provides a means for specifying the message parts that are inserted into the SOAP body.

The **soap12:body** element has an optional attribute, **parts**, that takes a space delimited list of part names. When **parts** is defined, only the message parts listed are inserted into the body of the SOAP 1.2 message. You can then insert the remaining parts into the message's header.



NOTE

When you define a SOAP header using parts of the parent message, Apache CXF automatically fills in the SOAP headers for you.

Example

[Example 7.4, "SOAP 1.2 Binding with a SOAP Header"](#) shows a modified version of the **orderWidgets** service shown in [Example 7.1, "Ordering System Interface"](#). This version is modified so that each order has an **xsd:base64binary** value placed in the header of the request and the response. The header is defined as being the **keyVal** part from the **widgetKey** message. In this case you are responsible for adding the application logic to create the header because it is not part of the input or output message.

Example 7.4. SOAP 1.2 Binding with a SOAP Header

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding"/>

<types>
    <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
        <element name="keyElem" type="xsd:base64Binary"/>
    </schema>
</types>

<message name="widgetOrder">
    <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
    <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
    <part name="numInventory" type="xsd:int"/>
</message>
<message name="widgetKey">
    <part name="keyVal" element="xsd1:keyElem"/>
</message>

<portType name="orderWidgets">
    <operation name="placeWidgetOrder">
        <input message="tns:widgetOrder" name="order"/>
        <output message="tns:widgetOrderBill" name="bill"/>
        <fault message="tns:badSize" name="sizeFault"/>
    </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
```

```

<soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="placeWidgetOrder">
  <soap12:operation soapAction="" style="document"/>
  <input name="order">
    <soap12:body use="literal"/>
    <soap12:header message="tns:widgetKey" part="keyVal"/>
  </input>
  <output name="bill">
    <soap12:body use="literal"/>
    <soap12:header message="tns:widgetKey" part="keyVal"/>
  </output>
  <fault name="sizeFault">
    <soap12:body use="literal"/>
  </fault>
</operation>
</binding>
...
</definitions>

```

You can modify [Example 7.4, "SOAP 1.2 Binding with a SOAP Header"](#) so that the header value is a part of the input and output messages, as shown in [Example 7.5, "SOAP 1.2 Binding for orderWidgets with a SOAP Header"](#). In this case **keyVal** is a part of the input and output messages. In the **soap12:body** elements the **parts** attribute specifies that **keyVal** should not be inserted into the body. However, it is inserted into the header.

Example 7.5. SOAP 1.2 Binding for orderWidgets with a SOAP Header

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
  targetNamespace="http://widgetVendor.com/widgetOrderForm"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:tns="http://widgetVendor.com/widgetOrderForm"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

  <types>
    <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <element name="keyElem" type="xsd:base64Binary"/>
    </schema>
  </types>

  <message name="widgetOrder">
    <part name="numOrdered" type="xsd:int"/>
    <part name="keyVal" element="xsd1:keyElem"/>
  </message>
  <message name="widgetOrderBill">
    <part name="price" type="xsd:float"/>
    <part name="keyVal" element="xsd1:keyElem"/>
  </message>
  <message name="badSize">

```

```
<part name="numInventory" type="xsd:int"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap12:operation soapAction="" style="document"/>
      <input name="order">
        <soap12:body use="literal" parts="numOrdered"/>
        <soap12:header message="tns:widgetOrder" part="keyVal"/>
      </input>
      <output name="bill">
        <soap12:body use="literal" parts="bill"/>
        <soap12:header message="tns:widgetOrderBill" part="keyVal"/>
      </output>
      <fault name="sizeFault">
        <soap12:body use="literal"/>
      </fault>
    </operation>
  </binding>
  ...
</definitions>
```

CHAPTER 8. SENDING BINARY DATA USING SOAP WITH ATTACHMENTS

Abstract

SOAP attachments provide a mechanism for sending binary data as part of a SOAP message. Using SOAP with attachments requires that you define your SOAP messages as MIME multipart messages.

OVERVIEW

SOAP messages generally do not carry binary data. However, the W3C SOAP 1.1 specification allows for using MIME multipart/related messages to send binary data in SOAP messages. This technique is called using SOAP with attachments. SOAP attachments are defined in the W3C's [SOAP Messages with Attachments Note](#).

NAMESPACE

The WSDL extensions used to define the MIME multipart/related messages are defined in the namespace <http://schemas.xmlsoap.org/wsdl/mime/>.

In the discussion that follows, it is assumed that this namespace is prefixed with **mime**. The entry in the WSDL **definitions** element to set this up is shown in [Example 8.1, "MIME Namespace Specification in a Contract"](#).

Example 8.1. MIME Namespace Specification in a Contract

```
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
```

CHANGING THE MESSAGE BINDING

In a default SOAP binding, the first child element of the **input**, **output**, and **fault** elements is a **soap:body** element describing the body of the SOAP message representing the data. When using SOAP with attachments, the **soap:body** element is replaced with a **mime:multipartRelated** element.



NOTE

WSDL does not support using **mime:multipartRelated** for **fault** messages.

The **mime:multipartRelated** element tells Apache CXF that the message body is a multipart message that potentially contains binary data. The contents of the element define the parts of the message and their contents. **mime:multipartRelated** elements contain one or more **mime:part** elements that describe the individual parts of the message.

The first **mime:part** element must contain the **soap:body** element that would normally appear in a default SOAP binding. The remaining **mime:part** elements define the attachments that are being sent in the message.

DESCRIBING A MIME MULTIPART MESSAGE

MIME multipart messages are described using a **mime:multipartRelated** element that contains a number of **mime:part** elements. To fully describe a MIME multipart message you must do the following:

1. Inside the **input** or **output** message you are sending as a MIME multipart message, add a **mime:multipartRelated** element as the first child element of the enclosing message.
2. Add a **mime:part** child element to the **mime:multipartRelated** element and set its **name** attribute to a unique string.
3. Add a **soap:body** element as the child of the **mime:part** element and set its attributes appropriately.



NOTE

If the contract had a default SOAP binding, you can copy the **soap:body** element from the corresponding message from the default binding into the MIME multipart message.

4. Add another **mime:part** child element to the **mime:multipartRelated** element and set its **name** attribute to a unique string.
5. Add a **mime:content** child element to the **mime:part** element to describe the contents of this part of the message.

To fully describe the contents of a MIME message part the **mime:content** element has the following attributes:

Table 8.1. mime:content Attributes

Attribute	Description +
part	Specifies the name of the WSDL message part , from the parent message definition, that is used as the content of this part of the MIME multipart message being placed on the wire. + +
type	The MIME type of the data in this message part. MIME types are defined as a type and a subtype using the syntax <i>type/subtype</i> . + There are a number of predefined MIME types such as image/jpeg and text/plain . The MIME types are maintained by the Internet Assigned Numbers Authority (IANA) and described in detail in Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies and Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types .
	+ +

6. For each additional MIME part, repeat steps [i303819] and [i303821].

EXAMPLE

Example 8.2, “Contract using SOAP with Attachments” shows a WSDL fragment defining a service that stores X-rays in JPEG format. The image data, **xRay**, is stored as an **xsd:base64binary** and is packed into the MIME multipart message’s second part, **imageData**. The remaining two parts of the input message, **patientName** and **patientNumber**, are sent in the first part of the MIME multipart image as part of the SOAP body.

Example 8.2. Contract using SOAP with Attachments

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="XrayStorage"
  targetNamespace="http://mediStor.org/x-rays"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://mediStor.org/x-rays"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <message name="storRequest">
    <part name="patientName" type="xsd:string"/>
    <part name="patientNumber" type="xsd:int"/>
    <part name="xRay" type="xsd:base64Binary"/>
  </message>
  <message name="storResponse">
    <part name="success" type="xsd:boolean"/>
  </message>

  <portType name="xRayStorage">
    <operation name="store">
      <input message="tns:storRequest" name="storRequest"/>
      <output message="tns:storResponse" name="storResponse"/>
    </operation>
  </portType>

  <binding name="xRayStorageBinding" type="tns:xRayStorage">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="store">
      <soap:operation soapAction="" style="document"/>
      <input name="storRequest">
        <mime:multipartRelated>
          <mime:part name="bodyPart">
            <soap:body use="literal"/>
          </mime:part>
          <mime:part name="imageData">
            <mime:content part="xRay" type="image/jpeg"/>
          </mime:part>
        </mime:multipartRelated>
      </input>
      <output name="storResponse">
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>

  <service name="xRayStorageService">
```

```
<port binding="tns:xRayStorageBinding" name="xRayStoragePort">
  <soap:address location="http://localhost:9000"/>
</port>
</service>
</definitions>
```

CHAPTER 9. SENDING BINARY DATA WITH SOAP MTOM

Abstract

SOAP Message Transmission Optimization Mechanism (MTOM) replaces SOAP with attachments as a mechanism for sending binary data as part of an XML message. Using MTOM with Apache CXF requires adding the correct schema types to a service's contract and enabling the MTOM optimizations.

9.1. OVERVIEW OF MTOM

SOAP Message Transmission Optimization Mechanism (MTOM) specifies an optimized method for sending binary data as part of a SOAP message. Unlike SOAP with Attachments, MTOM requires the use of XML-binary Optimized Packaging (XOP) packages for transmitting binary data. Using MTOM to send binary data does not require you to fully define the MIME Multipart/Related message as part of the SOAP binding. It does, however, require that you do the following:

1. [Annotate](#) the data that you are going to send as an attachment.
You can annotate either your WSDL or the Java class that implements your data.
2. [Enable](#) the runtime's MTOM support.
This can be done either programmatically or through configuration.
3. Develop a **DataHandler** for the data being passed as an attachment.



NOTE

Developing **DataHandlers** is beyond the scope of this book.

9.2. ANNOTATING DATA TYPES TO USE MTOM

Overview

In WSDL, when defining a data type for passing along a block of binary data, such as an image file or a sound file, you define the element for the data to be of type **xsd:base64Binary**. By default, any element of type **xsd:base64Binary** results in the generation of a **byte[]** which can be serialized using MTOM. However, the default behavior of the code generators does not take full advantage of the serialization.

In order to fully take advantage of MTOM you must add annotations to either your service's WSDL document or the JAXB class that implements the binary data structure. Adding the annotations to the WSDL document forces the code generators to generate streaming data handlers for the binary data. Annotating the JAXB class involves specifying the proper content types and might also involve changing the type specification of the field containing the binary data.

WSDL first

[Example 9.1, "Message for MTOM"](#) shows a WSDL document for a Web service that uses a message which contains one string field, one integer field, and a binary field. The binary field is intended to carry a large image file, so it is not appropriate to send it as part of a normal SOAP message.

Example 9.1. Message for MTOM

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<definitions name="XrayStorage"
    targetNamespace="http://mediStor.org/x-rays"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://mediStor.org/x-rays"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:xsd1="http://mediStor.org/types/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <types>
        <schema targetNamespace="http://mediStor.org/types/"
            xmlns="http://www.w3.org/2001/XMLSchema">
            <complexType name="xRayType">
                <sequence>
                    <element name="patientName" type="xsd:string" />
                    <element name="patientNumber" type="xsd:int" />
                    <element name="imageData" type="xsd:base64Binary" />
                </sequence>
            </complexType>
            <element name="xRay" type="xsd1:xRayType" />
        </schema>
    </types>

    <message name="storRequest">
        <part name="record" element="xsd1:xRay"/>
    </message>
    <message name="storResponse">
        <part name="success" type="xsd:boolean"/>
    </message>

    <portType name="xRayStorage">
        <operation name="store">
            <input message="tns:storRequest" name="storRequest"/>
            <output message="tns:storResponse" name="storResponse"/>
        </operation>
    </portType>

    <binding name="xRayStorageSOAPBinding" type="tns:xRayStorage">
        <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="store">
            <soap12:operation soapAction="" style="document"/>
            <input name="storRequest">
                <soap12:body use="literal"/>
            </input>
            <output name="storResponse">
                <soap12:body use="literal"/>
            </output>
        </operation>
    </binding>
    ...
</definitions>

```

If you want to use MTOM to send the binary part of the message as an optimized attachment you must add the **xmime:expectedContentTypes** attribute to the element containing the binary data. This attribute is defined in the <http://www.w3.org/2005/05/xmlmime> namespace and specifies the MIME types that the element is expected to contain. You can specify a comma separated list of MIME types.

The setting of this attribute changes how the code generators create the JAXB class for the data. For most MIME types, the code generator creates a DataHandler. Some MIME types, such as those for images, have defined mappings.



NOTE

The MIME types are maintained by the Internet Assigned Numbers Authority(IANA) and are described in detail in [Multipurpose Internet Mail Extensions \(MIME\) Part One: Format of Internet Message Bodies](#) and [Multipurpose Internet Mail Extensions \(MIME\) Part Two: Media Types](#).

For most uses you specify **application/octet-stream**.

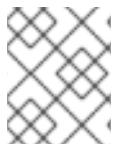
[Example 9.2, "Binary Data for MTOM"](#) shows how you can modify **xRayType** from [Example 9.1, "Message for MTOM"](#) for using MTOM.

Example 9.2. Binary Data for MTOM

```
...
<types>
  <schema targetNamespace="http://mediStor.org/types/"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:xmime="http://www.w3.org/2005/05/xmlmime">
    <complexType name="xRayType">
      <sequence>
        <element name="patientName" type="xsd:string" />
        <element name="patientNumber" type="xsd:int" />
        <element name="imageData" type="xsd:base64Binary"
          xmime:expectedContentTypes="application/octet-stream"/>
      </sequence>
    </complexType>
    <element name="xRay" type="xsd1:xRayType" />
  </schema>
</types>
...

```

The generated JAXB class generated for **xRayType** no longer contains a **byte[]**. Instead the code generator sees the **xmime:expectedContentTypes** attribute and generates a DataHandler for the imageData field.



NOTE

You do not need to change the **binding** element to use MTOM. The runtime makes the appropriate changes when the data is sent.

Java first

If you are doing Java first development you can make your JAXB class MTOM ready by doing the following:

1. Make sure the field holding the binary data is a DataHandler.

2. Add the `@XmlMimeType()` annotation to the field containing the data you want to stream as an MTOM attachment.

[Example 9.3, "JAXB Class for MTOM"](#) shows a JAXB class annotated for using MTOM.

Example 9.3. JAXB Class for MTOM

```

@XmlType
public class XRayType {
    protected String patientName;
    protected int patientNumber;
    @XmlMimeType("application/octet-stream")
    protected DataHandler imageData;
    ...
}

```

9.3. ENABLING MTOM

By default the Apache CXF runtime does not enable MTOM support. It sends all binary data as either part of the normal SOAP message or as an unoptimized attachment. You can activate MTOM support either programmatically or through the use of configuration.

9.3.1. Using JAX-WS APIs

Overview

Both service providers and consumers must have the MTOM optimizations enabled. The JAX-WS APIs offer different mechanisms for each type of endpoint.

Service provider

If you published your service provider using the JAX-WS APIs you enable the runtime's MTOM support as follows:

1. Access the **Endpoint** object for your published service.
The easiest way to access the **Endpoint** object is when you publish the endpoint. For more information see [Chapter 31, Publishing a Service](#).
2. Get the SOAP binding from the **Endpoint** using its `getBinding()` method, as shown in [Example 9.4, "Getting the SOAP Binding from an Endpoint"](#).

Example 9.4. Getting the SOAP Binding from an Endpoint

```

// Endpoint ep is declared previously
SOAPBinding binding = (SOAPBinding)ep.getBinding();

```

You must cast the returned binding object to a **SOAPBinding** object to access the MTOM property.

3. Set the binding's MTOM enabled property to **true** using the binding's `setMTOMEnabled()` method, as shown in [Example 9.5, "Setting a Service Provider's MTOM Enabled Property"](#).

Example 9.5. Setting a Service Provider's MTOM Enabled Property

```
binding.setMTOMEnabled(true);
```

Consumer

To MTOM enable a JAX-WS consumer you must do the following:

1. Cast the consumer's proxy to a **BindingProvider** object.
For information on getting a consumer proxy see [Chapter 25, Developing a Consumer Without a WSDL Contract](#) or [Chapter 28, Developing a Consumer From a WSDL Contract](#).
2. Get the SOAP binding from the **BindingProvider** using its **getBinding()** method, as shown in [Example 9.6, "Getting a SOAP Binding from a BindingProvider"](#).

Example 9.6. Getting a SOAP Binding from aBindingProvider

```
// BindingProvider bp declared previously
SOAPBinding binding = (SOAPBinding)bp.getBinding();
```

3. Set the bindings MTOM enabled property to **true** using the binding's **setMTOMEnabled()** method, as shown in [Example 9.7, "Setting a Consumer's MTOM Enabled Property"](#).

Example 9.7. Setting a Consumer's MTOM Enabled Property

```
binding.setMTOMEnabled(true);
```

9.3.2. Using configuration**Overview**

If you publish your service using XML, such as when deploying to a container, you can enable your endpoint's MTOM support in the endpoint's configuration file. For more information on configuring endpoint's see [Part IV, "Configuring Web Service Endpoints"](#).

Procedure

The MTOM property is set inside the **jaxws:endpoint** element for your endpoint. To enable MTOM do the following:

1. Add a **jaxws:property** child element to the endpoint's **jaxws:endpoint** element.
2. Add a **entry** child element to the **jaxws:property** element.
3. Set the **entry** element's **key** attribute to **mtom-enabled**.
4. Set the **entry** element's **value** attribute to **true**.

Example

[Example 9.8, "Configuration for Enabling MTOM"](#) shows an endpoint that is MTOM enabled.

Example 9.8. Configuration for Enabling MTOM

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                           http://cxf.apache.org/jaxws http://cxf.apache.org/schema/jaxws.xsd">

    <jaxws:endpoint id="xRayStorage"
                    implementor="demo.spring.xRayStorImpl"
                    address="http://localhost/xRayStorage">
        <jaxws:properties>
            <entry key="mtom-enabled" value="true"/>
        </jaxws:properties>
    </jaxws:endpoint>
</beans>
```

CHAPTER 10. USING XML DOCUMENTS

Abstract

The pure XML payload format provides an alternative to the SOAP binding by allowing services to exchange data using straight XML documents without the overhead of a SOAP envelope.

XML BINDING NAMESPACE

The extensions used to describe XML format bindings are defined in the namespace <http://cxf.apache.org/bindings/xformat>. Apache CXF tools use the prefix **xformat** to represent the XML binding extensions. Add the following line to your contracts:

```
xmlns:xformat="http://cxf.apache.org/bindings/xformat"
```

HAND EDITING

To map an interface to a pure XML payload format do the following:

1. Add the namespace declaration to include the extensions defining the XML binding. See [the section called "XML binding namespace"](#).
2. Add a standard WSDL **binding** element to your contract to hold the XML binding, give the binding a unique **name**, and specify the name of the WSDL **portType** element that represents the interface being bound.
3. Add an **xformat:binding** child element to the **binding** element to identify that the messages are being handled as pure XML documents without SOAP envelopes.
4. Optionally, set the **xformat:binding** element's **rootNode** attribute to a valid QName. For more information on the effect of the **rootNode** attribute see [the section called "XML messages on the wire"](#).
5. For each operation defined in the bound interface, add a standard WSDL **operation** element to hold the binding information for the operation's messages.
6. For each operation added to the binding, add the **input**, **output**, and **fault** children elements to represent the messages used by the operation.
These elements correspond to the messages defined in the interface definition of the logical operation.
7. Optionally add an **xformat:body** element with a valid **rootNode** attribute to the added **input**, **output**, and **fault** elements to override the value of **rootNode** set at the binding level.



NOTE

If any of your messages have no parts, for example the output message for an operation that returns void, you must set the **rootNode** attribute for the message to ensure that the message written on the wire is a valid, but empty, XML document.

XML MESSAGES ON THE WIRE

When you specify that an interface's messages are to be passed as XML documents, without a SOAP

envelope, you must take care to ensure that your messages form valid XML documents when they are written on the wire. You also need to ensure that non-Apache CXF participants that receive the XML documents understand the messages generated by Apache CXF.

A simple way to solve both problems is to use the optional **rootNode** attribute on either the global **xformat:binding** element or on the individual message's **xformat:body** elements. The **rootNode** attribute specifies the QName for the element that serves as the root node for the XML document generated by Apache CXF. When the **rootNode** attribute is not set, Apache CXF uses the root element of the message part as the root element when using doc style messages, or an element using the message part name as the root element when using rpc style messages.

For example, if the **rootNode** attribute is not set the message defined in [Example 10.1, "Valid XML Binding Message"](#) would generate an XML document with the root element **lineNumber**.

Example 10.1. Valid XML Binding Message

```
<type ... >
...
<element name="operatorID" type="xsd:int"/>
...
</types>
<message name="operator">
  <part name="lineNumber" element="ns1:operatorID"/>
</message>
```

For messages with one part, Apache CXF will always generate a valid XML document even if the **rootNode** attribute is not set. However, the message in [Example 10.2, "Invalid XML Binding Message"](#) would generate an invalid XML document.

Example 10.2. Invalid XML Binding Message

```
<types>
...
<element name="pairName" type="xsd:string"/>
<element name="entryNum" type="xsd:int"/>
...
</types>

<message name="matildas">
  <part name="dancing" element="ns1:pairName"/>
  <part name="number" element="ns1:entryNum"/>
</message>
```

Without the **rootNode** attribute specified in the XML binding, Apache CXF will generate an XML document similar to [Example 10.3, "Invalid XML Document"](#) for the message defined in [Example 10.2, "Invalid XML Binding Message"](#). The generated XML document is invalid because it has two root elements: **pairName** and **entryNum**.

Example 10.3. Invalid XML Document

```
<pairName>
Fred&Linda
```

```
</pairName>
<entryNum>
  123
</entryNum>
```

If you set the **rootNode** attribute, as shown in [Example 10.4, "XML Binding with rootNode set"](#) Apache CXF will wrap the elements in the specified root element. In this example, the **rootNode** attribute is defined for the entire binding and specifies that the root element will be named entrants.

Example 10.4. XML Binding with rootNode set

```
<portType name="danceParty">
  <operation name="register">
    <input message="tns:matildas" name="contestant"/>
  </operation>
</portType>

<binding name="matildaXMLBinding" type="tns:dancingMatildas">
  <xmlelement:binding rootNode="entrants"/>
  <operation name="register">
    <input name="contestant"/>
    <output name="entered"/>
  </operation>
</binding>
```

An XML document generated from the input message would be similar to [Example 10.5, "XML Document generated using the rootNode attribute"](#). Notice that the XML document now only has one root element.

Example 10.5. XML Document generated using the rootNode attribute

```
<entrants>
  <pairName>
    Fred&Linda
  <entryNum>
    123
  </entryNum>
</entrants>
```

OVERRIDING THE BINDING'S ROOTNODE ATTRIBUTE SETTING

You can also set the **rootNode** attribute for each individual message, or override the global setting for a particular message, by using the **xformat:body** element inside of the message binding. For example, if you wanted the output message defined in [Example 10.4, "XML Binding with rootNode set"](#) to have a different root element from the input message, you could override the binding's root element as shown in [Example 10.6, "Using xformat:body"](#).

Example 10.6. Using xformat:body

```
<binding name="matildaXMLBinding" type="tns:dancingMatildas">
  <xmlelement:binding rootNode="entrants"/>
```

```
<operation name="register">
  <input name="contestant"/>
  <output name="entered">
    <xformat:body rootNode="entryStatus" />
  </output>
</operation>
</binding>
```

PART III. WEB SERVICES TRANSPORTS

This part describes how to add Apache CXF transports to a WSDL document.

CHAPTER 11. UNDERSTANDING HOW ENDPOINTS ARE DEFINED IN WSDL

Abstract

Endpoints represent an instantiated service. They are defined by combining a binding and the networking details used to expose the endpoint.

OVERVIEW

An endpoint can be thought of as a physical manifestation of a service. It combines a binding, which specifies the physical representation of the logical data used by a service, and a set of networking details that define the physical connection details used to make the service contactable by other endpoints.



NOTE

CXF providers are servers for CXF consumers, which correspond to clients. If you are using the CXF (**camel-cxf**) component as the starting endpoint in a route, then the endpoint is both a Camel consumer and a CXF provider. If you are using the Camel CXF component, as an ending endpoint in a route, then the endpoint is both a Camel producer and a CXF consumer.

ENDPOINTS AND SERVICES

In the same way a binding can only map a single interface, an endpoint can only map to a single service. However, a service can be manifested by any number of endpoints. For example, you could define a ticket selling service that was manifested by four different endpoints. However, you could not have a single endpoint that manifested both a ticket selling service and a widget selling service.

THE WSDL ELEMENTS

Endpoints are defined in a contract using a combination of the WSDL **service** element and the WSDL **port** element. The **service** element is a collection of related **port** elements. The **port** elements define the actual endpoints.

The WSDL **service** element has a single attribute, **name**, that specifies a unique name. The **service** element is used as the parent element of a collection of related **port** elements. WSDL makes no specification about how the **port** elements are related. You can associate the **port** elements in any manner you see fit.

The WSDL **port** element has a **binding** attribute, that specifies the binding used by the endpoint and is a reference to the **wsdl:binding** element. It also includes the **name** attribute, which is a mandatory attribute that provides a unique name among all ports. The **port** element is the parent element of the elements that specify the actual transport details used by the endpoint. The elements used to specify the transport details are discussed in the following sections.

ADDING ENDPOINTS TO A CONTRACT

Apache CXF provides command line tools that can generate endpoints for predefined service interface and binding combinations.

The tools will add the proper elements to your contract for you. However, it is recommended that you have some knowledge of how the different transports used in defining an endpoint work.

You can also add an endpoint to a contract using any text editor. When you hand edit a contract, you are responsible for ensuring that the contract is valid.

SUPPORTED TRANSPORTS

Endpoint definitions are built using extensions defined for each of the transports Apache CXF supports. This includes the following transports:

- HTTP
- CORBA
- Java Messaging Service

CHAPTER 12. USING HTTP

Abstract

HTTP is the underlying transport for the Web. It provides a standardized, robust, and flexible platform for communicating between endpoints. Because of these factors it is the assumed transport for most WS-* specifications and is integral to RESTful architectures.

12.1. ADDING A BASIC HTTP ENDPOINT

Alternative HTTP runtimes

Apache CXF supports the following alternative HTTP runtime implementations:

- [Undertow](#), which is described in detail in [Section 12.4, “Configuring the Undertow Runtime”](#).
- [Netty](#), which is described in detail in [Section 12.5, “Configuring the Netty Runtime”](#).

Netty HTTP URL

Normally, a HTTP endpoint uses whichever HTTP runtime is included on the classpath (either Undertow or Netty). If both the Undertow runtime and Netty runtime are included on the classpath, however, you need to specify explicitly when you want to use the Netty runtime, because the Undertow runtime will be used by default.

In the case where more than one HTTP runtime is available on the classpath, you can select the Undertow runtime by specifying the endpoint URL to have the following format:

`netty://http://RestOfURL`

Payload types

There are three ways of specifying an HTTP endpoint’s address depending on the payload format you are using.

- SOAP 1.1 uses the standardized **soap:address** element.
- SOAP 1.2 uses the **soap12:address** element.
- All other payload formats use the **http:address element**.



NOTE

From Camel 2.16.0 release, Apache Camel CXF Payload supports stream cache out of box.

SOAP 1.1

When you are sending SOAP 1.1 messages over HTTP you must use the SOAP 1.1 **address** element to specify the endpoint’s address. It has one attribute, **location**, that specifies the endpoint’s address as a URL. The SOAP 1.1 **address** element is defined in the namespace

<http://schemas.xmlsoap.org/wsdl/soap/>.

[Example 12.1, “SOAP 1.1 Port Element”](#) shows a **port** element used to send SOAP 1.1 messages over HTTP.

Example 12.1. SOAP 1.1 Port Element

```
<definitions ...>
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" ...>
    ...
    <service name="SOAP11Service">
        <port binding="SOAP11Binding" name="SOAP11Port">
            <soap:address location="http://artie.com/index.xml">
        </port>
    </service>
    ...
</definitions>
```

SOAP 1.2

When you are sending SOAP 1.2 messages over HTTP you must use the SOAP 1.2 **address** element to specify the endpoint’s address. It has one attribute, **location**, that specifies the endpoint’s address as a URL. The SOAP 1.2 **address** element is defined in the namespace <http://schemas.xmlsoap.org/wsdl/soap12/>.

[Example 12.2, “SOAP 1.2 Port Element”](#) shows a **port** element used to send SOAP 1.2 messages over HTTP.

Example 12.2. SOAP 1.2 Port Element

```
<definitions ...>
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" ...>
    <service name="SOAP12Service">
        <port binding="SOAP12Binding" name="SOAP12Port">
            <soap12:address location="http://artie.com/index.xml">
        </port>
    </service>
    ...
</definitions>
```

Other messages types

When your messages are mapped to any payload format other than SOAP you must use the HTTP **address** element to specify the endpoint’s address. It has one attribute, **location**, that specifies the endpoint’s address as a URL. The HTTP **address** element is defined in the namespace <http://schemas.xmlsoap.org/wsdl/http/>.

[Example 12.3, “HTTP Port Element”](#) shows a **port** element used to send an XML message.

Example 12.3. HTTP Port Element

```
<definitions ...>
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" ...>
```

```

<service name="HTTPService">
  <port binding="HTTPBinding" name="HTTPPort">
    <http:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
</definitions>

```

12.2. CONFIGURING A CONSUMER

12.2.1. Mechanisms for HTTP Consumer Endpoints

HTTP consumer endpoints can specify a number of HTTP connection attributes including whether the endpoint automatically accepts redirect responses, whether the endpoint can use chunking, whether the endpoint will request a keep-alive, and how the endpoint interacts with proxies. In addition to the HTTP connection properties, an HTTP consumer endpoint can specify how it is secured.

A consumer endpoint can be configured using two mechanisms:

- [Configuration](#)
- [WSDL](#)

12.2.2. Using Configuration

Namespace

The elements used to configure an HTTP consumer endpoint are defined in the namespace <http://cxf.apache.org/transports/http/configuration>. It is commonly referred to using the prefix **http-conf**. In order to use the HTTP configuration elements you must add the lines shown in [Example 12.4](#), “[HTTP Consumer Configuration Namespace](#)” to the **beans** element of your endpoint’s configuration file. In addition, you must add the configuration elements’ namespace to the **xsi:schemaLocation** attribute.

Example 12.4. HTTP Consumer Configuration Namespace

```

<beans ...
  xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
  ...
  xsi:schemaLocation="...
    http://cxf.apache.org/transports/http/configuration
    http://cxf.apache.org/schemas/configuration/http-conf.xsd
  ...">

```

Undertow runtime or Netty runtime

You can use the elements from the **http-conf** namespace to configure either the Undertow runtime or the Netty runtime.

The conduit element

You configure an HTTP consumer endpoint using the **http-conf:conduit** element and its children. The **http-conf:conduit** element takes a single attribute, **name**, that specifies the WSDL **port** element corresponding to the endpoint. The value for the **name** attribute takes the form *portQName` .http-conduit`*. [Example 12.5, “http-conf:conduit Element”](#) shows the **http-conf:conduit** element that would be used to add configuration for an endpoint that is specified by the WSDL fragment **<port binding="widgetSOAPBinding" name="widgetSOAPPort">** when the endpoint’s target namespace is <http://widgets.widgetvendor.net>.

Example 12.5. http-conf:conduit Element

```
...
<http-conf:conduit name="{http://widgets/widgetvendor.net}widgetSOAPPort.http-conduit">
  ...
</http-conf:conduit>
...
```

The **http-conf:conduit** element has child elements that specify configuration information. They are described in [Table 12.1, “Elements Used to Configure an HTTP Consumer Endpoint”](#).

Table 12.1. Elements Used to Configure an HTTP Consumer Endpoint

Element	Description
http-conf:client	Specifies the HTTP connection properties such as timeouts, keep-alive requests, content types, etc. See the section called “The client element” .
http-conf:authorization	Specifies the parameters for configuring the basic authentication method that the endpoint uses preemptively. The preferred approach is to supply a http-conf:basicAuthSupplier object.
http-conf:proxyAuthorization	Specifies the parameters for configuring basic authentication against outgoing HTTP proxy servers.
http-conf:tlsClientParameters	Specifies the parameters used to configure SSL/TLS.
http-conf:basicAuthSupplier	Specifies the bean reference or class name of the object that supplies the basic authentication information used by the endpoint, either preemptively or in response to a 401 HTTP challenge.
http-conf:trustDecider	Specifies the bean reference or class name of the object that checks the HTTP(S) URLConnection object to establish trust for a connection with an HTTPS service provider before any information is transmitted.

The client element

The **http-conf:client** element is used to configure the non-security properties of a consumer endpoint's HTTP connection. Its attributes, described in [Table 12.2, "HTTP Consumer Configuration Attributes"](#), specify the connection's properties.

Table 12.2. HTTP Consumer Configuration Attributes

Attribute	Description
ConnectionTimeout	<p>Specifies the amount of time, in milliseconds, that the consumer attempts to establish a connection before it times out. The default is 30000.</p> <p>0 specifies that the consumer will continue to send the request indefinitely.</p>
ReceiveTimeout	<p>Specifies the amount of time, in milliseconds, that the consumer will wait for a response before it times out. The default is 30000.</p> <p>0 specifies that the consumer will wait indefinitely.</p>
AutoRedirect	<p>Specifies if the consumer will automatically follow a server issued redirection. The default is false.</p>
MaxRetransmits	<p>Specifies the maximum number of times a consumer will retransmit a request to satisfy a redirect. The default is -1 which specifies that unlimited retransmissions are allowed.</p>
AllowChunking	<p>Specifies whether the consumer will send requests using chunking. The default is true which specifies that the consumer will use chunking when sending requests.</p> <p>Chunking cannot be used if either of the following are true:</p> <ul style="list-style-type: none"> • http-conf:basicAuthSupplier is configured to provide credentials preemptively. • AutoRedirect is set to true. <p>In both cases the value of AllowChunking is ignored and chunking is disallowed.</p>
Accept	<p>Specifies what media types the consumer is prepared to handle. The value is used as the value of the HTTP Accept property. The value of the attribute is specified using multipurpose internet mail extensions (MIME) types.</p>

Attribute	Description
AcceptLanguage	<p>Specifies what language (for example, American English) the consumer prefers for the purpose of receiving a response. The value is used as the value of the HTTP AcceptLanguage property.</p> <p>Language tags are regulated by the International Organization for Standards (ISO) and are typically formed by combining a language code, determined by the ISO-639 standard, and country code, determined by the ISO-3166 standard, separated by a hyphen. For example, en-US represents American English.</p>
AcceptEncoding	<p>Specifies what content encodings the consumer is prepared to handle. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). The value is used as the value of the HTTP AcceptEncoding property.</p>
ContentType	<p>Specifies the media type of the data being sent in the body of a message. Media types are specified using multipurpose internet mail extensions (MIME) types. The value is used as the value of the HTTP ContentType property. The default is text/xml.</p> <p>For web services, this should be set to text/xml. If the client is sending HTML form data to a CGI script, this should be set to application/x-www-form-urlencoded. If the HTTP POST request is bound to a fixed payload format (as opposed to SOAP), the content type is typically set to application/octet-stream.</p>
Host	<p>Specifies the Internet host and port number of the resource on which the request is being invoked. The value is used as the value of the HTTP Host property.</p> <p>This attribute is typically not required. It is only required by certain DNS scenarios or application designs. For example, it indicates what host the client prefers for clusters (that is, for virtual servers mapping to the same Internet protocol (IP) address).</p>

Attribute	Description
Connection	<p>Specifies whether a particular connection is to be kept open or closed after each request/response dialog. There are two valid values:</p> <ul style="list-style-type: none"> • Keep-Alive – Specifies that the consumer wants the connection kept open after the initial request/response sequence. If the server honors it, the connection is kept open until the consumer closes it. • close(default) – Specifies that the connection to the server is closed after each request/response sequence.
CacheControl	<p>Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a request from a consumer to a service provider. See Section 12.2.4, "Consumer Cache Control Directives".</p>
Cookie	<p>Specifies a static cookie to be sent with all requests.</p>
BrowserType	<p>Specifies information about the browser from which the request originates. In the HTTP specification from the World Wide Web consortium (W3C) this is also known as the user-agent. Some servers optimize based on the client that is sending the request.</p>
Referer	<p>Specifies the URL of the resource that directed the consumer to make requests on a particular service. The value is used as the value of the HTTP Referer property.</p> <p>This HTTP property is used when a request is the result of a browser user clicking on a hyperlink rather than typing a URL. This can allow the server to optimize processing based upon previous task flow, and to generate lists of back-links to resources for the purposes of logging, optimized caching, tracing of obsolete or mistyped links, and so on. However, it is typically not used in web services applications.</p> <p>If the AutoRedirect attribute is set to true and the request is redirected, any value specified in the Referer attribute is overridden. The value of the HTTP Referer property is set to the URL of the service that redirected the consumer's original request.</p>

Attribute	Description
DecoupledEndpoint	<p>Specifies the URL of a decoupled endpoint for the receipt of responses over a separate provider→consumer connection. For more information on using decoupled endpoints see, Section 12.6, “Using the HTTP Transport in Decoupled Mode”.</p> <p>You must configure both the consumer endpoint and the service provider endpoint to use WS-Addressing for the decoupled endpoint to work.</p>
ProxyServer	Specifies the URL of the proxy server through which requests are routed.
ProxyServerPort	Specifies the port number of the proxy server through which requests are routed.
ProxyServerType	<p>Specifies the type of proxy server used to route requests. Valid values are:</p> <ul style="list-style-type: none"> • HTTP(default) • SOCKS

Example

[Example 12.6, “HTTP Consumer Endpoint Configuration”](#) shows the configuration of an HTTP consumer endpoint that wants to keep its connection to the provider open between requests, that will only retransmit requests once per invocation, and that cannot use chunking streams.

Example 12.6. HTTP Consumer Endpoint Configuration

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http-conf="http://cx.f.apache.org/transports/http/configuration"
       xsi:schemaLocation="http://cx.f.apache.org/transports/http/configuration
                           http://cx.f.apache.org/schemas/configuration/http-conf.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <http-conf:conduit name="{http://apache.org/hello_world_soap_http}SoapPort.http-conduit">
        <http-conf:client Connection="Keep-Alive"
                           MaxRetransmits="1"
                           AllowChunking="false" />
    </http-conf:conduit>
</beans>
```

More information

For more information on HTTP conduits see [Chapter 16, Conduits](#).

12.2.3. Using WSDL

Namespace

The WSDL extension elements used to configure an HTTP consumer endpoint are defined in the namespace <http://cxf.apache.org/transports/http/configuration>. It is commonly referred to using the prefix **http-conf**. In order to use the HTTP configuration elements you must add the line shown in [Example 12.7, “HTTP Consumer WSDL Element’s Namespace”](#) to the **definitions** element of your endpoint’s WSDL document.

Example 12.7. HTTP Consumer WSDL Element’s Namespace

```
<definitions ...>
    xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
```

Undertow runtime or Netty runtime

You can use the elements from the **http-conf** namespace to configure either the Undertow runtime or the Netty runtime.

The client element

The **http-conf:client** element is used to specify the connection properties of an HTTP consumer in a WSDL document. The **http-conf:client** element is a child of the WSDL **port** element. It has the same attributes as the **client** element used in the configuration file. The attributes are described in [Table 12.2, “HTTP Consumer Configuration Attributes”](#).

Example

[Example 12.8, “WSDL to Configure an HTTP Consumer Endpoint”](#) shows a WSDL fragment that configures an HTTP consumer endpoint to specify that it does not interact with caches.

Example 12.8. WSDL to Configure an HTTP Consumer Endpoint

```
<service ...>
    <port ...>
        <soap:address ... />
        <http-conf:client CacheControl="no-cache" />
    </port>
</service>
```

12.2.4. Consumer Cache Control Directives

[Table 12.3, “http-conf:client Cache Control Directives”](#) lists the cache control directives supported by an HTTP consumer.

Table 12.3. http-conf:client Cache Control Directives

Directive	Behavior
no-cache	Caches cannot use a particular response to satisfy subsequent requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
no-store	Caches must not store either any part of a response or any part of the request that invoked it.
max-age	The consumer can accept a response whose age is no greater than the specified time in seconds.
max-stale	The consumer can accept a response that has exceeded its expiration time. If a value is assigned to max-stale, it represents the number of seconds beyond the expiration time of a response up to which the consumer can still accept that response. If no value is assigned, the consumer can accept a stale response of any age.
min-fresh	The consumer wants a response that is still fresh for at least the specified number of seconds indicated.
no-transform	Caches must not modify media type or location of the content in a response between a provider and a consumer.
only-if-cached	Caches should return only responses that are currently stored in the cache, and not responses that need to be reloaded or revalidated.
cache-extension	Specifies additional extensions to the other cache directives. Extensions can be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can adhere to the behavior mandated by the standard directive.

12.3. CONFIGURING A SERVICE PROVIDER

12.3.1. Mechanisms for a HTTP Service Provider

HTTP service provider endpoints can specify a number of HTTP connection attributes including if it will honor keep alive requests, how it interacts with caches, and how tolerant it is of errors in communicating with a consumer.

A service provider endpoint can be configured using two mechanisms:

- Configuration
- WSDL

12.3.2. Using Configuration

Namespace

The elements used to configure an HTTP provider endpoint are defined in the namespace <http://cxf.apache.org/transports/http/configuration>. It is commonly referred to using the prefix **http-conf**. In order to use the HTTP configuration elements you must add the lines shown in [Example 12.9, “HTTP Provider Configuration Namespace”](#) to the **beans** element of your endpoint's configuration file. In addition, you must add the configuration elements' namespace to the **xsi:schemaLocation** attribute.

Example 12.9. HTTP Provider Configuration Namespace

```
<beans ...
    xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
    ...
    xsi:schemaLocation="...
        http://cxf.apache.org/transports/http/configuration
        http://cxf.apache.org/schemas/configuration/http-conf.xsd
    ...">
```

Undertow runtime or Netty runtime

You can use the elements from the **http-conf** namespace to configure either the Undertow runtime or the Netty runtime.

The destination element

You configure an HTTP service provider endpoint using the **http-conf:destination** element and its children. The **http-conf:destination** element takes a single attribute, **name**, that specifies the WSDL **port** element that corresponds to the endpoint. The value for the **name** attribute takes the form *portQName` .http-destination`*. [Example 12.10, “http-conf:destination Element”](#) shows the **http-conf:destination** element that is used to add configuration for an endpoint that is specified by the WSDL fragment **<port binding="widgetSOAPBinding" name="widgetSOAPPort>** when the endpoint's target namespace is <http://widgets.widgetvendor.net>.

Example 12.10. http-conf:destination Element

```
...
<http-conf:destination name="{http://widgets/widgetvendor.net}widgetSOAPPort.http-
destination">
...
</http-conf:destination>
...
```

The **http-conf:destination** element has a number of child elements that specify configuration information. They are described in [Table 12.4, “Elements Used to Configure an HTTP Service Provider Endpoint”](#).

Table 12.4. Elements Used to Configure an HTTP Service Provider Endpoint

Element	Description
http-conf:server	Specifies the HTTP connection properties. See the section called "The server element" .
http-conf:contextMatchStrategy	Specifies the parameters that configure the context match strategy for processing HTTP requests.
http-conf:fixedParameterOrder	Specifies whether the parameter order of an HTTP request handled by this destination is fixed.

The server element

The **http-conf:server** element is used to configure the properties of a service provider endpoint's HTTP connection. Its attributes, described in [Table 12.5, "HTTP Service Provider Configuration Attributes"](#), specify the connection's properties.

Table 12.5. HTTP Service Provider Configuration Attributes

Attribute	Description
ReceiveTimeout	Sets the length of time, in milliseconds, the service provider attempts to receive a request before the connection times out. The default is 30000 . 0 specifies that the provider will not timeout.
SuppressClientSendErrors	Specifies whether exceptions are to be thrown when an error is encountered on receiving a request. The default is false ; exceptions are thrown on encountering errors.
SuppressClientReceiveErrors	Specifies whether exceptions are to be thrown when an error is encountered on sending a response to a consumer. The default is false ; exceptions are thrown on encountering errors.
HonorKeepAlive	Specifies whether the service provider honors requests for a connection to remain open after a response has been sent. The default is false ; keep-alive requests are ignored.

Attribute	Description
RedirectURL	Specifies the URL to which the client request should be redirected if the URL specified in the client request is no longer appropriate for the requested resource. In this case, if a status code is not automatically set in the first line of the server response, the status code is set to 302 and the status description is set to Object Moved. The value is used as the value of the HTTP RedirectURL property.
CacheControl	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a response from a service provider to a consumer. See Section 12.3.4, “Service Provider Cache Control Directives” .
ContentLocation	Sets the URL where the resource being sent in a response is located.
ContentType	Specifies the media type of the information being sent in a response. Media types are specified using multipurpose internet mail extensions (MIME) types. The value is used as the value of the HTTP ContentType location.
ContentEncoding	<p>Specifies any additional content encodings that have been applied to the information being sent by the service provider. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). Possible content encoding values include zip, gzip, compress, deflate, and identity. This value is used as the value of the HTTP ContentEncoding property.</p> <p>The primary use of content encodings is to allow documents to be compressed using some encoding mechanism, such as zip or gzip. Apache CXF performs no validation on content codings. It is the user's responsibility to ensure that a specified content coding is supported at application level.</p>
ServerType	Specifies what type of server is sending the response. Values take the form <i>program-name/version</i> ; for example, Apache/1.2.5 .

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- A simple response -->
<response>
```

[Example 12.11, “HTTP Service Provider Endpoint Configuration”](#) shows the configuration for an HTTP service provider endpoint that honors keep-alive requests and suppresses all communication errors.

Example 12.11. HTTP Service Provider Endpoint Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http-conf="http://cx.f.apache.org/transport/http/configuration"
       xsi:schemaLocation="http://cx.f.apache.org/transport/http/configuration
                           http://cx.f.apache.org/schemas/configuration/http-conf.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <http-conf:destination name="{http://apache.org/hello_world_soap_http}SoapPort.http-
destination">
        <http-conf:server SuppressClientSendErrors="true"
                          SuppressClientReceiveErrors="true"
                          HonorKeepAlive="true" />
    </http-conf:destination>
</beans>
```

12.3.3. Using WSDL

Namespace

The WSDL extension elements used to configure an HTTP provider endpoint are defined in the namespace [http://cx.f.apache.org/transport/http/configuration](#). It is commonly referred to using the prefix **http-conf**. To use the HTTP configuration elements you must add the line shown in [Example 12.12, “HTTP Provider WSDL Element’s Namespace”](#) to the **definitions** element of your endpoint’s WSDL document.

Example 12.12. HTTP Provider WSDL Element’s Namespace

```
<definitions ...
    xmlns:http-conf="http://cx.f.apache.org/transport/http/configuration"
```

Undertow runtime or Netty runtime

You can use the elements from the **http-conf** namespace to configure either the Undertow runtime or the Netty runtime.

The server element

The **http-conf:server** element is used to specify the connection properties of an HTTP service provider in a WSDL document. The **http-conf:server** element is a child of the WSDL **port** element. It has the same attributes as the **server** element used in the configuration file. The attributes are described in [Table 12.5, “HTTP Service Provider Configuration Attributes”](#).

Example

[Example 12.13, “WSDL to Configure an HTTP Service Provider Endpoint”](#) shows a WSDL fragment that configures an HTTP service provider endpoint specifying that it will not interact with caches.

Example 12.13. WSDL to Configure an HTTP Service Provider Endpoint

```
<service ... >
<port ... >
  <soap:address ... />
  <http-conf:server CacheControl="no-cache" />
</port>
</service>
```

12.3.4. Service Provider Cache Control Directives

[Table 12.6, “**http-conf:server** Cache Control Directives”](#) lists the cache control directives supported by an HTTP service provider.

Table 12.6. **http-conf:server** Cache Control Directives

Directive	Behavior
no-cache	Caches cannot use a particular response to satisfy subsequent requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
public	Any cache can store the response.
private	Public (shared) caches cannot store the response because the response is intended for a single user. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
no-store	Caches must not store any part of the response or any part of the request that invoked it.
no-transform	Caches must not modify the media type or location of the content in a response between a server and a client.
must-revalidate	Caches must revalidate expired entries that relate to a response before that entry can be used in a subsequent response.

Directive	Behavior
proxy-revalidate	Does the same as must-revalidate, except that it can only be enforced on shared caches and is ignored by private unshared caches. When using this directive, the public cache directive must also be used.
max-age	Clients can accept a response whose age is no greater than the specified number of seconds.
s-max-age	Does the same as max-age, except that it can only be enforced on shared caches and is ignored by private unshared caches. The age specified by s-max-age overrides the age specified by max-age. When using this directive, the proxy-revalidate directive must also be used.
cache-extension	Specifies additional extensions to the other cache directives. Extensions can be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can adhere to the behavior mandated by the standard directive.

12.4. CONFIGURING THE UNDERTOW RUNTIME

Overview

The Undertow runtime is used by HTTP service providers and HTTP consumers using a decoupled endpoint. The runtime's thread pool can be configured, and you can also set a number of the security settings for an HTTP service provider through the Undertow runtime.

Maven dependency

If you use Apache Maven as your build system, you can add the Undertow runtime to your project by including the following dependency in your project's **pom.xml** file:

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxfrt-transports-http-undertow</artifactId>
  <version>${cxfrt-version}</version>
</dependency>
```

Namespace

The elements used to configure the Undertow runtime are defined in the namespace <http://cxf.apache.org/transports/http-undertow/configuration>. In order to use the Undertow configuration elements you must add the lines shown in Example 12.14, "Undertow Runtime

[Configuration Namespace](#)” to the **beans** element of your endpoint’s configuration file. In this example, the namespace is assigned the prefix **httpu**. In addition, you must add the configuration element’s namespace to the **xsi:schemaLocation** attribute.

Example 12.14. Undertow Runtime Configuration Namespace

```
<beans ...  
    xmlns:httpu="http://cxf.apache.org/transports/http-undertow/configuration"  
    ...  
    xsi:schemaLocation="...  
        http://cxf.apache.org/transports/http-undertow/configuration  
        http://cxf.apache.org/schemas/configuration/http-undertow.xsd  
    ...">
```

The engine-factory element

The **httpu:engine-factory** element is the root element used to configure the Undertow runtime used by an application. It has a single required attribute, **bus**, whose value is the name of the **Bus** that manages the Undertow instances being configured.



NOTE

The value is typically **cxfr** which is the name of the default **Bus** instance.

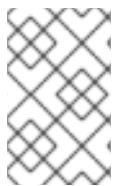
The **httpu:engine-factory** element has three children that contain the information used to configure the HTTP ports instantiated by the Undertow runtime factory. The children are described in [Table 12.7, “Elements for Configuring a Undertow Runtime Factory”](#).

Table 12.7. Elements for Configuring a Undertow Runtime Factory

Element	Description
httpu:engine	Specifies the configuration for a particular Undertow runtime instance. See the section called “The engine element” .
httpu:identifiedTLSServerParameters	Specifies a reusable set of properties for securing an HTTP service provider. It has a single attribute, id , that specifies a unique identifier by which the property set can be referred.
httpu:identifiedThreadingParameters	Specifies a reusable set of properties for controlling a Undertow instance’s thread pool. It has a single attribute, id , that specifies a unique identifier by which the property set can be referred. See the section called “Configuring the thread pool”

The engine element

The **httpu:engine** element is used to configure specific instances of the Undertow runtime. It has a single attribute, **port**, that specifies the number of the port being managed by the Undertow instance.



NOTE

You can specify a value of **0** for the **port** attribute. Any threading properties specified in an **httpu:engine** element with its **port** attribute set to **0** are used as the configuration for all Undertow listeners that are not explicitly configured.

Each **httpu:engine** element can have two children: one for configuring security properties and one for configuring the Undertow instance's thread pool. For each type of configuration you can either directly provide the configuration information or you can provide a reference to a set of configuration properties defined in the parent **httpu:engine-factory** element.

The child elements used to provide the configuration properties are described in [Table 12.8, "Elements for Configuring an Undertow Runtime Instance"](#).

Table 12.8. Elements for Configuring an Undertow Runtime Instance

Element	Description
httpu:tlsServerParameters	Specifies a set of properties for configuring the security used for the specific Undertow instance.
httpu:tlsServerParametersRef	Refers to a set of security properties defined by a identifiedTLSParameters element. The id attribute provides the id of the referred identifiedTLSParameters element.
httpu:threadingParameters	Specifies the size of the thread pool used by the specific Undertow instance. See the section called "Configuring the thread pool" .
httpu:threadingParametersRef	Refers to a set of properties defined by a identifiedThreadingParameters element. The id attribute provides the id of the referred identifiedThreadingParameters element.

Configuring the thread pool

You can configure the size of an Undertow instance's thread pool by either:

- Specifying the size of the thread pool using a **identifiedThreadingParameters** element in the **engine-factory** element. You then refer to the element using a **threadingParametersRef** element.
- Specifying the size of the of the thread pool directly using a **threadingParameters** element.

The **threadingParameters** has two attributes to specify the size of a thread pool. The attributes are described in [Table 12.9, "Attributes for Configuring an Undertow Thread Pool"](#).



NOTE

The **httpu:identifiedThreadingParameters** element has a single child **threadingParameters** element.

Table 12.9. Attributes for Configuring an Undertow Thread Pool

Attribute	Description
minThreads	Specifies the minimum number of threads available to the Undertow instance for processing requests.
maxThreads	Specifies the maximum number of threads available to the Undertow instance for processing requests.

Example

[Example 12.15, “Configuring an Undertow Instance”](#) shows a configuration fragment that configures an Undertow instance on port number 9001.

Example 12.15. Configuring an Undertow Instance

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:sec="http://cxf.apache.org/configuration/security"
    xmlns:http="http://cxf.apache.org/transports/http/configuration"
    xmlns:https="http://cxf.apache.org/transports/http-undertow/configuration"
    xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
    xsi:schemaLocation="http://cxf.apache.org/configuration/security
        http://cxf.apache.org/schemas/configuration/security.xsd
        http://cxf.apache.org/transports/http/configuration
        http://cxf.apache.org/schemas/configuration/http-conf.xsd
        http://cxf.apache.org/transports/http-undertow/configuration
        http://cxf.apache.org/schemas/configuration/http-undertow.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
    ...
<httpu:engine-factory bus="cxf">
    <httpu:identifiedTLSServerParameters id="secure">
        <sec:keyManagers keyPassword="password">
            <sec:keyStore type="JKS" password="password">
                file="certs/cherry.jks"/>
            </sec:keyManagers>
        </httpu:identifiedTLSServerParameters>

        <httpu:engine port="9001">
            <httpu:tlsServerParametersRef id="secure" />
            <httpu:threadingParameters minThreads="5"
                maxThreads="15" />
        </httpu:engine>
    </httpu:engine-factory>
</beans>
```

12.5. CONFIGURING THE NETTY RUNTIME

Overview

The Netty runtime is used by HTTP service providers and HTTP consumers using a decoupled endpoint. The runtime's thread pool can be configured, and you can also set a number of the security settings for an HTTP service provider through the Netty runtime.

Maven dependencies

If you use Apache Maven as your build system, you can add the server-side implementation of the Netty runtime (for defining Web service endpoints) to your project by including the following dependency in your project's **pom.xml** file:

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxfrt-transports-http-netty-server</artifactId>
  <version>${cxf-version}</version>
</dependency>
```

You can add the client-side implementation of the Netty runtime (for defining Web service clients) to your project by including the following dependency in your project's **pom.xml** file:

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxfrt-transports-http-netty-client</artifactId>
  <version>${cxf-version}</version>
</dependency>
```

Namespace

The elements used to configure the Netty runtime are defined in the namespace <http://cxf.apache.org/transports/http-netty-server/configuration>. It is commonly referred to using the prefix **httpn**. In order to use the Netty configuration elements you must add the lines shown in [Example 12.16, "Netty Runtime Configuration Namespace"](#) to the **beans** element of your endpoint's configuration file. In addition, you must add the configuration elements' namespace to the **xsi:schemaLocation** attribute.

Example 12.16. Netty Runtime Configuration Namespace

```
<beans ...
  xmlns:httpn="http://cxf.apache.org/transports/http-netty-server/configuration"
  ...
  xsi:schemaLocation="...
    http://cxf.apache.org/transports/http-netty-server/configuration
    http://cxf.apache.org/schemas/configuration/http-netty-server.xsd
  ...">
```

The engine-factory element

The **httpn:engine-factory** element is the root element used to configure the Netty runtime used by an application. It has a single required attribute, **bus**, whose value is the name of the **Bus** that manages the Netty instances being configured.



NOTE

The value is typically **cxf**, which is the name of the default **Bus** instance.

The **httpn:engine-factory** element has three children that contain the information used to configure the HTTP ports instantiated by the Netty runtime factory. The children are described in [Table 12.10, "Elements for Configuring a Netty Runtime Factory"](#).

Table 12.10. Elements for Configuring a Netty Runtime Factory

Element	Description
httpn:engine	Specifies the configuration for a particular Netty runtime instance. See the section called "The engine element" .
httpn:identifiedTLSServerParameters	Specifies a reusable set of properties for securing an HTTP service provider. It has a single attribute, id , that specifies a unique identifier by which the property set can be referred.
httpn:identifiedThreadingParameters	Specifies a reusable set of properties for controlling a Netty instance's thread pool. It has a single attribute, id , that specifies a unique identifier by which the property set can be referred. See the section called "Configuring the thread pool" .

The engine element

The **httpn:engine** element is used to configure specific instances of the Netty runtime. [Table 12.11, "Attributes for Configuring a Netty Runtime Instance"](#) shows the attributes supported by the **httpn:engine** element.

Table 12.11. Attributes for Configuring a Netty Runtime Instance

Attribute	Description
port	Specifies the port used by the Netty HTTP server instance. You can specify a value of 0 for the port attribute. Any threading properties specified in an engine element with its port attribute set to 0 are used as the configuration for all Netty listeners that are not explicitly configured.

Attribute	Description
host	Specifies the listen address used by the Netty HTTP server instance. The value can be a hostname or an IP address. If not specified, Netty HTTP server will listen on all local addresses.
readIdleTime	Specifies the maximum read idle time for a Netty connection. The timer is reset whenever there are any read actions on the underlying stream.
writetime	Specifies the maximum write idle time for a Netty connection. The timer is reset whenever there are any write actions on the underlying stream.
maxChunkContentSize	Specifies the maximum aggregated content size for a Netty connection. The default value is 10MB.

A **httpn:engine** element has one child element for configuring security properties and one child element for configuring the Netty instance's thread pool. For each type of configuration you can either directly provide the configuration information or you can provide a reference to a set of configuration properties defined in the parent **httpn:engine-factory** element.

The supported child elements of **httpn:engine** are shown in [Table 12.12, “Elements for Configuring a Netty Runtime Instance”](#).

Table 12.12. Elements for Configuring a Netty Runtime Instance

Element	Description
httpn:tlsServerParameters	Specifies a set of properties for configuring the security used for the specific Netty instance.
httpn:tlsServerParametersRef	Refers to a set of security properties defined by a identifiedTLSParameters element. The id attribute provides the id of the referred identifiedTLSParameters element.
httpn:threadingParameters	Specifies the size of the thread pool used by the specific Netty instance. See the section called “Configuring the thread pool” .
httpn:threadingParametersRef	Refers to a set of properties defined by a identifiedThreadingParameters element. The id attribute provides the id of the referred identifiedThreadingParameters element.
httpn:sessionSupport	When true , enables support for HTTP sessions. Default is false .

Element	Description
httpn:reuseAddress	Specifies a boolean value to set the ReuseAddress TCP socket option. Default is false .

Configuring the thread pool

You can configure the size of a Netty instance's thread pool by either:

- Specifying the size of the thread pool using a **identifiedThreadingParameters** element in the **engine-factory** element. You then refer to the element using a **threadingParametersRef** element.
- Specifying the size of the of the thread pool directly using a **threadingParameters** element.

The **threadingParameters** element has one attribute to specify the size of a thread pool, as described in [Table 12.13, "Attributes for Configuring a Netty Thread Pool"](#) .



NOTE

The **httpn:identifiedThreadingParameters** element has a single child **threadingParameters** element.

Table 12.13. Attributes for Configuring a Netty Thread Pool

Attribute	Description
threadPoolSize	Specifies the number of threads available to the Netty instance for processing requests.

Example

[Example 12.17, "Configuring a Netty Instance"](#) shows a configuration fragment that configures a variety of Netty ports.

Example 12.17. Configuring a Netty Instance

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:beans="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:h="http://cxf.apache.org/transports/http/configuration"
       xmlns:httpn="http://cxf.apache.org/transports/http-netty-server/configuration"
       xmlns:sec="http://cxf.apache.org/configuration/security"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://cxf.apache.org/configuration/security
           http://cxf.apache.org/schemas/configuration/security.xsd
           http://cxf.apache.org/transports/http/configuration"
```

```

http://cxf.apache.org/schemas/configuration/http-conf.xsd
http://cxf.apache.org/transports/http-netty-server/configuration
http://cxf.apache.org/schemas/configuration/http-netty-server.xsd"
>
...
<httpn:engine-factory bus="cxf">
  <httpn:identifiedTLSServerParameters id="sample1">
    <httpn:tlsServerParameters jsseProvider="SUN" secureSocketProtocol="TLS">
      <sec:clientAuthentication want="false" required="false"/>
    </httpn:tlsServerParameters>
  </httpn:identifiedTLSServerParameters>

  <httpn:identifiedThreadingParameters id="sampleThreading1">
    <httpn:threadingParameters threadPoolSize="120"/>
  </httpn:identifiedThreadingParameters>

  <httpn:engine port="9000" readIdleTime="30000" writeIdleTime="90000">
    <httpn:threadingParametersRef id="sampleThreading1"/>
  </httpn:engine>

  <httpn:engine port="0">
    <httpn:threadingParameters threadPoolSize="400"/>
  </httpn:engine>

  <httpn:engine port="9001" readIdleTime="40000" maxChunkContentSize="10000">
    <httpn:threadingParameters threadPoolSize="99" />
    <httpn:sessionSupport>true</httpn:sessionSupport>
  </httpn:engine>

  <httpn:engine port="9002">
    <httpn:tlsServerParameters>
      <sec:clientAuthentication want="true" required="true"/>
    </httpn:tlsServerParameters>
  </httpn:engine>

  <httpn:engine port="9003">
    <httpn:tlsServerParametersRef id="sample1"/>
  </httpn:engine>

</httpn:engine-factory>
</beans>

```

12.6. USING THE HTTP TRANSPORT IN DECOUPLED MODE

Overview

In normal HTTP request/response scenarios, the request and the response are sent using the same HTTP connection. The service provider processes the request and responds with a response containing the appropriate HTTP status code and the contents of the response. In the case of a successful request, the HTTP status code is set to 200.

In some instances, such as when using WS-RM or when requests take an extended period of time to execute, it makes sense to decouple the request and response message. In this case the service

providers sends the consumer a 202 Accepted response to the consumer over the back-channel of the HTTP connection on which the request was received. It then processes the request and sends the response back to the consumer using a new decoupled server→client HTTP connection. The consumer runtime receives the incoming response and correlates it with the appropriate request before returning to the application code.

Configuring decoupled interactions

Using the HTTP transport in decoupled mode requires that you do the following:

1. Configure the consumer to use WS-Addressing.
See [the section called “Configuring an endpoint to use WS-Addressing”](#).
2. Configure the consumer to use a decoupled endpoint.
See [the section called “Configuring the consumer”](#).
3. Configure any service providers that the consumer interacts with to use WS-Addressing.
See [the section called “Configuring an endpoint to use WS-Addressing”](#).

Configuring an endpoint to use WS-Addressing

Specify that the consumer and any service provider with which the consumer interacts use WS-Addressing.

You can specify that an endpoint uses WS-Addressing in one of two ways:

- Adding the **wswa:UsingAddressing** element to the endpoint’s WSDL **port** element as shown in [Example 12.18, “Activating WS-Addressing using WSDL”](#).

Example 12.18. Activating WS-Addressing using WSDL

```
...
<service name="WidgetSOAPService">
  <port name="WidgetSOAPPoRt" binding="tns:WidgetSOAPBinding">
    <soap:address="http://widgetvendor.net/widgetSeller" />
    <wsa:UsingAddressing xmlns:wsa="http://www.w3.org/2005/02/addressing/wsdl"/>
  </port>
</service>
...
```

- Adding the WS-Addressing policy to the endpoint’s WSDL **port** element as shown in [Example 12.19, “Activating WS-Addressing using a Policy”](#).

Example 12.19. Activating WS-Addressing using a Policy

```
...
<service name="WidgetSOAPService">
  <port name="WidgetSOAPPoRt" binding="tns:WidgetSOAPBinding">
    <soap:address="http://widgetvendor.net/widgetSeller" />
    <wsp:Policy xmlns:wsp="http://www.w3.org/2006/07/ws-policy"> <wsam:Addressing
      xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata"> <wsp:Policy/>
    </wsam:Addressing> </wsp:Policy>
  </port>
</service>
...
```

**NOTE**

The WS-Addressing policy supersedes the **wswa:UsingAddressing** WSDL element.

Configuring the consumer

Configure the consumer endpoint to use a decoupled endpoint using the **DecoupledEndpoint** attribute of the **http-conf:conduit** element.

[Example 12.20, “Configuring a Consumer to Use a Decoupled HTTP Endpoint”](#) shows the configuration for setting up the endpoint defined in [Example 12.18, “Activating WS-Addressing using WSDL”](#) to use a decoupled endpoint. The consumer now receives all responses at <http://widgetvendor.net/widgetSellerInbox>.

Example 12.20. Configuring a Consumer to Use a Decoupled HTTP Endpoint

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http="http://cxf.apache.org/transports/http/configuration"
       xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
                           http://cxf.apache.org/schemas/configuration/http-conf.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

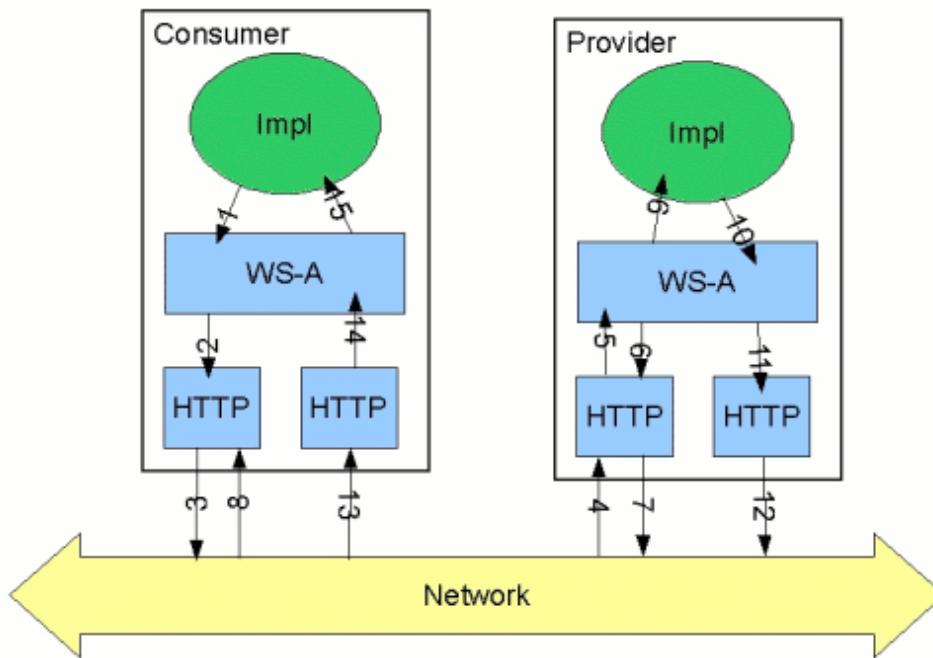
    <http:conduit name="{http://widgetvendor.net/services}WidgetSOAPPort.http-conduit">
        <http:client DecoupledEndpoint="http://widgetvendor.net:9999/decoupled_endpoint" />
    </http:conduit>
</beans>
```

How messages are processed

Using the HTTP transport in decoupled mode adds extra layers of complexity to the processing of HTTP messages. While the added complexity is transparent to the implementation level code in an application, it might be important to understand what happens for debugging reasons.

[Figure 12.1, “Message Flow in for a Decoupled HTTP Transport”](#) shows the flow of messages when using HTTP in decoupled mode.

Figure 12.1. Message Flow in for a Decoupled HTTP Transport



A request starts the following process:

1. The consumer implementation invokes an operation and a request message is generated.
2. The WS-Addressing layer adds the WS-A headers to the message.
When a decoupled endpoint is specified in the consumer's configuration, the address of the decoupled endpoint is placed in the WS-A ReplyTo header.
3. The message is sent to the service provider.
4. The service provider receives the message.
5. The request message from the consumer is dispatched to the provider's WS-A layer.
6. Because the WS-A ReplyTo header is not set to anonymous, the provider sends back a message with the HTTP status code set to 202, acknowledging that the request has been received.
7. The HTTP layer sends a 202 Accepted message back to the consumer using the original connection's back-channel.
8. The consumer receives the 202 Accepted reply on the back-channel of the HTTP connection used to send the original message.
When the consumer receives the 202 Accepted reply, the HTTP connection closes.
9. The request is passed to the service provider's implementation where the request is processed.

10. When the response is ready, it is dispatched to the WS-A layer.
11. The WS-A layer adds the WS-Addressing headers to the response message.
12. The HTTP transport sends the response to the consumer's decoupled endpoint.
13. The consumer's decoupled endpoint receives the response from the service provider.
14. The response is dispatched to the consumer's WS-A layer where it is correlated to the proper request using the WS-A RelatesTo header.
15. The correlated response is returned to the client implementation and the invoking call is unblocked.

CHAPTER 13. USING SOAP OVER JMS

Abstract

Apache CXF implements the W3C standard SOAP/JMS transport. This standard is intended to provide a more robust alternative to SOAP/HTTP services. Apache CXF applications using this transport should be able to interoperate with applications that also implement the SOAP/JMS standard. The transport is configured directly in an endpoint's WSDL.

NOTE: Support for the JMS 1.0.2 APIs has been removed in CXF 3.0. If you are using Red Hat JBoss Fuse 6.2 or higher (includes CXF 3.0), your JMS provider must support the JMS 1.1 APIs.

13.1. BASIC CONFIGURATION

Overview

The [SOAP over JMS protocol](#) is defined by the World Wide Web Consortium(W3C) as a way of providing a more reliable transport layer to the customary SOAP/HTTP protocol used by most services. The Apache CXF implementation is fully compliant with the specification and should be compatible with any framework that is also compliant.

This transport uses JNDI to find the JMS destinations. When an operation is invoked, the request is packaged as a SOAP message and sent in the body of a JMS message to the specified destination.

To use the SOAP/JMS transport:

1. Specify that the transport type is SOAP/JMS.
2. Specify the target destination using a JMS URI.
3. Optionally, configure the JNDI connection.
4. Optionally, add additional JMS configuration.

Specifying the JMS transport type

You configure a SOAP binding to use the JMS transport when specifying the WSDL binding. You set the **soap:binding** element's **transport** attribute to <http://www.w3.org/2010/soapjms/>. [Example 13.1, "SOAP over JMS binding specification"](#) shows a WSDL binding that uses SOAP/JMS.

Example 13.1. SOAP over JMS binding specification

```
<wsdl:binding ... >
  <soap:binding style="document"
    transport="http://www.w3.org/2010/soapjms/" />
  ...
</wsdl:binding>
```

Specifying the target destination

You specify the address of the JMS target destination when specifying the WSDL port for the endpoint. The address specification for a SOAP/JMS endpoint uses the same **soap:address** element and

attribute as a SOAP/HTTP endpoint. The difference is the address specification. JMS endpoints use a JMS URI as defined in the [URI Scheme for JMS 1.0](#). [Example 13.2, “JMS URI syntax”](#) shows the syntax for a JMS URI.

Example 13.2. JMS URI syntax

```
jms:variant:destination?options
```

[Table 13.1, “JMS URI variants”](#) describes the available variants for the JMS URI.

Table 13.1. JMS URI variants

Variant	Description
jndi	Specifies that the destination name is a JNDI queue name. When using this variant, you must provide the configuration for accessing the JNDI provider.
jndi-topic	Specifies that the destination name is a JNDI topic name. When using this variant, you must provide the configuration for accessing the JNDI provider.
queue	Specifies that the destination is a queue name resolved using JMS. The string provided is passed into Session.createQueue() to create a representation of the destination.
topic	Specifies that the destination is a topic name resolved using JMS. The string provided is passed into Session.createTopic() to create a representation of the destination.

The *options* portion of a JMS URI are used to configure the transport and are discussed in [Section 13.2, “JMS URIs”](#).

[Example 13.3, “SOAP/JMS endpoint address”](#) shows the WSDL port entry for a SOAP/JMS endpoint whose target destination is looked up using JNDI.

Example 13.3. SOAP/JMS endpoint address

```
<wsdl:port ... >
...
<soap:address location="jms:jndi:dynamicQueues/test.cxf.jmstransport.queue" />
</wsdl:port>
```

Configuring JNDI and the JMS transport

The SOAP/JMS provides several ways to configure the JNDI connection and the JMS transport:

- Section 13.2, "JMS URIs"
- Section 13.3, "WSDL extensions"

13.2. JMS URIS

Overview

When using SOAP/JMS, a JMS URI is used to specify the endpoint's target destination. The JMS URI can also be used to configure JMS connection by appending one or more options to the URI. These options are detailed in the IETF standard, [URI Scheme for Java Message Service 1.0](#). They can be used to configure the JNDI system, the reply destination, the delivery mode to use, and other JMS properties.

Syntax

As shown in [Example 13.4, "Syntax for JMS URI options"](#), you can append one or more options to the end of a JMS URI by separating them from the destination's address with a question mark(?). Multiple options are separated by an ampersand(&). [Example 13.4, "Syntax for JMS URI options"](#) shows the syntax for using multiple options in a JMS URI.

Example 13.4. Syntax for JMS URI options

```
jms:variant:jmsAddress?option1=value1&option2=value2&_optionN_=valueN
```

JMS properties

[Table 13.2, "JMS properties settable as URI options"](#) shows the URI options that affect the JMS transport layer.

Table 13.2. JMS properties settable as URI options

Property	Default	Description
conduitIdSelectorPrefix		[Optional] A string value that is prefixed to all correlation IDs that the conduit creates. The selector can use it to listen for replies.
deliveryMode	PERSISTENT	Specifies whether to use JMS PERSISTENT or NON_PERSISTENT message semantics. In the case of PERSISTENT delivery mode, the JMS broker stores messages in persistent storage before acknowledging them; whereas NON_PERSISTENT messages are kept in memory only.

Property	Default	Description
durableSubscriptionClientID		[Optional] Specifies the client identifier for the connection. This property is used to associate a connection with a state that the provider maintains on behalf of the client. This enables subsequent subscribers with the same identity to resume the subscription in the state that the preceding subscriber left it.
durableSubscriptionName		[Optional] Specifies the name of the subscription.
messageType	byte	Specifies the JMS message type used by CXF. Valid values are: <ul style="list-style-type: none"> ● byte ● text ● binary
password		[Optional] Specifies the password for creating the connection. Appending this property to the URI is discouraged.
priority	4	Specifies the JMS message priority, which ranges from 0 (lowest) to 9 (highest).
receiveTimout	60000	Specifies the time, in milliseconds, the client will wait for a reply when request/reply exchanges are used.
reconnectOnException	true	[Deprecated in CXF 3.0] <p>Specifies whether the transport should reconnect when exceptions occur.</p> <p>As of 3.0, the transport will always reconnect when an exception occurs.</p>

Property	Default	Description
replyToName		<p>[Optional] Specifies the reply destination for queue messages. The reply destination appears in the JMSReplyTo header. Setting this property is recommended for applications that have request-reply semantics because the JMS provider will assign a temporary reply queue if one is not specified.</p> <p>The value of this property is interpreted according to the variant specified in the JMS URI:</p> <ul style="list-style-type: none"> • jndi variant—the name of the destination queue resolved by JNDI • queue variant—the name of the destination queue resolved using JMS
sessionTransacted	false	<p>Specifies the transaction type. Valid values are:</p> <ul style="list-style-type: none"> • true—resource local transactions • false—JTA transactions
timeToLive	0	<p>Specifies the time, in milliseconds, after which the JMS provider will discard the message. A value of 0 indicates an infinite lifetime.</p>
topicReplyToName		<p>[Optional] Specifies the reply destination for topic messages. The value of this property is interpreted according to the variant specified in the JMS URI:</p> <ul style="list-style-type: none"> • jndi-topic—the name of the destination topic resolved by JNDI • topic—the name of the destination topic resolved by JMS

Property	Default	Description
useConduitIdSelector	true	<p>Specifies whether the conduit's UUID will be used as the prefix for all correlation IDs.</p> <p>As all conduits are assigned a unique UUID, setting this property to true enables multiple endpoints to share a JMS queue or topic.</p>
username		[Optional] Specifies the username to use to create the connection.

JNDI properties

Table 13.3, “JNDI properties settable as URI options” shows the URI options that can be used to configure JNDI for this endpoint.

Table 13.3. JNDI properties settable as URI options

Property	Description
jndiConnectionFactoryName	Specifies the JNDI name of the JMS connection factory.
jndiInitialContextFactory	Specifies the fully qualified Java class name of the JNDI provider (which must be of javax.jms.InitialContextFactory type). Equivalent to setting the java.naming.factory.initial Java system property.
jndiTransactionManagerName	Specifies the name of the JTA transaction manager that will be searched for in Spring, Blueprint, or JNDI. If a transaction manager is found, JTA transactions will be enabled. See the sessionTransacted JMS property.
jndiURL	Specifies the URL that initializes the JNDI provider. Equivalent to setting the java.naming.provider.url Java system property.

Additional JNDI properties

The properties, **java.naming.factory.initial** and **java.naming.provider.url**, are standard properties, which are required to initialize any JNDI provider. Sometimes, however, a JNDI provider might support custom properties in addition to the standard ones. In this case, you can set an arbitrary JNDI property by setting a URL option of the form **jndi-PropertyName**.

For example, if you were using SUN's LDAP implementation of JNDI, you could set the JNDI property, **java.naming.factory.control**, in a JMS URI as shown in [Example 13.5, "Setting a JNDI property in a JMS URI"](#).

Example 13.5. Setting a JNDI property in a JMS URI

```
jms:queue:FOO.BAR?jndi-
  java.naming.factory.control=com.sun.jndi.ldap.ResponseControlFactory
```

Example

If the JMS provider is **not** already configured, it is possible to provide the requisite JNDI configuration details in the URI using options (see [Table 13.3, "JNDI properties settable as URI options"](#)). For example, to configure an endpoint to use the Apache ActiveMQ JMS provider and connect to the queue called **test.cxf.jmstransport.queue**, use the URI shown in [Example 13.6, "JMS URI that configures a JNDI connection"](#).

Example 13.6. JMS URI that configures a JNDI connection

```
jms:jndi:dynamicQueues/test.cxf.jmstransport.queue
  ?jndiInitialContextFactory=org.apache.activemq.jndi.ActiveMQInitialContextFactory
  &jndiConnectionFactoryName=ConnectionFactory
  &jndiURL=tcp://localhost:61616
```

Publishing a service

The JAX-WS standard **publish()** method cannot be used to publish a SOAP/JMS service. Instead, you must use the Apache CXF's **JaxWsServerFactoryBean** class as shown in [Example 13.7, "Publishing a SOAP/JMS service"](#).

Example 13.7. Publishing a SOAP/JMS service

```
String address = "jms:jndi:dynamicQueues/test.cxf.jmstransport.queue3"
  + "?jndiInitialContextFactory"
  + "=org.apache.activemq.jndi.ActiveMQInitialContextFactory"
  + "&jndiConnectionFactoryName=ConnectionFactory"
  + "&jndiURL=tcp://localhost:61500";
Hello implementor = new HelloImpl();
JaxWsServerFactoryBean svrFactory = new JaxWsServerFactoryBean();
svrFactory.setServiceClass(Hello.class);
svrFactory.setAddress(address);
svrFactory.setTransportId(JMSSpecConstants.SOAP_JMS_SPECIFICATION_TRANSPORTID);
svrFactory.setServiceBean(implementor);
svrFactory.create();
```

The code in [Example 13.7, "Publishing a SOAP/JMS service"](#) does the following:

Creates the JMS URI representing the endpoint's address.

Instantiates a **JaxWsServerFactoryBean** to publish the service.

Sets the **address** field of the factory bean with the JMS URI of the service.

Specifies that the service created by the factory will use the SOAP/JMS transport.

Consuming a service

The standard JAX-WS APIs cannot be used to consume a SOAP/JMS service. Instead, you must use the Apache CXF's **JaxWsProxyFactoryBean** class as shown in [Example 13.8, "Consuming a SOAP/JMS service"](#).

Example 13.8. Consuming a SOAP/JMS service

```
// Java
public void invoke() throws Exception {
    String address = "jms:jndi:dynamicQueues/test.cxf.jmstransport.queue3"
        + "?jndiInitialContextFactory"
        + "=org.apache.activemq.jndi.ActiveMQInitialContextFactory"
        + "&jndiConnectionFactory=ConnectionFactory&jndiURL=tcp://localhost:61500";
    JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
    factory.setAddress(address);
    factory.setTransportId(JMSSpecConstants.SOAP_JMS_SPECIFICATION_TRANSPORTID);
    factory.setServiceClass(Hello.class);
    Hello client = (Hello)factory.create();
    String reply = client.sayHi(" HI");
    System.out.println(reply);
}
```

The code in [Example 13.8, "Consuming a SOAP/JMS service"](#) does the following:

Creates the JMS URI representing the endpoint's address.

Instantiates a **JaxWsProxyFactoryBean** to create the proxy.

Sets the **address** field of the factory bean with the JMS URI of the service.

Specifies that the proxy created by the factory will use the SOAP/JMS transport.

13.3. WSDL EXTENSIONS

Overview

You can specify the basic configuration of the JMS transport by inserting WSDL extension elements into the contract, either at binding scope, service scope, or port scope. The WSDL extensions enable you to specify the properties for bootstrapping a JNDI **InitialContext**, which can then be used to look up JMS destinations. You can also set some properties that affect the behavior of the JMS transport layer.

SOAP/JMS namespace

the SOAP/JMS WSDL extensions are defined in the <http://www.w3.org/2010/soapjms/> namespace. To use them in your WSDL contracts add the following setting to the **wsdl:definitions** element:

```
<wsdl:definitions ...
  xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  ... >
```

WSDL extension elements

Table 13.4, “SOAP/JMS WSDL extension elements” shows all of the WSDL extension elements you can use to configure the JMS transport.

Table 13.4. SOAP/JMS WSDL extension elements

Element	Default	Description
soapjms:jndiInitialContextFactory		Specifies the fully qualified Java class name of the JNDI provider. Equivalent to setting the java.naming.factory.initial Java system property.
soapjms:jndiURL		Specifies the URL that initializes the JNDI provider. Equivalent to setting the java.naming.provider.url Java system property.
soapjms:jndiContextParameter		Specifies an additional property for creating the JNDI InitialContext . Use the name and value attributes to specify the property.
soapjms:jndiConnectionFactoryName		Specifies the JNDI name of the JMS connection factory.
soapjms:deliveryMode	PERSISTENT	Specifies whether to use JMS PERSISTENT or NON_PERSISTENT message semantics. In the case of PERSISTENT delivery mode, the JMS broker stores messages in persistent storage before acknowledging them; whereas NON_PERSISTENT messages are kept in memory only.

Element	Default	Description
soapjms:replyToName		<p>[Optional] Specifies the reply destination for queue messages. The reply destination appears in the JMSReplyTo header. Setting this property is recommended for applications that have request-reply semantics because the JMS provider will assign a temporary reply queue if one is not specified.</p> <p>The value of this property is interpreted according to the variant specified in the JMS URI:</p> <ul style="list-style-type: none"> • jndi variant—the name of the destination queue resolved by JNDI • queue variant—the name of the destination queue resolved using JMS
soapjms:priority	4	Specifies the JMS message priority, which ranges from 0 (lowest) to 9 (highest).
soapjms:timeToLive	0	Time, in milliseconds, after which the JMS provider will discard the message. A value of 0 represents an infinite lifetime.

Configuration scopes

The WSDL elements placement in the WSDL contract effect the scope of the configuration changes on the endpoints defined in the contract. The SOAP/JMS WSDL elements can be placed as children of either the **wsdl:binding** element, the **wsdl:service** element, or the **wsdl:port** element. The parent of the SOAP/JMS elements determine which of the following scopes the configuration is placed into.

Binding scope

You can configure the JMS transport at the *binding* scope by placing extension elements inside the **wsdl:binding** element. Elements in this scope define the default configuration for all endpoints that use this binding. Any settings in the binding scope can be overridden at the service scope or the port scope.

Service scope

You can configure the JMS transport at the *service* scope by placing extension elements inside a **wsdl:service** element. Elements in this scope define the default configuration for all endpoints in this service. Any settings in the service scope can be overridden at the port scope.

Port scope

You can configure the JMS transport at the *port* scope by placing extension elements inside a **wsdl:port** element. Elements in the port scope define the configuration for this port. They override the defaults of the same extension elements defined at the service scope or at the binding scope.

Example

[Example 13.9, “WSDL contract with SOAP/JMS configuration”](#) shows a WSDL contract for a SOAP/JMS service. It configures the JNDI layer in the binding scope, the message delivery details in the service scope, and the reply destination in the port scope.

Example 13.9. WSDL contract with SOAP/JMS configuration

```
<wsdl:definitions ...
  xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  ...
  ...
<wsdl:binding name="JMSGreeterPortBinding" type="tns:JMSGreeterPortType">
  ...
  <soapjms:jndiInitialContextFactory>
    org.apache.activemq.jndi.ActiveMQInitialContextFactory
  </soapjms:jndiInitialContextFactory>
  <soapjms:jndiURL>tcp://localhost:61616</soapjms:jndiURL>
  <soapjms:jndiConnectionFactoryName>
    ConnectionFactory
  </soapjms:jndiConnectionFactoryName>
  ...
</wsdl:binding>
...
<wsdl:service name="JMSGreeterService">
  ...
  <soapjms:deliveryMode>NON_PERSISTENT</soapjms:deliveryMode>
  <soapjms:timeToLive>60000</soapjms:timeToLive>
  ...
  <wsdl:port binding="tns:JMSGreeterPortBinding" name="GreeterPort">
    <soap:address location="jms:jndi:dynamicQueues/test.cxf.jmstransport.queue" />
    <soapjms:replyToName>
      dynamicQueues/greeterReply.queue
    </soapjms:replyToName>
    ...
  </wsdl:port>
  ...
</wsdl:service>
...
</wsdl:definitions>
```

The WSDL in [Example 13.9, “WSDL contract with SOAP/JMS configuration”](#) does the following:

- Declares the namespace for the SOAP/JMS extensions.

- Configures the JNDI connections in the binding scope.

- Sets the JMS delivery style to non-persistent and each message to live for one minute.

Specifies the target destination.

Configures the JMS transport so that reply messages are delivered on the **greeterReply.queue** queue.

CHAPTER 14. USING GENERIC JMS

Abstract

Apache CXF provides a generic implementation of a JMS transport. The generic JMS transport is not restricted to using SOAP messages and allows for connecting to any application that uses JMS.

NOTE: Support for the JMS 1.0.2 APIs has been removed in CXF 3.0. If you are using Red Hat JBoss Fuse 6.2 or higher (includes CXF 3.0), your JMS provider must support the JMS 1.1 APIs.

14.1. APPROACHES TO CONFIGURING JMS

The Apache CXF generic JMS transport can connect to any JMS provider and work with applications that exchange JMS messages with bodies of either **TextMessage** or **ByteMessage**.

There are two ways to enable and configure the JMS transport:

- [Section 14.2, "Using the JMS configuration bean"](#)
- [Section 14.5, "Using WSDL to configure JMS"](#)

14.2. USING THE JMS CONFIGURATION BEAN

Overview

To simplify JMS configuration and make it more powerful, Apache CXF uses a single JMS configuration bean to configure JMS endpoints. The bean is implemented by the **org.apache.cxf.transport.jms.JMSConfiguration** class. It can be used to either configure endpoint's directly or to configure the JMS conduits and destinations.

Configuration namespace

The JMS configuration bean uses the [Spring p-namespace](#) to make the configuration as simple as possible. To use this namespace you need to declare it in the configuration's root element as shown in [Example 14.1, "Declaring the Spring p-namespace"](#).

Example 14.1. Declaring the Spring p-namespace

```
<beans ...
  xmlns:p="http://www.springframework.org/schema/p"
  ...
>
...
</beans>
```

Specifying the configuration

You specify the JMS configuration by defining a bean of class **org.apache.cxf.transport.jms.JMSConfiguration**. The properties of the bean provide the configuration settings for the transport.



IMPORTANT

In CXF 3.0, the JMS transport no longer has a dependency on Spring JMS, so some Spring JMS-related options have been removed.

[Table 14.1, “General JMS Configuration Properties”](#) lists properties that are common to both providers and consumers.

Table 14.1. General JMS Configuration Properties

Property	Default	Description
connectionFactory		[Required] Specifies a reference to a bean that defines a JMS ConnectionFactory.
wrapInSingleConnectionFactory	true [pre v3.0]	<p>Removed in CXF 3.0</p> <p>pre CXF 3.0 Specifies whether to wrap the ConnectionFactory with a Spring SingleConnectionFactory.</p> <p>Enable this property when using a ConnectionFactory that does not pool connections, as it will improve the performance of the JMS transport. This is so because the JMS transport creates a new connection for each message, and the SingleConnectionFactory is needed to cache the connection, so it can be reused.</p>

Property	Default	Description
reconnectOnException	false	<p>Deprecated in CXF 3.0 CXF always reconnects when an exception occurs.</p> <p>pre CXF 3.0 Specifies whether to create a new connection when an exception occurs.</p> <p>When wrapping the ConnectionFactory with a Spring SingleConnectionFactory:</p> <ul style="list-style-type: none"> • true – on an exception, create a new connection Do not enable this option when using a PooledConnectionFactory, as this option only returns the pooled connection, but does not reconnect. • false – on an exception, do not try to reconnect
targetDestination		Specifies the JNDI name or provider-specific name of a destination.
replyDestination		Specifies the JMS name of the JMS destination where replies are sent. This property allows the use of a user-defined destination for replies. For more details see Section 14.6, “Using a Named Reply Destination” .

Property	Default	Description
destinationResolver	DynamicDestinationResolver	<p>Specifies a reference to a Spring DestinationResolver.</p> <p>This property allows you to define how destination names are resolved to JMS destinations. Valid values are:</p> <ul style="list-style-type: none"> • DynamicDestinationResolver – resolve destination names using the features of the JMS provider. • JndiDestinationResolver – resolve destination names using JNDI.
transactionManager		Specifies a reference to a Spring transaction manager. This enables the service to participate in JTA transactions.
taskExecutor	SimpleAsyncTaskExecutor	<p>Removed in CXF 3.0</p> <p>pre CXF 3.0 Specifies a reference to a Spring TaskExecutor. This is used in listeners to decide how to handle incoming messages.</p>
useJms11	false	<p>Removed in CXF 3.0 CXF 3.0 supports JMS 1.1 features only.</p> <p>pre CXF 3.0 Specifies whether JMS 1.1 features are used. Valid values are:</p> <ul style="list-style-type: none"> • true – JMS 1.1 features • false – JMS 1.0.2 features

Property	Default	Description
<code>messageIdEnabled</code>	<code>true</code>	<p>Removed in CXF 3.0</p> <p>pre CXF 3.0 Specifies whether the JMS transport wants the JMS broker to provide message IDs. Valid values are:</p> <ul style="list-style-type: none"> • true – broker needs to provide message IDs • false – broker need not provide message IDs. In this case, the endpoint calls its message producer's <code>setDisableMessageID()</code> method with a value of true. The broker is then given a hint that it need not generate message IDs or add them to the endpoint's messages. The broker either accepts the hint or ignores it.
<code>messageTimestampEnabled</code>	<code>true</code>	<p>Removed in CXF 3.0</p> <p>pre CXF 3.0 Specifies whether the JMS transport wants the JMS broker to provide message time stamps. Valid values are:</p> <ul style="list-style-type: none"> • true – broker needs to provide message timestamps • false – broker need not provide message timestamps. In this case, the endpoint calls its message producer's <code>setDisableMessageTimestamp()</code> method with a value of true. The broker is then given a hint that it need not generate time stamps or add them to the endpoint's messages. The broker either accepts the hint or ignores it.

Property	Default	Description
cacheLevel	-1 (feature disabled)	<p>Removed in CXF 3.0</p> <p>pre CXF 3.0 Specifies the level of caching that the JMS listener container may apply. Valid values are:</p> <ul style="list-style-type: none"> • 0 – CACHE_NONE • 1 – CACHE_CONNECTION • 2 – CACHE_SESSION • 3 – CACHE_CONSUMER • 4 – CACHE_AUTO <p>For details, see Class DefaultMessageListenerContainer</p>
pubSubNoLocal	false	<p>Specifies whether to receive your own messages when using topics.</p> <ul style="list-style-type: none"> • true – do not receive your own messages • false – receive your own messages
receiveTimeout	60000	Specifies the time, in milliseconds, to wait for response messages.
explicitQosEnabled	false	Specifies whether the QoS settings (such as priority, persistence, time to live) are explicitly set for each message (true) or use the default values (false).
deliveryMode	2	<p>Specifies whether a message is persistent. Valid values are:</p> <ul style="list-style-type: none"> • 1 (NON_PERSISTENT)– messages are kept in memory only • 2 (PERSISTENT)– messages are persisted to disk

Property	Default	Description
priority	4	Specifies message priority. JMS priority values range from 0 (lowest) to 9 (highest). See your JMS provider's documentation for details.
timeToLive	0 (indefinitely)	Specifies the time, in milliseconds, before a message that has been sent is discarded.
sessionTransacted	false	Specifies whether JMS transactions are used.
concurrentConsumers	1	Removed in CXF 3.0 pre CXF 3.0 Specifies the minimum number of concurrent consumers for the listener.
maxConcurrentConsumers	1	Removed in CXF 3.0 pre CXF 3.0 Specifies the maximum number of concurrent consumers for the listener.
messageSelector		Specifies the string value of the selector used to filter incoming messages. This property enables multiple connections to share a queue. For more information on the syntax used to specify message selectors, see the JMS 1.1 specification .
subscriptionDurable	false	Specifies whether the server uses durable subscriptions.
durableSubscriptionName		Specifies the name (string) used to register the durable subscription.

Property	Default	Description
messageType	text	<p>Specifies how the message data will be packaged as a JMS message. Valid values are:</p> <ul style="list-style-type: none"> • text – specifies that the data will be packaged as a TextMessage • byte – specifies that the data will be packaged as an array of bytes (byte[]) • binary – specifies that the data will be packaged as an ByteMessage
pubSubDomain	false	<p>Specifies whether the target destination is a topic or a queue. Valid values are:</p> <ul style="list-style-type: none"> • true – topic • false – queue
jmsProviderTibcoEms	false	<p>Specifies whether the JMS provider is Tibco EMS.</p> <p>When set to true, the principal in the security context is populated from the JMS_TIBCO_SENDER header.</p>
useMessageIDAsCorrelationID	false	<p>Removed in CXF 3.0</p> <p>Specifies whether JMS will use the message ID to correlate messages.</p> <p>When set to true, the client sets a generated correlation ID.</p>
maxSuspendedContinuations	-1 (feature disabled)	<p>CXF 3.0 Specifies the maximum number of suspended continuations the JMS destination may have. When the current number exceeds the specified maximum, the JMSListenerContainer is stopped.</p>

Property	Default	Description
reconnectPercentOfMax	70	<p>CXF 3.0 Specifies when to restart the JMSListenerContainer stopped for exceeding maxSuspendedContinuations.</p> <p>The listener container is restarted when its current number of suspended continuations falls below the value of (maxSuspendedContinuations * reconnectPercentOfMax/100).</p>

As shown in [Example 14.2, "JMS configuration bean"](#), the bean's properties are specified as attributes to the **bean** element. They are all declared in the Spring **p** namespace.

Example 14.2. JMS configuration bean

```
<bean id="jmsConfig"
  class="org.apache.cxf.transport.jms.JMSConfiguration"
  p:connectionFactory="jmsConnectionFactory"
  p:targetDestination="dynamicQueues/greeter.request.queue"
  p:pubSubDomain="false" />
```

Applying the configuration to an endpoint

The **JMSConfiguration** bean can be applied directly to both server and client endpoints using the Apache CXF features mechanism. To do so:

1. Set the endpoint's **address** attribute to **jms://**.
2. Add a **jaxws:feature** element to the endpoint's configuration.
3. Add a bean of type **org.apache.cxf.transport.jms.JMSConfigFeature** to the feature.
4. Set the **bean** element's **p:jmsConfig-ref** attribute to the ID of the **JMSConfiguration** bean.

[Example 14.3, "Adding JMS configuration to a JAX-WS client"](#) shows a JAX-WS client that uses the JMS configuration from [Example 14.2, "JMS configuration bean"](#).

Example 14.3. Adding JMS configuration to a JAX-WS client

```
<jaxws:client id="CustomerService"
  xmlns:customer="http://customerservice.example.com/"
  serviceName="customer:CustomerServiceService"
  endpointName="customer:CustomerServiceEndpoint"
  address="jms://"
  serviceClass="com.example.customerservice.CustomerService">
```

```
<jaxws:features>
  <bean xmlns="http://www.springframework.org/schema/beans"
        class="org.apache.cxf.transport.jms.JMSConfigFeature"
        p:jmsConfig-ref="jmsConfig"/>
</jaxws:features>
</jaxws:client>
```

Applying the configuration to the transport

The **JMSConfiguration** bean can be applied to JMS conduits and JMS destinations using the **jms:jmsConfig-ref** element. The **jms:jmsConfig-ref** element's value is the ID of the **JMSConfiguration** bean.

[Example 14.4, "Adding JMS configuration to a JMS conduit"](#) shows a JMS conduit that uses the JMS configuration from [Example 14.2, "JMS configuration bean"](#).

Example 14.4. Adding JMS configuration to a JMS conduit

```
<jms:conduit name="{http://cxf.apache.org/jms_conf_test}HelloWorldQueueBinMsgPort.jms-
conduit">
  ...
  <jms:jmsConfig-ref>jmsConf</jms:jmsConfig-ref>
</jms:conduit>
```

14.3. OPTIMIZING CLIENT-SIDE JMS PERFORMANCE

Overview

Two major settings affect the JMS performance of clients: pooling and synchronous receives.

Pooling

On the client side, CXF creates a new JMS session and JMS producer for each message. This is so because neither session nor producer objects are thread safe. Creating a producer is especially time intensive because it requires communicating with the server.

Pooling connection factories improves performance by caching the connection, session, and producer.

For ActiveMQ, configuring pooling is simple; for example:

```
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.pool.PooledConnectionFactory;

ConnectionFactory cf = new ActiveMQConnectionFactory("tcp://localhost:61616");
PooledConnectionFactory pcf = new PooledConnectionFactory();

//Set expiry timeout because the default (0) prevents reconnection on failure
pcf.setExpiryTimeout(5000);
pcf.setConnectionFactory(cf);
```

```
JMSConfiguration jmsConfig = new JMSConfiguration();
jmsConfig.setConnectionFactory(pdf);
```

For more information on pooling, see "Appendix A Optimizing Performance of JMS Single- and Multiple-Resource Transactions" in the [Red Hat JBoss Fuse Transaction Guide](#)

Avoiding synchronous receives

For request/reply exchanges, the JMS transport sends a request and then waits for a reply. Whenever possible, request/reply messaging is implemented asynchronously using a JMS **MessageListener**.

However, CXF must use a synchronous **Consumer.receive()** method when it needs to share queues between endpoints. This scenario requires the **MessageListener** to use a message selector to filter the messages. The message selector must be known in advance, so the **MessageListener** is opened only once.

Two cases in which the message selector cannot be known in advance should be avoided:

- When **JMSMessageID** is used as the **JMSCorrelationID**
If the JMS properties **useConduitIdSelector** and **conduitSelectorPrefix** are not set on the JMS transport, the client does not set a **JMSCorrelationId**. This causes the server to use the **JMSMessageId** of the request message as the **JMSCorrelationId**. As **JMSMessageID** cannot be known in advance, the client has to use a synchronous **Consumer.receive()** method.

Note that you must use the **Consumer.receive()** method with IBM JMS endpoints (their default).
- The user sets the **JMstype** in the request message and then sets a custom **JMSCorrelationID**. Again, as the custom **JMSCorrelationID** cannot be known in advance, the client has to use a synchronous **Consumer.receive()** method.

So the general rule is to avoid using settings that require using a synchronous receive.

14.4. CONFIGURING JMS TRANSACTIONS

Overview

CXF 3.0 supports both local JMS transactions and JTA transactions on CXF endpoints, when using one-way messaging.

Local transactions

Transactions using local resources roll back the JMS message only when an exception occurs. They do not directly coordinate other resources, such as database transactions.

To set up a local transaction, configure the endpoint as you normally would, and set the property **sessionTrasnsacted** to **true**.



NOTE

For more information on transactions and pooling, see the [Red Hat JBoss Fuse Transaction Guide](#).

JTA transactions

Using JTA transactions, you can coordinate any number of XA resources. If a CXF endpoint is configured for JTA transactions, it starts a transaction before calling the service implementation. The transaction will be committed if no exception occurs. Otherwise, it will be rolled back.

In JTA transactions, a JMS message is consumed and the data written to a database. When an exception occurs, both resources are rolled back, so either the message is consumed and the data is written to the database, or the message is rolled back and the data is not written to the database.

Configuring JTA transactions requires two steps:

1. Defining a transaction manager

- bean method
 - Define a transaction manager

```
<bean id="transactionManager"
  class="org.apache.geronimo.transaction.manager.GeronimoTransactionManager"/>
```

- Set the name of the transaction manager in the JMS URI

```
jms:queue:myqueue?jndiTransactionManager=TransactionManager
```

This example finds a bean with the ID **TransactionManager**.

- OSGi reference method

- Look up the transaction manager as an OSGi service using Blueprint

```
<reference id="TransactionManager"
  interface="javax.transaction.TransactionManager"/>
```

- Set the name of the transaction manager in the JMS URI

```
jms:jndi:myqueue?jndiTransactionManager=java:comp/env/TransactionManager
```

This example looks up the transaction manager in JNDI.

2. Configuring a JCA pooled connection factory

Using Spring to define the JCA pooled connection factory:

```
<bean id="xacf" class="org.apache.activemq.ActiveMQXAConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616" />
</bean>

<bean id="ConnectionFactory"
  class="org.apache.activemq.jms.pool.JcaPooledConnectionFactory">
  <property name="transactionManager" ref="transactionManager" />
  <property name="connectionFactory" ref="xacf" />
</bean>
```

In this example, the first bean defines an ActiveMQ XA connection factory, which is given to a **JcaPooledConnectionFactory**. The **JcaPooledConnectionFactory** is then provided as the default bean with id **ConnectionFactory**.

Note that the **JcaPooledConnectionFactory** looks like a normal ConnectionFactory. But when a new connection and session are opened, it checks for an XA transaction and, if found, automatically registers the JMS session as an XA resource. This allows the JMS session to participate in the JMS transaction.



IMPORTANT

Directly setting an XA ConnectionFactory on the JMS transport will not work!

14.5. USING WSDL TO CONFIGURE JMS

14.5.1. JMS WSDL Extension Namespace

The WSDL extensions for defining a JMS endpoint are defined in the namespace <http://cxf.apache.org/transports/jms>. In order to use the JMS extensions you will need to add the line shown in [Example 14.5, “JMS WSDL extension namespace”](#) to the definitions element of your contract.

Example 14.5. JMS WSDL extension namespace

```
xmlns:jms="http://cxf.apache.org/transports/jms"
```

14.5.2. Basic JMS configuration

Overview

The JMS address information is provided using the **jms:address** element and its child, the **jms:JMSNamingProperties** element. The **jms:address** element's attributes specify the information needed to identify the JMS broker and the destination. The **jms:JMSNamingProperties** element specifies the Java properties used to connect to the JNDI service.



IMPORTANT

Information specified using the JMS feature will override the information in the endpoint's WSDL file.

Specifying the JMS address

The basic configuration for a JMS endpoint is done by using a **jms:address** element as the child of your service's **port** element. The **jms:address** element used in WSDL is identical to the one used in the configuration file. Its attributes are listed in [Table 14.2, “JMS endpoint attributes”](#).

Table 14.2. JMS endpoint attributes

Attribute	Description
-----------	-------------

Attribute	Description
destinationStyle	Specifies if the JMS destination is a JMS queue or a JMS topic.
jndiConnectionFactoryName	Specifies the JNDI name bound to the JMS connection factory to use when connecting to the JMS destination.
jmsDestinationName	Specifies the JMS name of the JMS destination to which requests are sent.
jmsReplyDestinationName	Specifies the JMS name of the JMS destinations where replies are sent. This attribute allows you to use a user defined destination for replies. For more details see Section 14.6, "Using a Named Reply Destination" .
jndiDestinationName	Specifies the JNDI name bound to the JMS destination to which requests are sent.
jndiReplyDestinationName	Specifies the JNDI name bound to the JMS destinations where replies are sent. This attribute allows you to use a user defined destination for replies. For more details see Section 14.6, "Using a Named Reply Destination" .
connectionUserName	Specifies the user name to use when connecting to a JMS broker.
connectionPassword	Specifies the password to use when connecting to a JMS broker.

The **jms:address** WSDL element uses a **jms:JMSNamingProperties** child element to specify additional information needed to connect to a JNDI provider.

Specifying JNDI properties

To increase interoperability with JMS and JNDI providers, the **jms:address** element has a child element, **jms:JMSNamingProperties**, that allows you to specify the values used to populate the properties used when connecting to the JNDI provider. The **jms:JMSNamingProperties** element has two attributes: **name** and **value**. **name** specifies the name of the property to set. **value** attribute specifies the value for the specified property. **jms:JMSNamingProperties** element can also be used for specification of provider specific properties.

The following is a list of common JNDI properties that can be set:

1. **java.naming.factory.initial**
2. **java.naming.provider.url**
3. **java.naming.factory.object**

4. **java.naming.factory.state**
5. **java.naming.factory.url.pkgs**
6. **java.naming.dns.url**
7. **java.naming.authoritative**
8. **java.naming.batchsize**
9. **java.naming.referral**
10. **java.naming.security.protocol**
11. **java.naming.security.authentication**
12. **java.naming.security.principal**
13. **java.naming.security.credentials**
14. **java.naming.language**
15. **java.naming.applet**

For more details on what information to use in these attributes, check your JNDI provider's documentation and consult the Java API reference material.

Example

[Example 14.6, "JMS WSDL port specification"](#) shows an example of a JMS WSDL **port** specification.

Example 14.6. JMS WSDL port specification

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
      jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
        value="org.apache.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
        value="tcp://localhost:61616" />
    </jms:address>
  </port>
</service>
```

14.5.3. JMS client configuration

Overview

JMS consumer endpoints specify the type of messages they use. JMS consumer endpoint can use either a JMS **ByteMessage** or a JMS **TextMessage**.

When using an **ByteMessage** the consumer endpoint uses a **byte[]** as the method for storing data into and retrieving data from the JMS message body. When messages are sent, the message data, including

any formating information, is packaged into a **byte[]** and placed into the message body before it is placed on the wire. When messages are received, the consumer endpoint will attempt to unmarshall the data stored in the message body as if it were packed in a **byte[]**.

When using a **TextMessage**, the consumer endpoint uses a string as the method for storing and retrieving data from the message body. When messages are sent, the message information, including any format-specific information, is converted into a string and placed into the JMS message body. When messages are received the consumer endpoint will attempt to unmarshall the data stored in the JMS message body as if it were packed into a string.

When native JMS applications interact with Apache CXF consumers, the JMS application is responsible for interpreting the message and the formatting information. For example, if the Apache CXF contract specifies that the binding used for a JMS endpoint is SOAP, and the messages are packaged as **TextMessage**, the receiving JMS application will get a text message containing all of the SOAP envelope information.

Specifying the message type

The type of messages accepted by a JMS consumer endpoint is configured using the optional **jms:client** element. The **jms:client** element is a child of the WSDL **port** element and has one attribute:

Table 14.3. JMS Client WSDL Extensions

messageType
Specifies how the message data will be packaged as a JMS message. text specifies that the data will be packaged as a TextMessage . binary specifies that the data will be packaged as an ByteMessage .

Example

[Example 14.7, "WSDL for a JMS consumer endpoint"](#) shows the WSDL for configuring a JMS consumer endpoint.

Example 14.7. WSDL for a JMS consumer endpoint

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
      jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
        value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
        value="tcp://localhost:61616" />
    </jms:address>
    <jms:client messageType="binary" />
  </port>
</service>
```

14.5.4. JMS provider configuration

Overview

JMS provider endpoints have a number of behaviors that are configurable. These include:

- how messages are correlated
- the use of durable subscriptions
- if the service uses local JMS transactions
- the message selectors used by the endpoint

Specifying the configuration

Provider endpoint behaviors are configured using the optional **jms:server** element. The **jms:server** element is a child of the WSDL **wsdl:port** element and has the following attributes:

Table 14.4. JMS provider endpoint WSDL extensions

Attribute	Description
useMessageIDAsCorrealationID	Specifies whether JMS will use the message ID to correlate messages. The default is false .
durableSubscriberName	Specifies the name used to register a durable subscription.
messageSelector	Specifies the string value of a message selector to use. For more information on the syntax used to specify message selectors, see the JMS 1.1 specification.
transactional	Specifies whether the local JMS broker will create transactions around message processing. The default is false . [a]

[a] Currently, setting the **transactional** attribute to **true** is not supported by the runtime.

Example

[Example 14.8, “WSDL for a JMS provider endpoint”](#) shows the WSDL for configuring a JMS provider endpoint.

Example 14.8. WSDL for a JMS provider endpoint

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
      jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
        value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
        value="tcp://localhost:61616" />
    </jms:address>
```

```

<jms:server messageSelector="cxf_message_selector"
    useMessageIDAsCorrelationID="true"
    transactional="true"
    durableSubscriberName="cxf_subscriber" />
</port>
</service>
```

14.6. USING A NAMED REPLY DESTINATION

Overview

By default, Apache CXF endpoints using JMS create a temporary queue for sending replies back and forth. If you prefer to use named queues, you can configure the queue used to send replies as part of an endpoint's JMS configuration.

Setting the reply destination name

You specify the reply destination using either the **jmsReplyDestinationName** attribute or the **jndiReplyDestinationName** attribute in the endpoint's JMS configuration. A client endpoint will listen for replies on the specified destination and it will specify the value of the attribute in the **ReplyTo** field of all outgoing requests. A service endpoint will use the value of the **jndiReplyDestinationName** attribute as the location for placing replies if there is no destination specified in the request's **ReplyTo** field.

Example

[Example 14.9, "JMS Consumer Specification Using a Named Reply Queue"](#) shows the configuration for a JMS client endpoint.

Example 14.9. JMS Consumer Specification Using a Named Reply Queue

```

<jms:conduit name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
    <jms:address destinationStyle="queue"
        jndiConnectionFactoryName="myConnectionFactory"
        jndiDestinationName="myDestination"
        jndiReplyDestinationName="myReplyDestination" >
        <jms:JMSNamingProperty name="java.naming.factory.initial"
            value="org.apache.cxf.transport.jms.MyInitialContextFactory" />
        <jms:JMSNamingProperty name="java.naming.provider.url"
            value="tcp://localhost:61616" />
    </jms:address>
</jms:conduit>
```

CHAPTER 15. INTEGRATING WITH APACHE ACTIVEMQ

OVERVIEW

If you are using Apache ActiveMQ as your JMS provider, the JNDI name of your destinations can be specified in a special format that dynamically creates JNDI bindings for queues or topics. This means that it is **not** necessary to configure the JMS provider in advance with the JNDI bindings for your queues or topics.

THE INITIAL CONTEXT FACTORY

The key to integrating Apache ActiveMQ with JNDI is the **ActiveMQInitialContextFactory** class. This class is used to create a JNDI **InitialContext** instance, which you can then use to access JMS destinations in the JMS broker.

[Example 15.1, "SOAP/JMS WSDL to connect to Apache ActiveMQ"](#) shows SOAP/JMS WSDL extensions to create a JNDI **InitialContext** that is integrated with Apache ActiveMQ.

Example 15.1. SOAP/JMS WSDL to connect to Apache ActiveMQ

```
<soapjms:jndiInitialContextFactory>
    org.apache.activemq.jndi.ActiveMQInitialContextFactory
</soapjms:jndiInitialContextFactory>
<soapjms:jndiURL>tcp://localhost:61616</soapjms:jndiURL>
```

In [Example 15.1, "SOAP/JMS WSDL to connect to Apache ActiveMQ"](#), the Apache ActiveMQ client connects to the broker port located at **tcp://localhost:61616**.

LOOKING UP THE CONNECTION FACTORY

As well as creating a JNDI **InitialContext** instance, you must specify the JNDI name that is bound to a **javax.jms.ConnectionFactory** instance. In the case of Apache ActiveMQ, there is a predefined binding in the **InitialContext** instance, which maps the JNDI name **ConnectionFactory** to an **ActiveMQConnectionFactory** instance. [Example 15.2, "SOAP/JMS WSDL for specifying the Apache ActiveMQ connection factory"](#) shows the SOAP/JMS extension element for specifying the Apache ActiveMQ connection factory.

Example 15.2. SOAP/JMS WSDL for specifying the Apache ActiveMQ connection factory

```
<soapjms:jndiConnectionFactoryName>
    ConnectionFactory
</soapjms:jndiConnectionFactoryName>
```

SYNTAX FOR DYNAMIC DESTINATIONS

To access queues or topics dynamically, specify the destination's JNDI name as a JNDI composite name in either of the following formats:

```
dynamicQueues/QueueName
dynamicTopics/TopicName
```

QueueName and *TopicName* are the names that the Apache ActiveMQ broker uses. They are **not** abstract JNDI names.

[Example 15.3, “WSDL port specification with a dynamically created queue”](#) shows a WSDL port that uses a dynamically created queue.

Example 15.3. WSDL port specification with a dynamically created queue

```
<service name="JMSService">
  <port binding="tns:GreeterBinding" name="JMSPort">
    <jms:address jndiConnectionFactory="ConnectionFactory"
      jndiDestinationName="dynamicQueues/greeter.request.queue" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
        value="org.apache.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
        value="tcp://localhost:61616" />
    </jms:address>
  </port>
</service>
```

When the application attempts to open the JMS connection, Apache ActiveMQ will check to see if a queue with the JNDI name **greeter.request.queue** exists. If it does not exist, it will create a new queue and bind it to the JNDI name **greeter.request.queue**.

CHAPTER 16. CONDUITS

Abstract

Conduits are a low-level piece of the transport architecture that are used to implement outbound connections. Their behavior and life-cycle can effect system performance and processing load.

OVERVIEW

Conduits manage the client-side, or outbound, transport details in the Apache CXF runtime. They are responsible for opening ports, establishing outbound connections, sending messages, and listening for any responses between an application and a single external endpoint. If an application connects to multiple endpoints, it will have one conduit instance for each endpoint.

Each transport type implements its own conduit using the Conduit interface. This allows for a standardized interface between the application level functionality and the transports.

In general, you only need to worry about the conduits being used by your application when configuring the client-side transport details. The underlying semantics of how the runtime handles conduits is, generally, not something a developer needs to worry about.

However, there are cases when an understanding of conduit's can prove helpful:

- Implementing a custom transport
- Advanced application tuning to manage limited resources

CONDUIT LIFE-CYCLE

Conduits are managed by the client implementation object. Once created, a conduit lives for the duration of the client implementation object. The conduit's life-cycle is:

1. When the client implementation object is created, it is given a reference to a **ConduitSelector** object.
2. When the client needs to send a message is request's a reference to a conduit from the conduit selector.
If the message is for a new endpoint, the conduit selector creates a new conduit and passes it to the client implementation. Otherwise, it passes the client a reference to the conduit for the target endpoint.
3. The conduit sends messages when needed.
4. When the client implementation object is destroyed, all of the conduits associated with it are destroyed.

CONDUIT WEIGHT

The weight of a conduit object depends on the transport implementation. HTTP conduits are extremely light weight. JMS conduits are heavy because they are associated with the JMS **Session** object and one or more **JMSListenerContainer** objects.

PART IV. CONFIGURING WEB SERVICE ENDPOINTS

This guide describes how to create Apache CXF endpoints in Red Hat Fuse.

CHAPTER 17. CONFIGURING JAX-WS ENDPOINTS

Abstract

JAX-WS endpoints are configured using one of three Spring configuration elements. The correct element depends on what type of endpoint you are configuring and which features you wish to use. For consumers you use the **jaxws:client** element. For service providers you can use either the **jaxws:endpoint** element or the **jaxws:server** element.

The information used to define an endpoint is typically defined in the endpoint's contract. You can use the configuration element's to override the information in the contract. You can also use the configuration elements to provide information that is not provided in the contract.

You must use the configuration elements to activate advanced features such as WS-RM. This is done by providing child elements to the endpoint's configuration element. Note that when dealing with endpoints developed using a Java-first approach it is likely that the SEL serving as the endpoint's contract is lacking information about the type of binding and transport to use.

17.1. CONFIGURING SERVICE PROVIDERS

17.1.1. Elements for Configuring Service Providers

Apache CXF has two elements that can be used to configure a service provider:

- [Section 17.1.2, "Using the jaxws:endpoint Element"](#)
- [Section 17.1.3, "Using the jaxws:server Element"](#)

The differences between the two elements are largely internal to the runtime. The **jaxws:endpoint** element injects properties into the **org.apache.cxf.jaxws.EndpointImpl** object created to support a service endpoint. The **jaxws:server** element injects properties into the **org.apache.cxf.jaxws.support.JaxWsServerFactoryBean** object created to support the endpoint. The **EndpointImpl** object passes the configuration data to the **JaxWsServerFactoryBean** object. The **JaxWsServerFactoryBean** object is used to create the actual service object. Because either configuration element will configure a service endpoint, you can choose based on the syntax you prefer.

17.1.2. Using the jaxws:endpoint Element

Overview

The **jaxws:endpoint** element is the default element for configuring JAX-WS service providers. Its attributes and children specify all of the information needed to instantiate a service provider. Many of the attributes map to information in the service's contract. The children are used to configure interceptors and other advanced features.

Identifying the endpoint being configured

For the runtime to apply the configuration to the proper service provider, it must be able to identify it. The basic means for identifying a service provider is to specify the class that implements the endpoint. This is done using the **jaxws:endpoint** element's **implementor** attribute.

For instances where different endpoint's share a common implementation, it is possible to provide different configuration for each endpoint. There are two approaches for distinguishing a specific endpoint in configuration:

- a combination of the **serviceName** attribute and the **endpointName** attribute
The **serviceName** attribute specifies the **wsdl:service** element defining the service's endpoint. The **endpointName** attribute specifies the specific **wsdl:port** element defining the service's endpoint. Both attributes are specified as QNames using the format **ns:name**. *ns* is the namespace of the element and *name* is the value of the element's **name** attribute.



NOTE

If the **wsdl:service** element only has one **wsdl:port** element, the **endpointName** attribute can be omitted.

- the **name** attribute

The **name** attribute specifies the QName of the specific **wsdl:port** element defining the service's endpoint. The QName is provided in the format **{ns}localPart**. *ns* is the namespace of the **wsdl:port** element and *localPart* is the value of the **wsdl:port** element's **name** attribute.

Attributes

The attributes of the **jaxws:endpoint** element configure the basic properties of the endpoint. These properties include the address of the endpoint, the class that implements the endpoint, and the **bus** that hosts the endpoint.

[Table 17.1, "Attributes for Configuring a JAX-WS Service Provider Using the jaxws:endpoint Element"](#) describes the attribute of the **jaxws:endpoint** element.

Table 17.1. Attributes for Configuring a JAX-WS Service Provider Using the jaxws:endpoint Element

Attribute	Description
id	Specifies a unique identifier that other configuration elements can use to refer to the endpoint.
implementor	Specifies the class implementing the service. You can specify the implementation class using either the class name or an ID reference to a Spring bean configuring the implementation class. This class must be on the classpath.
implementorClass	Specifies the class implementing the service. This attribute is useful when the value provided to the implementor attribute is a reference to a bean that is wrapped using Spring AOP.
address	Specifies the address of an HTTP endpoint. This value overrides the value specified in the services contract.

Attribute	Description
wsdlLocation	Specifies the location of the endpoint's WSDL contract. The WSDL contract's location is relative to the folder from which the service is deployed.
endpointName	Specifies the value of the service's wsdl:port element's name attribute. It is specified as a QName using the format ns:name where ns is the namespace of the wsdl:port element.
serviceName	Specifies the value of the service's wsdl:service element's name attribute. It is specified as a QName using the format ns:name where ns is the namespace of the wsdl:service element.
publish	Specifies if the service should be automatically published. If this is set to false , the developer must explicitly publish the endpoints described in Chapter 31, Publishing a Service .
bus	Specifies the ID of the Spring bean configuring the bus used to manage the service endpoint. This is useful when configuring several endpoints to use a common set of features.
bindingUri	Specifies the ID of the message binding the service uses. A list of valid binding IDs is provided in Chapter 23, Apache CXF Binding IDs .
name	Specifies the stringified QName of the service's wsdl:port element. It is specified as a QName using the format {ns}localPart . ns is the namespace of the wsdl:port element and localPart is the value of the wsdl:port element's name attribute.
abstract	Specifies if the bean is an abstract bean. Abstract beans act as parents for concrete bean definitions and are not instantiated. The default is false . Setting this to true instructs the bean factory not to instantiate the bean.
depends-on	Specifies a list of beans that the endpoint depends on being instantiated before it can be instantiated.

Attribute	Description
createdFromAPI	<p>Specifies that the user created that bean using Apache CXF APIs, such as Endpoint.publish() or Service.getPort().</p> <p>The default is false.</p> <p>Setting this to true does the following:</p> <ul style="list-style-type: none"> Changes the internal name of the bean by appending .jaxws-endpoint to its id Makes the bean abstract
publishedEndpointUrl	<p>The URL that is placed in the address element of the generated WSDL. If this value is not specified, the value of the address attribute is used. This attribute is useful when the "public" URL is not be the same as the URL on which the service is deployed.</p>

In addition to the attributes listed in [Table 17.1, "Attributes for Configuring a JAX-WS Service Provider Using the jaxws:endpoint Element"](#), you might need to use multiple **xmlns:shortName** attributes to declare the namespaces used by the **endpointName** and **serviceName** attributes.

Example

[Example 17.1, "Simple JAX-WS Endpoint Configuration"](#) shows the configuration for a JAX-WS endpoint that specifies the address where the endpoint is published. The example assumes that you want to use the defaults for all other values or that the implementation has specified values in the annotations.

Example 17.1. Simple JAX-WS Endpoint Configuration

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
  ...">
<jaxws:endpoint id="example"
  implementor="org.apache.cxf.example.DemoImpl"
  address="http://localhost:8080/demo" />
</beans>
```

[Example 17.2, "JAX-WS Endpoint Configuration with a Service Name"](#) shows the configuration for a JAX-WS endpoint whose contract contains two service definitions. In this case, you must specify which service definition to instantiate using the **serviceName** attribute.

Example 17.2. JAX-WS Endpoint Configuration with a Service Name

```
<beans ...  
    xmlns:jaxws="http://cxf.apache.org/jaxws"  
    ...  
    schemaLocation="...  
        http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd  
    ...">  
  
<jaxws:endpoint id="example2"  
    implementor="org.apache.cxf.example.DemoImpl"  
    serviceName="samp:demoService2"  
    xmlns:samp="http://org.apache.cxf/wsdl/example" />  
  
</beans>
```

The **xmlns:samp** attribute specifies the namespace in which the WSDL **service** element is defined.

17.1.3. Using the jaxws:server Element

Overview

The **jaxws:server** element is an element for configuring JAX-WS service providers. It injects the configuration information into the **org.apache.cxf.jaxws.support.JaxWsServerFactoryBean**. This is a Apache CXF specific object. If you are using a pure Spring approach to building your services, you will not be forced to use Apache CXF specific APIs to interact with the service.

The attributes and children of the **jaxws:server** element specify all of the information needed to instantiate a service provider. The attributes specify the information that is required to instantiate an endpoint. The children are used to configure interceptors and other advanced features.

Identifying the endpoint being configured

In order for the runtime to apply the configuration to the proper service provider, it must be able to identify it. The basic means for identifying a service provider is to specify the class that implements the endpoint. This is done using the **jaxws:server** element's **serviceBean** attribute.

For instances where different endpoint's share a common implementation, it is possible to provide different configuration for each endpoint. There are two approaches for distinguishing a specific endpoint in configuration:

- a combination of the **serviceName** attribute and the **endpointName** attribute
The **serviceName** attribute specifies the **wsdl:service** element defining the service's endpoint.
The **endpointName** attribute specifies the specific **wsdl:port** element defining the service's endpoint. Both attributes are specified as QNames using the format ***ns:name***. *ns* is the namespace of the element and *name* is the value of the element's **name** attribute.



NOTE

If the **wsdl:service** element only has one **wsdl:port** element, the **endpointName** attribute can be omitted.

- the **name** attribute

The **name** attribute specifies the QName of the specific **wsdl:port** element defining the service's endpoint. The QName is provided in the format **{ns}localPart**. *ns* is the namespace of the **wsdl:port** element and *localPart* is the value of the **wsdl:port** element's **name** attribute.

Attributes

The attributes of the **jaxws:server** element configure the basic properties of the endpoint. These properties include the address of the endpoint, the class that implements the endpoint, and the **bus** that hosts the endpoint.

[Table 17.2, "Attributes for Configuring a JAX-WS Service Provider Using the jaxws:server Element"](#) describes the attribute of the **jaxws:server** element.

Table 17.2. Attributes for Configuring a JAX-WS Service Provider Using the jaxws:server Element

Attribute	Description
id	Specifies a unique identifier that other configuration elements can use to refer to the endpoint.
serviceBean	Specifies the class implementing the service. You can specify the implementation class using either the class name or an ID reference to a Spring bean configuring the implementation class. This class must be on the classpath.
serviceClass	Specifies the class implementing the service. This attribute is useful when the value provided to the implementor attribute is a reference to a bean that is wrapped using Spring AOP.
address	Specifies the address of an HTTP endpoint. This value will override the value specified in the services contract.
wsdlLocation	Specifies the location of the endpoint's WSDL contract. The WSDL contract's location is relative to the folder from which the service is deployed.
endpointName	Specifies the value of the service's wsdl:port element's name attribute. It is specified as a QName using the format ns:name , where <i>ns</i> is the namespace of the wsdl:port element.
serviceName	Specifies the value of the service's wsdl:service element's name attribute. It is specified as a QName using the format ns:name , where <i>ns</i> is the namespace of the wsdl:service element.
publish	Specifies if the service should be automatically published. If this is set to false , the developer must explicitly publish the endpoints described in Chapter 31, Publishing a Service .

Attribute	Description
bus	Specifies the ID of the Spring bean configuring the bus used to manage the service endpoint. This is useful when configuring several endpoints to use a common set of features.
bindingId	Specifies the ID of the message binding the service uses. A list of valid binding IDs is provided in Chapter 23, Apache CXF Binding IDs .
name	Specifies the stringified QName of the service's wsdl:port element. It is specified as a QName using the format {ns}localPart , where <i>ns</i> is the namespace of the wsdl:port element and <i>localPart</i> is the value of the wsdl:port element's name attribute.
abstract	Specifies if the bean is an abstract bean. Abstract beans act as parents for concrete bean definitions and are not instantiated. The default is false . Setting this to true instructs the bean factory not to instantiate the bean.
depends-on	Specifies a list of beans that the endpoint depends on being instantiated before the endpoint can be instantiated.
createdFromAPI	<p>Specifies that the user created that bean using Apache CXF APIs, such as Endpoint.publish() or Service.getPort().</p> <p>The default is false.</p> <p>Setting this to true does the following:</p> <ul style="list-style-type: none"> ● Changes the internal name of the bean by appending .jaxws-endpoint to its id ● Makes the bean abstract

In addition to the attributes listed in [Table 17.2, “Attributes for Configuring a JAX-WS Service Provider Using the `jaxws:server` Element”](#), you might need to use multiple **xmlns:shortName** attributes to declare the namespaces used by the **endpointName** and **serviceName** attributes.

Example

[Example 17.3, “Simple JAX-WS Server Configuration”](#) shows the configuration for a JAX-WS endpoint that specifies the address where the endpoint is published.

Example 17.3. Simple JAX-WS Server Configuration

```

<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
  ...">
  <jaxws:server id="exampleServer"
    serviceBean="org.apache.cxf.example.DemoImpl"
    address="http://localhost:8080/demo" />
</beans>

```

17.1.4. Adding Functionality to Service Providers

Overview

The **jaxws:endpoint** and the **jaxws:server** elements provide the basic configuration information needed to instantiate a service provider. To add functionality to your service provider or to perform advanced configuration you must add child elements to the configuration.

Child elements allow you to do the following:

- [Chapter 19, Apache CXF Logging](#)
- [Chapter 59, Configuring Endpoints to Use Interceptors](#)
- [Chapter 20, Deploying WS-Addressing](#)
- [Chapter 21, Enabling Reliable Messaging](#)
- [Section 17.1.5, "Enable Schema Validation on a JAX-WS Endpoint"](#)

Elements

[Table 17.3, "Elements Used to Configure JAX-WS Service Providers"](#) describes the child elements that **jaxws:endpoint** supports.

Table 17.3. Elements Used to Configure JAX-WS Service Providers

Element	Description
jaxws:handlers	Specifies a list of JAX-WS Handler implementations for processing messages. For more information on JAX-WS Handler implementations see Chapter 43, Writing Handlers .
jaxws:inInterceptors	Specifies a list of interceptors that process inbound requests. For more information see Part VII, "Developing Apache CXF Interceptors" .
jaxws:inFaultInterceptors	Specifies a list of interceptors that process inbound fault messages. For more information see Part VII, "Developing Apache CXF Interceptors" .

Element	Description
jaxws:outInterceptors	Specifies a list of interceptors that process outbound replies. For more information see Part VII, "Developing Apache CXF Interceptors" .
jaxws:outFaultInterceptors	Specifies a list of interceptors that process outbound fault messages. For more information see Part VII, "Developing Apache CXF Interceptors" .
jaxws:binding	Specifies a bean configuring the message binding used by the endpoint. Message bindings are configured using implementations of the org.apache.cxf.binding.BindingFactory interface. ^[a]
jaxws:dataBinding ^[b]	Specifies the class implementing the data binding used by the endpoint. This is specified using an embedded bean definition.
jaxws:executor	Specifies a Java executor that is used for the service. This is specified using an embedded bean definition.
jaxws:features	Specifies a list of beans that configure advanced features of Apache CXF. You can provide either a list of bean references or a list of embedded beans.
jaxws:invoker	Specifies an implementation of the org.apache.cxf.service.Invoker interface used by the service. ^[c]
jaxws:properties	Specifies a Spring map of properties that are passed along to the endpoint. These properties can be used to control features like enabling MTOM support.
jaxws:serviceFactory	Specifies a bean configuring the JaxWsServiceFactoryBean object used to instantiate the service.

[a] The SOAP binding is configured using the **soap:soapBinding** bean.

[b] The **jaxws:endpoint** element does not support the **jaxws:dataBinding** element.

[c] The Invoker implementation controls how a service is invoked. For example, it controls whether each request is handled by a new instance of the service implementation or if state is preserved across invocations.

17.1.5. Enable Schema Validation on a JAX-WS Endpoint

Overview

You can set the **schema-validation-enabled** property to enable schema validation on a **jaxws:endpoint** element or on a **jaxws:server** element. When schema validation is enabled, the messages sent between client and server are checked for conformity to the schema. By default, schema validation is turned off, because it has a significant impact on performance.

Example

To enable schema validation on a JAX-WS endpoint, set the **schema-validation-enabled** property in the **jaxws:properties** child element of the **jaxws:endpoint** element or of the **jaxws:server** element. For example, to enable schema validation on a **jaxws:endpoint** element:

```
<jaxws:endpoint name="{http://apache.org/hello_world_soap_http}SoapPort"
    wsdlLocation="wsdl/hello_world.wsdl"
    createdFromAPI="true">
    <jaxws:properties>
        <entry key="schema-validation-enabled" value="BOTH" />
    </jaxws:properties>
</jaxws:endpoint>
```

For the list of allowed values of the **schema-validation-enabled** property, see [Section 24.3.4.7, "Schema Validation Type Values"](#).

17.2. CONFIGURING CONSUMER ENDPOINTS

Overview

JAX-WS consumer endpoints are configured using the **jaxws:client** element. The element's attributes provide the basic information necessary to create a consumer.

To add other functionality, like WS-RM, to the consumer you add children to the **jaxws:client** element. Child elements are also used to configure the endpoint's logging behavior and to inject other properties into the endpoint's implementation.

Basic Configuration Properties

The attributes described in [Table 17.4, "Attributes Used to Configure a JAX-WS Consumer"](#) provide the basic information necessary to configure a JAX-WS consumer. You only need to provide values for the specific properties you want to configure. Most of the properties have sensible defaults, or they rely on information provided by the endpoint's contract.

Table 17.4. Attributes Used to Configure a JAX-WS Consumer

Attribute	Description
address	Specifies the HTTP address of the endpoint where the consumer will make requests. This value overrides the value set in the contract.
bindingId	Specifies the ID of the message binding the consumer uses. A list of valid binding IDs is provided in Chapter 23, Apache CXF Binding IDs .

Attribute	Description
bus	Specifies the ID of the Spring bean configuring the bus managing the endpoint.
endpointName	Specifies the value of the wsdl:port element's name attribute for the service on which the consumer is making requests. It is specified as a QName using the format ns:name , where <i>ns</i> is the namespace of the wsdl:port element.
serviceName	Specifies the value of the wsdl:service element's name attribute for the service on which the consumer is making requests. It is specified as a QName using the format ns:name where <i>ns</i> is the namespace of the wsdl:service element.
username	Specifies the username used for simple username/password authentication.
password	Specifies the password used for simple username/password authentication.
serviceClass	Specifies the name of the service endpoint interface(SEI).
wsdlLocation	Specifies the location of the endpoint's WSDL contract. The WSDL contract's location is relative to the folder from which the client is deployed.
name	Specifies the stringified QName of the wsdl:port element for the service on which the consumer is making requests. It is specified as a QName using the format {ns}localPart , where <i>ns</i> is the namespace of the wsdl:port element and <i>localPart</i> is the value of the wsdl:port element's name attribute.
abstract	Specifies if the bean is an abstract bean. Abstract beans act as parents for concrete bean definitions and are not instantiated. The default is false . Setting this to true instructs the bean factory not to instantiate the bean.
depends-on	Specifies a list of beans that the endpoint depends on being instantiated before it can be instantiated.

Attribute	Description
createdFromAPI	<p>Specifies that the user created that bean using Apache CXF APIs like Service.getPort().</p> <p>The default is false.</p> <p>Setting this to true does the following:</p> <ul style="list-style-type: none"> Changes the internal name of the bean by appending .jaxws-client to its id Makes the bean abstract

In addition to the attributes listed in [Table 17.4, “Attributes Used to Configure a JAX-WS Consumer”](#), it might be necessary to use multiple **xmlns:shortName** attributes to declare the namespaces used by the **endpointName** and the **serviceName** attributes.

Adding functionality

To add functionality to your consumer or to perform advanced configuration, you must add child elements to the configuration.

Child elements allow you to do the following:

- [Chapter 19, Apache CXF Logging](#)
- [Chapter 59, Configuring Endpoints to Use Interceptors](#)
- [Chapter 20, Deploying WS-Addressing](#)
- [Chapter 21, Enabling Reliable Messaging](#)
- the section called “Enable schema validation on a JAX-WS consumer”

[Table 17.5, “Elements For Configuring a Consumer Endpoint”](#) describes the child element’s you can use to configure a JAX-WS consumer.

Table 17.5. Elements For Configuring a Consumer Endpoint

Element	Description
jaxws:binding	Specifies a bean configuring the message binding used by the endpoint. Message bindings are configured using implementations of the org.apache.cxf.binding.BindingFactory interface. ^[a]
jaxws:dataBinding	Specifies the class implementing the data binding used by the endpoint. You specify this using an embedded bean definition. The class implementing the JAXB data binding is org.apache.cxf.jaxb.JAXBDataBinding .

Element	Description
jaxws:features	Specifies a list of beans that configure advanced features of Apache CXF. You can provide either a list of bean references or a list of embedded beans.
jaxws:handlers	Specifies a list of JAX-WS Handler implementations for processing messages. For more information in JAX-WS Handler implementations see Chapter 43, Writing Handlers .
jaxws:inInterceptors	Specifies a list of interceptors that process inbound responses. For more information see Part VII, "Developing Apache CXF Interceptors" .
jaxws:inFaultInterceptors	Specifies a list of interceptors that process inbound fault messages. For more information see Part VII, "Developing Apache CXF Interceptors" .
jaxws:outInterceptors	Specifies a list of interceptors that process outbound requests. For more information see Part VII, "Developing Apache CXF Interceptors" .
jaxws:outFaultInterceptors	Specifies a list of interceptors that process outbound fault messages. For more information see Part VII, "Developing Apache CXF Interceptors" .
jaxws:properties	Specifies a map of properties that are passed to the endpoint.
jaxws:conduitSelector	Specifies an org.apache.cxf.endpoint.ConduitSelector implementation for the client to use. A ConduitSelector implementation will override the default process used to select the Conduit object that is used to process outbound requests.

[a] The SOAP binding is configured using the **soap:soapBinding** bean.

Example

[Example 17.4, "Simple Consumer Configuration"](#) shows a simple consumer configuration.

Example 17.4. Simple Consumer Configuration

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
```

```
...">
<jaxws:client id="bookClient"
    serviceClass="org.apache.cxf.demo.BookClientImpl"
    address="http://localhost:8080/books"/>
...
</beans>
```

Enable schema validation on a JAX-WS consumer

To enable schema validation on a JAX-WS consumer, set the **schema-validation-enabled** property in the **jaxws:properties** child element of the **jaxws:client** element—for example:

```
<jaxws:client name="{http://apache.org/hello_world_soap_http}SoapPort"
    createdFromAPI="true">
    <jaxws:properties>
        <entry key="schema-validation-enabled" value="BOTH" />
    </jaxws:properties>
</jaxws:client>
```

For the list of allowed values of the **schema-validation-enabled** property, see [Section 24.3.4.7, "Schema Validation Type Values"](#).

CHAPTER 18. CONFIGURING JAX-RS ENDPOINTS

Abstract

This chapter explains how to instantiate and configure JAX-RS server endpoints in Blueprint XML and in Spring XML, and also how to instantiate and configure JAX-RS client endpoints (client proxy beans) in XML.

18.1. CONFIGURING JAX-RS SERVER ENDPOINTS

18.1.1. Defining a JAX-RS Server Endpoint

Basic server endpoint definition

To define a JAX-RS server endpoint in XML, you need to specify at least the following:

1. A **jaxrs:server** element, which is used to define the endpoint in XML. Note that the **jaxrs:** namespace prefix maps to **different** namespaces in Blueprint and in Spring respectively.
2. The base URL of the JAX-RS service, using the **address** attribute of the **jaxrs:server** element. Note that there are two different ways of specifying the address URL, which affects how the endpoint gets deployed:
 - As a **relative URL**—for example, **/customers**. In this case, the endpoint is deployed into the default HTTP container, and the endpoint's base URL is implicitly obtained by combining the CXF servlet base URL with the specified relative URL.
For example, if you deploy a JAX-RS endpoint to the Fuse container, the specified **/customers** URL would get resolved to the URL, **http://Hostname:8181/cxf/customers** (assuming that the container is using the default **8181** port).
 - As an **absolute URL**—for example, **http://0.0.0.0:8200/cxf/customers**. In this case, a new HTTP listener port is opened for the JAX-RS endpoint (if it is not already open). For example, in the context of Fuse, a new Undertow container would implicitly be created to host the JAX-RS endpoint. The special IP address, **0.0.0.0**, acts as a wildcard, matching any of the hostnames assigned to the current host (which can be useful on multi-homed host machines).
3. One or more JAX-RS root resource classes, which provide the implementation of the JAX-RS service. The simplest way to specify the resource classes is to list them inside a **jaxrs:serviceBeans** element.

Blueprint example

The following Blueprint XML example shows how to define a JAX-RS endpoint, which specifies the relative address, **/customers** (so that it deploys into the default HTTP container) and is implemented by the **service.CustomerService** resource class:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
  xmlns:cxf="http://cxf.apache.org/blueprint/core"
  xsi:schemaLocation="
    http://www.osgi.org/xmlns/blueprint/v1.0.0 https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
    http://cxf.apache.org/blueprint/jaxrs http://cxf.apache.org/schemas/blueprint/jaxrs.xsd
  ">

  <jaxrs:server>
    <jaxrs:address>/customers</jaxrs:address>
    <jaxrs:resourceClasses>
      <jaxrs:resourceClass>service.CustomerService</jaxrs:resourceClass>
    </jaxrs:resourceClasses>
  </jaxrs:server>

```

```

http://cxf.apache.org/blueprint/core http://cxf.apache.org/schemas/blueprint/core.xsd
">

<cxft:bus>
  <cxft:features>
    <cxft:logging/>
  </cxft:features>
</cxft:bus>

<jaxrs:server id="customerService" address="/customers">
  <jaxrs:serviceBeans>
    <ref component-id="serviceBean" />
  </jaxrs:serviceBeans>
</jaxrs:server>

<bean id="serviceBean" class="service.CustomerService"/>
</blueprint>

```

Blueprint XML namespaces

To define a JAX-RS endpoint in Blueprint, you typically require at least the following XML namespaces:

Prefix	Namespace
(default)	http://www.osgi.org/xmlns/blueprint/v1.0.0
cxf	http://cxf.apache.org/blueprint/core
jaxrs	http://cxf.apache.org/blueprint/jaxrs

Spring example

The following Spring XML example shows how to define a JAX-RS endpoint, which specifies the relative address, **/customers** (so that it deploys into the default HTTP container) and is implemented by the **service.CustomerService** resource class:

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxrs="http://cxf.apache.org/jaxrs"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
           http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd">

  <jaxrs:server id="customerService" address="/customers">
    <jaxrs:serviceBeans>
      <ref bean="serviceBean"/>
    </jaxrs:serviceBeans>
  </jaxrs:server>

  <bean id="serviceBean" class="service.CustomerService"/>
</beans>

```

Spring XML namespaces

To define a JAX-RS endpoint in Spring, you typically require at least the following XML namespaces:

Prefix	Namespace
(default)	http://www.springframework.org/schema/beans
cxf	http://cxf.apache.org/core
jaxrs	http://cxf.apache.org/jaxrs

Auto-discovery in Spring XML

(Spring only) Instead of specifying the JAX-RS root resource classes explicitly, Spring XML enables you to configure auto-discovery, so that specific Java packages are searched for resource classes (classes annotated by `@Path`) and all of the discovered resource classes are automatically attached to the endpoint. In this case, you need to specify just the **address** attribute and the **basePackages** attribute in the **jaxrs:server** element.

For example, to define a JAX-RS endpoint which uses all of the JAX-RS resource classes under the **a.b.c** Java package, you can define the endpoint in Spring XML, as follows:

```
<jaxrs:server address="/customers" basePackages="a.b.c"/>
```

The auto-discovery mechanism also discovers and installs into the endpoint any JAX-RS provider classes that it finds under the specified Java packages.

Lifecycle management in Spring XML

(Spring only) Spring XML enables you to control the lifecycle of beans by setting the **scope** attribute on a **bean** element. The following scope values are supported by Spring:

singleton

(Default) Creates a single bean instance, which is used everywhere and lasts for the entire lifetime of the Spring container.

prototype

Creates a new bean instance every time the bean is injected into another bean or when a bean is obtained by invoking **getBean()** on the bean registry.

request

(Only available in a Web-aware container)Creates a new bean instance for every request invoked on the bean.

session

(Only available in a Web-aware container)Creates a new bean for the lifetime of a single HTTP session.

globalSession

(Only available in a Web-aware container)Creates a new bean for the lifetime of a single HTTP session that is shared between portlets.

For more details about Spring scopes, please consult the Spring framework documentation on [Bean scopes](#).

Note that Spring scopes **do not work properly**, if you specify JAX-RS resource beans through the **jaxrs:serviceBeans** element. If you specify the **scope** attribute on the resource beans in this case, the **scope** attribute is effectively ignored.

In order to make bean scopes work properly within a JAX-RS server endpoint, you require a level of indirection that is provided by a service factory. The simplest way to configure bean scopes is to specify resource beans using the **beanNames** attribute on the **jaxrs:server** element, as follows:

```
<beans ... >
<jaxrs:server id="customerService" address="/service1"
  beanNames="customerBean1 customerBean2">

  <bean id="customerBean1" class="demo.jaxrs.server.CustomerRootResource1"
    scope="prototype"/>
  <bean id="customerBean2" class="demo.jaxrs.server.CustomerRootResource2"
    scope="prototype"/>
</beans>
```

Where the preceding example configures two resource beans, **customerBean1** and **customerBean2**. The **beanNames** attribute is specified as a space-separated list of resource bean IDs.

For the ultimate degree of flexibility, you have the option of defining service factory objects **explicitly**, when you configure the JAX-RS server endpoint, using the **jaxrs:serviceFactories** element. This more verbose approach has the advantage that you can replace the default service factory implementation with your custom implementation, thus giving you ultimate control over the bean lifecycle. The following example shows how to configure the two resource beans, **customerBean1** and **customerBean2**, using this approach:

```
<beans ... >
<jaxrs:server id="customerService" address="/service1">
  <jaxrs:serviceFactories>
    <ref bean="sfactory1" />
    <ref bean="sfactory2" />
  </jaxrs:serviceFactories>
</jaxrs:server>

<bean id="sfactory1" class="org.apache.cxf.jaxrs.spring.SpringResourceFactory">
  <property name="beanId" value="customerBean1"/>
</bean>
<bean id="sfactory2" class="org.apache.cxf.jaxrs.spring.SpringResourceFactory">
  <property name="beanId" value="customerBean2"/>
</bean>

<bean id="customerBean1" class="demo.jaxrs.server.CustomerRootResource1"
  scope="prototype"/>
<bean id="customerBean2" class="demo.jaxrs.server.CustomerRootResource2"
  scope="prototype"/>
</beans>
```



NOTE

If you specify a non-singleton lifecycle, it is often a good idea to implement and register a `org.apache.cxf.service.Invoker` bean (where the instance can be registered by referencing it from a **jaxrs:server/jaxrs:invoker** element).

Attaching a WADL document

You can optionally associate a WADL document with the JAX-RS server endpoint using the **docLocation** attribute on the **jaxrs:server** element. For example:

```
<jaxrs:server address="/rest" docLocation="wadl/bookStore.wadl">
  <jaxrs:serviceBeans>
    <bean class="org.bar.generated.BookStore"/>
  </jaxrs:serviceBeans>
</jaxrs:server>
```

Schema validation

If you have some external XML schemas, for describing message content in JAX-B format, you can associate these external schemas with the JAX-RS server endpoint through the **jaxrs:schemaLocations** element.

For example, if you have associated the server endpoint with a WADL document and you also want to enable schema validation on incoming messages, you can specify associated XML schema files as follows:

```
<jaxrs:server address="/rest"
  docLocation="wadl/bookStore.wadl">
  <jaxrs:serviceBeans>
    <bean class="org.bar.generated.BookStore"/>
  </jaxrs:serviceBeans>
  <jaxrs:schemaLocations>
    <jaxrs:schemaLocation>classpath:/schemas/a.xsd</jaxrs:schemaLocation>
    <jaxrs:schemaLocation>classpath:/schemas/b.xsd</jaxrs:schemaLocation>
  </jaxrs:schemaLocations>
</jaxrs:server>
```

Alternatively, if you want to include all of the schema files, ***.xsd**, in a given directory, you can just specify the directory name, as follows:

```
<jaxrs:server address="/rest"
  docLocation="wadl/bookStore.wadl">
  <jaxrs:serviceBeans>
    <bean class="org.bar.generated.BookStore"/>
  </jaxrs:serviceBeans>
  <jaxrs:schemaLocations>
    <jaxrs:schemaLocation>classpath:/schemas/</jaxrs:schemaLocation>
  </jaxrs:schemaLocations>
</jaxrs:server>
```

Specifying schemas in this way is generally useful for any kind of functionality that requires access to the JAX-B schemas.

Specifying the data binding

You can use the **jaxrs:dataBinding** element to specify the data binding that encodes the message body in request and reply messages. For example, to specify the JAX-B data binding, you could configure a JAX-RS endpoint as follows:

```
<jaxrs:server id="jaxbbook" address="/jaxb">
  <jaxrs:serviceBeans>
    <ref bean="serviceBean" />
  </jaxrs:serviceBeans>
  <jaxrs:dataBinding>
    <bean class="org.apache.cxf.jaxb.JAXBDataBinding"/>
  </jaxrs:dataBinding>
</jaxrs:server>
```

Or to specify the Aegis data binding, you could configure a JAX-RS endpoint as follows:

```
<jaxrs:server id="aegisbook" address="/aegis">
  <jaxrs:serviceBeans>
    <ref bean="serviceBean" />
  </jaxrs:serviceBeans>
  <jaxrs:dataBinding>
    <bean class="org.apache.cxf.aegis.databinding.AegisDatabinding">
      <property name="aegisContext">
        <bean class="org.apache.cxf.aegis.AegisContext">
          <property name="writeXsiTypes" value="true"/>
        </bean>
      </property>
    </bean>
  </jaxrs:dataBinding>
</jaxrs:server>
```

Using the JMS transport

It is possible to configure JAX-RS to use a JMS messaging library as a transport protocol, instead of HTTP. Because JMS itself is **not** a transport protocol, the actual messaging protocol depends on the particular JMS implementation that you configure.

For example, the following Spring XML example shows how to configure a JAX-RS server endpoint to use the JMS transport protocol:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jms="http://cxf.apache.org/transports/jms"
       xmlns:jaxrs="http://cxf.apache.org/jaxrs"
       xsi:schemaLocation="
         http://cxf.apache.org/transports/jms http://cxf.apache.org/schemas/configuration/jms.xsd
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd">

  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>
  <bean id="ConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL"
```

```

        value="tcp://localhost:${testutil.ports.EmbeddedJMSBrokerLauncher}" />
    </bean>

    <jaxrs:server xmlns:s="http://books.com"
        serviceName="s:BookService"
        transportId= "http://cf.apache.org/transports/jms"
        address="jms:queue:test.jmstransport.text?replyToName=test.jmstransport.response">
        <jaxrs:serviceBeans>
            <bean class="org.apache.cxf.systest.jaxrs.JMSBookStore"/>
        </jaxrs:serviceBeans>
    </jaxrs:server>

</beans>

```

Note the following points about the preceding example:

- **JMS implementation**—the JMS implementation is provided by the **ConnectionFactory** bean, which instantiates an Apache ActiveMQ connection factory object. After you instantiate the connection factory, it is automatically installed as the default JMS implementation layer.
- **JMS conduit or destination object**—Apache CXF implicitly instantiates a JMS conduit object (to represent a JMS consumer) or a JMS destination object (to represent a JMS provider). This object must be uniquely identified by a QName, which is defined through the attribute settings **xmlns:s="http://books.com"** (defining the namespace prefix) and **serviceName="s:BookService"** (defining the QName).
- **Transport ID**—to select the JMS transport, the **transportId** attribute must be set to <http://cf.apache.org/transports/jms>.
- **JMS address**—the **jaxrs:server/@address** attribute uses a standardized syntax to specify the JMS queue or JMS topic to send to. For details of this syntax, see <https://tools.ietf.org/id/draft-merrick-jms-uri-06.txt>.

Extension mappings and language mappings

A JAX-RS server endpoint can be configured so that it automatically maps a file suffix (appearing in the URL) to a MIME content type header, and maps a language suffix to a language type header. For example, consider a HTTP request of the following form:

```
GET /resource.xml
```

You can configure the JAX-RS server endpoint to map the **.xml** suffix automatically, as follows:

```

<jaxrs:server id="customerService" address="/">
    <jaxrs:serviceBeans>
        <bean class="org.apache.cxf.jaxrs.systests.CustomerService" />
    </jaxrs:serviceBeans>
    <jaxrs:extensionMappings>
        <entry key="json" value="application/json"/>
        <entry key="xml" value="application/xml"/>
    </jaxrs:extensionMappings>
</jaxrs:server>

```

When the preceding server endpoint receives the HTTP request, it automatically creates a new content type header of type, **application/xml**, and strips the **.xml** suffix from the resource URL.

For the language mapping, consider a HTTP request of the following form:

GET /resource.en

You can configure the JAX-RS server endpoint to map the **.en** suffix automatically, as follows:

```
<jaxrs:server id="customerService" address="/">
  <jaxrs:serviceBeans>
    <bean class="org.apache.cxf.jaxrs.systests.CustomerService" />
  </jaxrs:serviceBeans>
  <jaxrs:languageMappings>
    <entry key="en" value="en-gb"/>
  </jaxrs:languageMappings>
</jaxrs:server>
```

When the preceding server endpoint receives the HTTP request, it automatically creates a new accept language header with the value, **en-gb**, and strips the **.en** suffix from the resource URL.

18.1.2. jaxrs:server Attributes

Attributes

[Table 18.1, “JAX-RS Server Endpoint Attributes”](#) describes the attributes available on the **jaxrs:server** element.

Table 18.1. JAX-RS Server Endpoint Attributes

Attribute	Description
id	Specifies a unique identifier that other configuration elements can use to refer to the endpoint.
address	Specifies the address of an HTTP endpoint. This value will override the value specified in the services contract.
basePackages	(Spring only) Enables auto-discovery, by specifying a comma-separated list of Java packages, which are searched to discover JAX-RS root resource classes and/or JAX-RS provider classes.
beanNames	Specifies a space-separated list of bean IDs of JAX-RS root resource beans. In the context of Spring XML, it is possible to define a root resource beans' lifecycle by setting the scope attribute on the root resource bean element.
bindingId	Specifies the ID of the message binding the service uses. A list of valid binding IDs is provided in Chapter 23, Apache CXF Binding IDs .

Attribute	Description
bus	Specifies the ID of the Spring bean configuring the bus used to manage the service endpoint. This is useful when configuring several endpoints to use a common set of features.
docLocation	Specifies the location of an external WADL document.
modelRef	Specifies a model schema as a classpath resource (for example, a URL of the form classpath:/path/to/model.xml). For details of how to define a JAX-RS model schema, see Section 18.3, “Defining REST Services with the Model Schema” .
publish	Specifies if the service should be automatically published. If set to false , the developer must explicitly publish the endpoint.
publishedEndpointUrl	Specifies the URL base address, which gets inserted into the wadl:resources/@base attribute of the auto-generated WADL interface.
serviceAnnotation	(Spring only) Specifies the service annotation class name for auto-discovery in Spring. When used in combination with the basePackages property, this option restricts the collection of auto-discovered classes to include only the classes that are annotated by this annotation type. guess!! Is this correct?
serviceClass	Specifies the name of a JAX-RS root resource class (which implements a JAX-RS service). In this case, the class is instantiated by Apache CXF, not by Blueprint or Spring. If you want to instantiate the class in Blueprint or Spring, use the jaxrs:serviceBeans child element instead.
serviceName	Specifies the service QName (using the format ns:name) for the JAX-RS endpoint in the special case where a JMS transport is used. For details, see the section called “Using the JMS transport” .
staticSubresourceResolution	If true , disables dynamic resolution of static sub-resources. Default is false .

Attribute	Description
transportId	For selecting a non-standard transport layer (in place of HTTP). In particular, you can select the JMS transport by setting this property to http://cxf.apache.org/transports/jms . For details, see the section called “Using the JMS transport”.
abstract	(Spring only) Specifies if the bean is an abstract bean. Abstract beans act as parents for concrete bean definitions and are not instantiated. The default is false . Setting this to true instructs the bean factory not to instantiate the bean.
depends-on	(Spring only) Specifies a list of beans that the endpoint depends on being instantiated before the endpoint can be instantiated.

18.1.3. jaxrs:server Child Elements

Child elements

Table 18.2, “JAX-RS Server Endpoint Child Elements” describes the child elements of the **jaxrs:server** element.

Table 18.2. JAX-RS Server Endpoint Child Elements

Element	Description
jaxrs:executor	Specifies a Java Executor (thread pool implementation) that is used for the service. This is specified using an embedded bean definition.
jaxrs:features	Specifies a list of beans that configure advanced features of Apache CXF. You can provide either a list of bean references or a list of embedded beans.
jaxrs:binding	Not used.
jaxrs:dataBinding	Specifies the class implementing the data binding used by the endpoint. This is specified using an embedded bean definition. For more details, see the section called “Specifying the data binding”.

Element	Description
jaxrs:inInterceptors	Specifies a list of interceptors that process inbound requests. For more information see Part VII, "Developing Apache CXF Interceptors" .
jaxrs:inFaultInterceptors	Specifies a list of interceptors that process inbound fault messages. For more information see Part VII, "Developing Apache CXF Interceptors" .
jaxrs:outInterceptors	Specifies a list of interceptors that process outbound replies. For more information see Part VII, "Developing Apache CXF Interceptors" .
jaxrs:outFaultInterceptors	Specifies a list of interceptors that process outbound fault messages. For more information see Part VII, "Developing Apache CXF Interceptors" .
jaxrs:invoker	Specifies an implementation of the org.apache.cxf.service.Invoker interface used by the service. [a]
jaxrs:serviceFactories	Provides you with the maximum degree of control over the lifecycle of the JAX-RS root resources associated with this endpoint. The children of this element (which must be instances of org.apache.cxf.jaxrs.lifecycle.ResourceProvider type) are used to create JAX-RS root resource instances.
jaxrs:properties	Specifies a Spring map of properties that are passed along to the endpoint. These properties can be used to control features like enabling MTOM support.
jaxrs:serviceBeans	The children of this element are instances of (bean element) or references to (ref element) JAX-RS root resources. Note that in this case the scope attribute (Spring only), if present in the bean element, is ignored.
jaxrs:modelBeans	Consists of a list of references to one or more org.apache.cxf.jaxrs.model.UserResource beans, which are the basic elements of a resource model (corresponding to jaxrs:resource elements). For details, see Section 18.3, "Defining REST Services with the Model Schema" .

Element	Description
jaxrs:model	Defines a resource model directly in this endpoint (that is, this jaxrs:model element can contain one or more jaxrs:resource elements). For details, see Section 18.3, “Defining REST Services with the Model Schema” .
jaxrs:providers	Enables you to register one or more custom JAX-RS providers with this endpoint. The children of this element are instances of (bean element) or references to (ref element) JAX-RS providers.
jaxrs:extensionMappings	When the URL of a REST invocation ends in a file extension, you can use this element to associate it automatically with a particular content type. For example, the .xml file extension could be associated with the application/xml content type. For details, see the section called “Extension mappings and language mappings” .
jaxrs:languageMappings	When the URL of a REST invocation ends in a language suffix, you can use this element to map this to a particular language. For example, the .en language suffix could be associated with the en-GB language. For details, see the section called “Extension mappings and language mappings” .
jaxrs:schemaLocations	Specifies one or more XML schemas used for validating XML message content. This element can contain one or more jaxrs:schemaLocation elements, each specifying the location of an XML schema file (usually as a classpath URL). For details, see the section called “Schema validation” .
jaxrs:resourceComparator	Enables you to register a custom resource comparator, which implements the algorithm used to match an incoming URL path to a particular resource class or method.
jaxrs:resourceClasses	(Blueprint only) Can be used instead of the jaxrs:server/@serviceClass attribute, if you want to create multiple resources from class names. The children of jaxrs:resourceClasses must be class elements with a name attribute set to the name of the resource class. In this case, the classes are instantiated by Apache CXF, not by Blueprint or Spring.

[a] The Invoker implementation controls how a service is invoked. For example, it controls whether each request is handled by a new instance of the service implementation or if state is preserved across invocations.

18.2. CONFIGURING JAX-RS CLIENT ENDPOINTS

18.2.1. Defining a JAX-RS Client Endpoint

Injecting client proxies

The main point of instantiating a client proxy bean in an XML language (Blueprint XML or Spring XML) is in order to inject it into another bean, which can then use the client proxy to invoke the REST service. To create a client proxy bean in XML, use the **jaxrs:client** element.

Namespaces

The JAX-RS client endpoint is defined using a **different** XML namespace from the server endpoint. The following table shows which namespace to use for which XML language:

XML Language	Namespace for client endpoint
Blueprint	http://cxf.apache.org/blueprint/jaxrs-client
Spring	http://cxf.apache.org/jaxrs-client

Basic client endpoint definition

The following example shows how to create a client proxy bean in Blueprint XML or Spring XML:

```
<jaxrs:client id="restClient"
    address="http://localhost:8080/test/services/rest"
    serviceClass="org.apache.cxf.systest.jaxrs.BookStoreJaxrsJaxws"/>
```

Where you must set the following attributes to define the basic client endpoint:

id

The bean ID of the client proxy can be used to inject the client proxy into other beans in your XML configuration.

address

The address attribute specifies the base URL of the REST invocations.

serviceClass

The **serviceClass** attribute provides a description of the REST service by specifying a root resource class (annotated by **@Path**). In fact, this is a **server** class, but it is not used directly by the client. The specified class is used only for its metadata (through Java reflection and JAX-RS annotations), which is used to construct the client proxy dynamically.

Specifying headers

You can add HTTP headers to the client proxy's invocations using the **jaxrs:headers** child elements, as follows:

```
<jaxrs:client id="restClient"
    address="http://localhost:8080/test/services/rest"
    serviceClass="org.apache.cxf.systest.jaxrs.BookStoreJaxrsJaxws">
```

```

    inheritHeaders="true">
    <jaxrs:headers>
        <entry key="Accept" value="text/xml"/>
    </jaxrs:headers>
</jaxrs:client>

```

18.2.2. jaxrs:client Attributes

Attributes

[Table 18.3, “JAX-RS Client Endpoint Attributes”](#) describes the attributes available on the **jaxrs:client** element.

Table 18.3. JAX-RS Client Endpoint Attributes

Attribute	Description
address	Specifies the HTTP address of the endpoint where the consumer will make requests. This value overrides the value set in the contract.
bindingId	Specifies the ID of the message binding the consumer uses. A list of valid binding IDs is provided in Chapter 23, Apache CXF Binding IDs .
bus	Specifies the ID of the Spring bean configuring the bus managing the endpoint.
inheritHeaders	Specifies whether the headers set for this proxy will be inherited, if a subresource proxy is created from this proxy. Default is false .
username	Specifies the username used for simple username/password authentication.
password	Specifies the password used for simple username/password authentication.
modelRef	Specifies a model schema as a classpath resource (for example, a URL of the form classpath:/path/to/model.xml). For details of how to define a JAX-RS model schema, see Section 18.3, “Defining REST Services with the Model Schema” .

Attribute	Description
serviceClass	Specifies the name of a service interface or a resource class (that is annotated with @PATH), re-using it from the JAX-RS server implementation. In this case, the specified class is not invoked directly (it is actually a server class). The specified class is used only for its metadata (through Java reflection and JAX-RS annotations), which is used to construct the client proxy dynamically.
serviceName	Specifies the service QName (using the format ns:name) for the JAX-RS endpoint in the special case where a JMS transport is used. For details, see the section called "Using the JMS transport" .
threadSafe	Specifies whether or not the client proxy is thread-safe. Default is false .
transportId	For selecting a non-standard transport layer (in place of HTTP). In particular, you can select the JMS transport by setting this property to http://cxf.apache.org/transports/jms . For details, see the section called "Using the JMS transport" .
abstract	(Spring only) Specifies if the bean is an abstract bean. Abstract beans act as parents for concrete bean definitions and are not instantiated. The default is false . Setting this to true instructs the bean factory not to instantiate the bean.
depends-on	(Spring only) Specifies a list of beans that the endpoint depends on being instantiated before it can be instantiated.

18.2.3. `jaxrs:client` Child Elements

Child elements

[Table 18.4, "JAX-RS Client Endpoint Child Elements"](#) describes the child elements of the `jaxrs:client` element.

Table 18.4. JAX-RS Client Endpoint Child Elements

Element	Description
jaxrs:executor	

Element	Description
jaxrs:features	Specifies a list of beans that configure advanced features of Apache CXF. You can provide either a list of bean references or a list of embedded beans.
jaxrs:binding	Not used.
jaxrs:dataBinding	Specifies the class implementing the data binding used by the endpoint. This is specified using an embedded bean definition. For more details, see the section called "Specifying the data binding" .
jaxrs:inInterceptors	Specifies a list of interceptors that process inbound responses. For more information see Part VII, "Developing Apache CXF Interceptors" .
jaxrs:inFaultInterceptors	Specifies a list of interceptors that process inbound fault messages. For more information see Part VII, "Developing Apache CXF Interceptors" .
jaxrs:outInterceptors	Specifies a list of interceptors that process outbound requests. For more information see Part VII, "Developing Apache CXF Interceptors" .
jaxrs:outFaultInterceptors	Specifies a list of interceptors that process outbound fault messages. For more information see Part VII, "Developing Apache CXF Interceptors" .
jaxrs:properties	Specifies a map of properties that are passed to the endpoint.
jaxrs:providers	Enables you to register one or more custom JAX-RS providers with this endpoint. The children of this element are instances of (bean element) or references to (ref element) JAX-RS providers.
jaxrs:modelBeans	Consists of a list of references to one or more org.apache.cxf.jaxrs.model.UserResource beans, which are the basic elements of a resource model (corresponding to jaxrs:resource elements). For details, see Section 18.3, "Defining REST Services with the Model Schema" .
jaxrs:model	Defines a resource model directly in this endpoint (that is, a jaxrs:model element containing one or more jaxrs:resource elements). For details, see Section 18.3, "Defining REST Services with the Model Schema" .

Element	Description
jaxrs:headers	Used for setting headers on the outgoing message. For details, see the section called "Specifying headers" .
jaxrs:schemaLocations	Specifies one or more XML schemas used for validating XML message content. This element can contain one or more jaxrs:schemaLocation elements, each specifying the location of an XML schema file (usually as a classpath URL). For details, see the section called "Schema validation" .

18.3. DEFINING REST SERVICES WITH THE MODEL SCHEMA

RESTful services without annotations

The JAX-RS model schema makes it possible to define RESTful services **without** annotating Java classes. That is, instead of adding annotations like **@Path**, **@PathParam**, **@Consumes**, **@Produces**, and so on, directly to a Java class (or interface), you can provide all of the relevant REST metadata in a separate XML file, using the model schema. This can be useful, for example, in cases where you are unable to modify the Java source that implements the service.

Example model schema

[Example 18.1, "Sample JAX-RS Model Schema"](#) shows an example of a model schema that defines service metadata for the **BookStoreNoAnnotations** root resource class.

Example 18.1. Sample JAX-RS Model Schema

```
<model xmlns="http://cxf.apache.org/jaxrs">
  <resource name="org.apache.cxf.systest.jaxrs.BookStoreNoAnnotations" path="bookstore"
    produces="application/json" consumes="application/json">
    <operation name="getBook" verb="GET" path="/books/{id}" produces="application/xml">
      <param name="id" type="PATH"/>
    </operation>
    <operation name="getBookChapter" path="/books/{id}/chapter">
      <param name="id" type="PATH"/>
    </operation>
    <operation name="updateBook" verb="PUT">
      <param name="book" type="REQUEST_BODY"/>
    </operation>
  </resource>
  <resource name="org.apache.cxf.systest.jaxrs.ChapterNoAnnotations">
    <operation name="getItself" verb="GET"/>
    <operation name="updateChapter" verb="PUT" consumes="application/xml">
      <param name="content" type="REQUEST_BODY"/>
    </operation>
  </resource>
</model>
```

Namespaces

The XML namespace that you use to define a model schema depends on whether you are defining the corresponding JAX-RS endpoint in Blueprint XML or in Spring XML. The following table shows which namespace to use for which XML language:

XML Language	Namespace
Blueprint	http://cxf.apache.org/blueprint/jaxrs
Spring	http://cxf.apache.org/jaxrs

How to attach a model schema to an endpoint

To define and attach a model schema to an endpoint, perform the following steps:

1. Define the model schema, using the appropriate XML namespace for your chosen injection platform (Blueprint XML or Spring XML).
2. Add the model schema file to your project's resources, so that the schema file is available on the classpath in the final package (JAR, WAR, or OSGi bundle file).
3. Configure the endpoint to use the model schema, by setting the endpoint's **modelRef** attribute to the location of the model schema on the classpath (using a classpath URL).
4. If necessary, instantiate the root resources explicitly, using the **jaxrs:serviceBeans** element. You can skip this step, if the model schema references root resource classes directly (instead of referencing base interfaces).



NOTE

Alternatively, it is also possible to embed a model schema directly into a JAX-RS endpoint, using the endpoint's **jaxrs:model** child element.

Configuration of model schema referencing a class

If the model schema applies directly to root resource classes, there is no need to define any root resource beans using the **jaxrs:serviceBeans** element, because the model schema automatically instantiates the root resource beans.

For example, given that **customer-resources.xml** is a model schema that associates metadata with customer resource classes, you could instantiate a **customerService** service endpoint as follows:

```
<jaxrs:server id="customerService"
    address="/customers"
    modelRef="classpath:/org/example/schemas/customer-resources.xml" />
```

Configuration of model schema referencing an interface

If the model schema applies to Java interfaces (which are the base interfaces of the root resources), you must instantiate the root resource classes using the **jaxrs:serviceBeans** element in the endpoint.

For example, given that **customer-interfaces.xml** is a model schema that associates metadata with customer interfaces, you could instantiate a **customerService** service endpoint as follows:

```
<jaxrs:server id="customerService"
    address="/customers"
    modelRef="classpath:/org/example/schemas/customer-interfaces.xml">
    <jaxrs:serviceBeans>
        <ref component-id="serviceBean" />
    </jaxrs:serviceBeans>
</jaxrs:server>

<bean id="serviceBean" class="service.CustomerService"/>
```

Model Schema Reference

A model schema is defined using the following XML elements:

model

Root element of the model schema. If you need to reference the model schema (for example, from a JAX-RS endpoint using the **modelRef** attribute), you should set the **id** attribute on this element.

model/resource

The **resource** element is used to associate metadata with a specific root resource class (or with a corresponding interface). You can define the following attributes on the **resource** element:

Attribute	Description +
name	The name of the resource class (or corresponding interface) to which this resource model is applied. +
path	The component of the REST URL path that maps to this resource. +
consumes	Specifies the content type (Internet media type) consumed by this resource—for example, application/xml or application/json . +
produces	Specifies the content type (Internet media type) produced by this resource—for example, application/xml or application/json . +

model/resource/operation

The **operation** element is used to associate metadata with Java methods. You can define the following attributes on an **operation** element:

Attribute	Description +
name	The name of the Java method to which this element is applied. +
path	The component of the REST URL path that maps to this method. This attribute value can include parameter references, for example: path="/books/{id}/chapter" , where {id} extracts the value of the id parameter from the path. +
verb	Specifies the HTTP verb that maps to this method. Typically one of: GET , POST , PUT , or DELETE . If the HTTP verb is not specified, it is assumed that the Java method is a sub-resource locator , which returns a reference to a sub-resource object (where the sub-resource class must also be provided with metadata using a resource element). +
consumes	Specifies the content type (Internet media type) consumed by this operation—for example, application/xml or application/json . +
produces	Specifies the content type (Internet media type) produced by this operation—for example, application/xml or application/json . +
oneway	If true , configures the operation to be oneway , meaning that no reply message is needed. Defaults to false . +

model/resource/operation/param

The **param** element is used extract a value from the REST URL and inject it into one of the method parameters. You can define the following attributes on a **param** element:

Attribute	Description +
-----------	---------------

Attribute	Description +
name	The name of the Java method parameter to which this element is applied. +
type	Specifies how the parameter value is extracted from the REST URL or message. It can be set to one of the following values: PATH , QUERY , MATRIX , HEADER , COOKIE , FORM , CONTEXT , REQUEST_BODY . +
defaultValue	Default value to inject into the parameter, in case a value could not be extracted from the REST URL or message. +
encoded	If true , the parameter value is injected in its URI encoded form (that is, using %nn encoding). Default is false . For example, when extracting a parameter from the URL path, /name/Joe%20Bloggs with encoded set to true , the parameter is injected as Joe%20Bloggs ; otherwise, the parameter would be injected as Joe Bloggs . +

CHAPTER 19. APACHE CXF LOGGING

Abstract

This chapter describes how to configure logging in the Apache CXF runtime.

19.1. OVERVIEW OF APACHE CXF LOGGING

Overview

Apache CXF uses the Java logging utility, **java.util.logging**. Logging is configured in a logging configuration file that is written using the standard **java.util.Properties** format. To run logging on an application, you can specify logging programmatically or by defining a property at the command that points to the logging configuration file when you start the application.

Default properties file

Apache CXF comes with a default **logging.properties** file, which is located in your *InstallDir/etc* directory. This file configures both the output destination for the log messages and the message level that is published. The default configuration sets the loggers to print message flagged with the **WARNING** level to the console. You can either use the default file without changing any of the configuration settings or you can change the configuration settings to suit your specific application.

Logging feature

Apache CXF includes a logging feature that can be plugged into your client or your service to enable logging. [Example 19.1, “Configuration for Enabling Logging”](#) shows the configuration to enable the logging feature.

Example 19.1. Configuration for Enabling Logging

```
<jaxws:endpoint...>
  <jaxws:features>
    <bean class="org.apache.cxf.feature.LoggingFeature"/>
  </jaxws:features>
</jaxws:endpoint>
```

For more information, see [Section 19.6, “Logging Message Content”](#).

Where to begin?

To run a simple example of logging follow the instructions outlined in a [Section 19.2, “Simple Example of Using Logging”](#).

For more information on how logging works in Apache CXF, read this entire chapter.

More information on **java.util.logging**

The **java.util.logging** utility is one of the most widely used Java logging frameworks. There is a lot of information available online that describes how to use and extend this framework. As a starting point, however, the following documents gives a good overview of **java.util.logging**:

- <http://download.oracle.com/javase/1.5.0/docs/guide/logging/overview.html>
- <http://download.oracle.com/javase/1.5.0/docs/api/java/util/logging/package-summary.html>

19.2. SIMPLE EXAMPLE OF USING LOGGING

Changing the log levels and output destination

To change the log level and output destination of the log messages in the wsdl_first sample application, complete the following steps:

1. Run the sample server as described in the **Running the demo using java** section of the **README.txt** file in the **InstallDir/samples/wsdl_first** directory. Note that the **server start** command specifies the default **logging.properties** file, as follows:

Platform	Command +
Windows	start java - Djava.util.logging.config.file=%CXF_HOME%\etc\logging.properties demo.hw.server.Server +
UNIX	java - Djava.util.logging.config.file=\$CXF_HOME/Etc/logging.properties demo.hw.server.Server & +

The default **logging.properties** file is located in the **InstallDir/etc** directory. It configures the Apache CXF loggers to print **WARNING** level log messages to the console. As a result, you see very little printed to the console.

2. Stop the server as described in the **README.txt** file.
3. Make a copy of the default **logging.properties** file, name it **mylogging.properties** file, and save it in the same directory as the default **logging.properties** file.
4. Change the global logging level and the console logging levels in your **mylogging.properties** file to **INFO** by editing the following lines of configuration:

```
.level= INFO
java.util.logging.ConsoleHandler.level = INFO
```

5. Restart the server using the following command:

Platform	Command +

Platform	Command +
Windows	start java - Djava.util.logging.config.file=%CXF_HOME%\etc\mylogging.properties demo.hw.server.Server +
UNIX	java - Djava.util.logging.config.file=\$CXF_HOME /etc/mylogging.properties demo.hw.server.Server & +

Because you configured the global logging and the console logger to log messages of level **INFO**, you see a lot more log messages printed to the console.

19.3. DEFAULT LOGGING CONFIGURATION FILE

19.3.1. Overview of Logging Configuration

The default logging configuration file, **logging.properties**, is located in the **InstallDir/etc** directory. It configures the Apache CXF loggers to print **WARNING** level messages to the console. If this level of logging is suitable for your application, you do not have to make any changes to the file before using it. You can, however, change the level of detail in the log messages. For example, you can change whether log messages are sent to the console, to a file or to both. In addition, you can specify logging at the level of individual packages.



NOTE

This section discusses the configuration properties that appear in the default **logging.properties** file. There are, however, many other **java.util.logging** configuration properties that you can set. For more information on the **java.util.logging** API, see the **java.util.logging** javadoc at:
<http://download.oracle.com/javase/1.5/docs/api/java/util/logging/package-summary.html>.

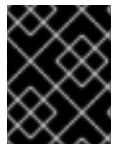
19.3.2. Configuring Logging Output

Overview

The Java logging utility, **java.util.logging**, uses handler classes to output log messages. [Table 19.1, "Java.util.logging Handler Classes"](#) shows the handlers that are configured in the default **logging.properties** file.

Table 19.1. Java.util.logging Handler Classes

Handler Class	Outputs to
ConsoleHandler	Outputs log messages to the console
FileHandler	Outputs log messages to a file



IMPORTANT

The handler classes must be on the system classpath in order to be installed by the Java VM when it starts. This is done when you set the Apache CXF environment.

Configuring the console handler

[Example 19.2, “Configuring the Console Handler”](#) shows the code for configuring the console logger.

Example 19.2. Configuring the Console Handler

```
handlers= java.util.logging.ConsoleHandler
```

The console handler also supports the configuration properties shown in [Example 19.3, “Console Handler Properties”](#).

Example 19.3. Console Handler Properties

```
java.util.logging.ConsoleHandler.level = WARNING
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
```

The configuration properties shown in [Example 19.3, “Console Handler Properties”](#) can be explained as follows:

The console handler supports a separate log level configuration property. This allows you to limit the log messages printed to the console while the global logging setting can be different (see [Section 19.3.3, “Configuring Logging Levels”](#)). The default setting is **WARNING**.

Specifies the **java.util.logging** formatter class that the console handler class uses to format the log messages. The default setting is the **java.util.logging.SimpleFormatter**.

Configuring the file handler

[Example 19.4, “Configuring the File Handler”](#) shows code that configures the file handler.

Example 19.4. Configuring the File Handler

```
handlers= java.util.logging.FileHandler
```

The file handler also supports the configuration properties shown in [Example 19.5, “File Handler Configuration Properties”](#).

Example 19.5. File Handler Configuration Properties

```
java.util.logging.FileHandler.pattern = %h/java%u.log  
java.util.logging.FileHandler.limit = 50000  
java.util.logging.FileHandler.count = 1  
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter
```

The configuration properties shown in [Example 19.5, “File Handler Configuration Properties”](#) can be explained as follows:

Specifies the location and pattern of the output file. The default setting is your home directory.

Specifies, in bytes, the maximum amount that the logger writes to any one file. The default setting is **50000**. If you set it to zero, there is no limit on the amount that the logger writes to any one file.

Specifies how many output files to cycle through. The default setting is **1**.

Specifies the **java.util.logging** formatter class that the file handler class uses to format the log messages. The default setting is the **java.util.logging.XMLFormatter**.

Configuring both the console handler and the file handler

You can set the logging utility to output log messages to both the console and to a file by specifying the console handler and the file handler, separated by a comma, as shown in [Configuring Both Console Logging and File](#).

Configuring Both Console Logging and File

Logging

```
handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler
```

19.3.3. Configuring Logging Levels

Logging levels

The **java.util.logging** framework supports the following levels of logging, from the least verbose to the most verbose:

- **SEVERE**
- **WARNING**
- **INFO**
- **CONFIG**
- **FINE**
- **FINER**
- **FINEST**

Configuring the global logging level

To configure the types of event that are logged across all loggers, configure the global logging level as shown in [Example 19.6, "Configuring Global Logging Levels"](#).

Example 19.6. Configuring Global Logging Levels

```
.level= WARNING
```

Configuring logging at an individual package

```
level
```

The **java.util.logging** framework supports configuring logging at the level of an individual package. For example, the line of code shown in [Example 19.7, "Configuring Logging at the Package Level"](#) configures logging at a **SEVERE** level on classes in the `com.xyz.foo` package.

Example 19.7. Configuring Logging at the Package Level

```
com.xyz.foo.level = SEVERE
```

19.4. ENABLING LOGGING AT THE COMMAND LINE

Overview

You can run the logging utility on an application by defining a **java.util.logging.config.file** property when you start the application. You can either specify the default **logging.properties** file or a **logging.properties** file that is unique to that application.

Specifying the log configuration file on application

```
start-up
```

To specify logging on application start-up add the flag shown in [Example 19.8, "Flag to Start Logging on the Command Line"](#) when starting the application.

Example 19.8. Flag to Start Logging on the Command Line

```
-Djava.util.logging.config.file=myfile
```

19.5. LOGGING FOR SUBSYSTEMS AND SERVICES

Overview

You can use the **com.xyz.foo.level** configuration property described in [the section called “Configuring logging at an individual package”](#) to set fine-grained logging for specified Apache CXF logging subsystems.

Apache CXF logging subsystems

[Table 19.2, “Apache CXF Logging Subsystems”](#) shows a list of available Apache CXF logging subsystems.

Table 19.2. Apache CXF Logging Subsystems

Subsystem	Description
org.apache.cxf.aegis	Aegis binding
org.apache.cxf.binding.coloc	colocated binding
org.apache.cxf.binding.http	HTTP binding
org.apache.cxf.binding.jbi	JBI binding
org.apache.cxf.binding.object	Java Object binding
org.apache.cxf.binding.soap	SOAP binding
org.apache.cxf.binding.xml	XML binding
org.apache.cxf.bus	Apache CXF bus
org.apache.cxf.configuration	configuration framework
org.apache.cxf.endpoint	server and client endpoints
org.apache.cxf.interceptor	interceptors
org.apache.cxf.jaxws	Front-end for JAX-WS style message exchange, JAX-WS handler processing, and interceptors relating to JAX-WS and configuration
org.apache.cxf.jbi	JBI container integration classes
org.apache.cxf.jca	JCA container integration classes
org.apache.cxf.js	JavaScript front-end
org.apache.cxf.transport.http	HTTP transport
org.apache.cxf.transport.https	secure version of HTTP transport, using HTTPS
org.apache.cxf.transport.jbi	JBI transport

Subsystem	Description
org.apache.cxf.transport.jms	JMS transport
org.apache.cxf.transport.local	transport implementation using local file system
org.apache.cxf.transport.servlet	HTTP transport and servlet implementation for loading JAX-WS endpoints into a servlet container
org.apache.cxf.ws.addressing	WS-Addressing implementation
org.apache.cxf.ws.policy	WS-Policy implementation
org.apache.cxf.ws.rm	WS-ReliableMessaging (WS-RM) implementation
org.apache.cxf.ws.security.wss4j	WSS4J security implementation

Example

The WS-Addressing sample is contained in the ***InstallDir/samples/ws_addressing*** directory. Logging is configured in the **logging.properties** file located in that directory. The relevant lines of configuration are shown in [Example 19.9, “Configuring Logging for WS-Addressing”](#).

Example 19.9. Configuring Logging for WS-Addressing

```
java.util.logging.ConsoleHandler.formatter = demos.ws_addressing.common.ConciseFormatter
...
org.apache.cxf.ws.addressing.soap.MAPCodec.level = INFO
```

The configuration in [Example 19.9, “Configuring Logging for WS-Addressing”](#) enables the snooping of log messages relating to WS-Addressing headers, and displays them to the console in a concise form.

For information on running this sample, see the **README.txt** file located in the ***InstallDir/samples/ws_addressing*** directory.

19.6. LOGGING MESSAGE CONTENT

Overview

You can log the content of the messages that are sent between a service and a consumer. For example, you might want to log the contents of SOAP messages that are being sent between a service and a consumer.

Configuring message content logging

To log the messages that are sent between a service and a consumer, and vice versa, complete the following steps:

1. Add the logging feature to your endpoint’s configuration.

2. Add the logging feature to your consumer's configuration.
3. Configure the logging system log **INFO** level messages.

Adding the logging feature to an endpoint

Add the logging feature to your endpoint's configuration as shown in [Example 19.10, "Adding Logging to Endpoint Configuration"](#).

Example 19.10. Adding Logging to Endpoint Configuration

```
<jaxws:endpoint ...>
  <jaxws:features>
    <bean class="org.apache.cxf.feature.LoggingFeature"/>
  </jaxws:features>
</jaxws:endpoint>
```

The example XML shown in [Example 19.10, "Adding Logging to Endpoint Configuration"](#) enables the logging of SOAP messages.

Adding the logging feature to a consumer

Add the logging feature to your client's configuration as shown in [Example 19.11, "Adding Logging to Client Configuration"](#).

Example 19.11. Adding Logging to Client Configuration

```
<jaxws:client ...>
  <jaxws:features>
    <bean class="org.apache.cxf.feature.LoggingFeature"/>
  </jaxws:features>
</jaxws:client>
```

The example XML shown in [Example 19.11, "Adding Logging to Client Configuration"](#) enables the logging of SOAP messages.

Set logging to log INFO level messages

Ensure that the **logging.properties** file associated with your service is configured to log **INFO** level messages, as shown in [Example 19.12, "Setting the Logging Level to INFO"](#).

Example 19.12. Setting the Logging Level to INFO

```
.level=INFO
java.util.logging.ConsoleHandler.level = INFO
```

Logging SOAP messages

To see the logging of SOAP messages modify the wsdl_first sample application located in the **InstallDir/samples/wsdl_first** directory, as follows:

1. Add the **jaxws:features** element shown in [Example 19.13, “Endpoint Configuration for Logging SOAP Messages”](#) to the **cxn.xml** configuration file located in the wsdl_first sample’s directory:

Example 19.13. Endpoint Configuration for Logging SOAP Messages

```
<jaxws:endpoint name="{http://apache.org/hello_world_soap_http}SoapPort"
    createdFromAPI="true">
    <jaxws:properties>
        <entry key="schema-validation-enabled" value="true" />
    </jaxws:properties>
    <jaxws:features>
        <bean class="org.apache.cxf.feature.LoggingFeature"/>
    </jaxws:features>
</jaxws:endpoint>
```

2. The sample uses the default **logging.properties** file, which is located in the **InstallDir/etc** directory. Make a copy of this file and name it **mylogging.properties**.
3. In the **mylogging.properties** file, change the logging levels to **INFO** by editing the **.level** and the **java.util.logging.ConsoleHandler.level** configuration properties as follows:

```
.level= INFO
java.util.logging.ConsoleHandler.level = INFO
```

4. Start the server using the new configuration settings in both the **cxn.xml** file and the **mylogging.properties** file as follows:

Platform	Command +
Windows	start java - Djava.util.logging.config.file=%CXF_HOME%\etc\mylogging.properties demo.hw.server.Server +
UNIX	java - Djava.util.logging.config.file=\$CXF_HOME/Etc/mylogging.properties demo.hw.server.Server & +

5. Start the hello world client using the following command:

Platform	Command +
----------	-----------

Platform	Command +
Windows	<pre>java -Djava.util.logging.config.file=%CXF_HOME%\etc\mylogging.properties demo.hw.client.Client ./wsdl\hello_world.wsdl</pre> <p>+</p>
UNIX	<pre>java -Djava.util.logging.config.file=\$CXF_HOME/etc/mylogging.properties demo.hw.client.Client ./wsdl/hello_world.wsdl</pre> <p>+</p>

The SOAP messages are logged to the console.

CHAPTER 20. DEPLOYING WS-ADDRESSING

Abstract

Apache CXF supports WS-Addressing for JAX-WS applications. This chapter explains how to deploy WS-Addressing in the Apache CXF runtime environment.

20.1. INTRODUCTION TO WS-ADDRESSING

Overview

WS-Addressing is a specification that allows services to communicate addressing information in a transport neutral way. It consists of two parts:

- A structure for communicating a reference to a Web service endpoint
- A set of Message Addressing Properties (MAP) that associate addressing information with a particular message

Supported specifications

Apache CXF supports both the WS-Addressing 2004/08 specification and the WS-Addressing 2005/03 specification.

Further information

For detailed information on WS-Addressing, see the 2004/08 submission at <http://www.w3.org/Submission/ws-addressing/>.

20.2. WS-ADDRESSING INTERCEPTORS

Overview

In Apache CXF, WS-Addressing functionality is implemented as interceptors. The Apache CXF runtime uses interceptors to intercept and work with the raw messages that are being sent and received. When a transport receives a message, it creates a message object and sends that message through an interceptor chain. If the WS-Addressing interceptors are added to the application's interceptor chain, any WS-Addressing information included with a message is processed.

WS-Addressing Interceptors

The WS-Addressing implementation consists of two interceptors, as described in [Table 20.1, "WS-Addressing Interceptors"](#).

Table 20.1. WS-Addressing Interceptors

Interceptor	Description
<code>org.apache.cxf.ws.addressing.MAPAggregat or</code>	A logical interceptor responsible for aggregating the Message Addressing Properties (MAPs) for outgoing messages.

Interceptor	Description
org.apache.cxf.ws.addressing.soap.MAPCodec	A protocol-specific interceptor responsible for encoding and decoding the Message Addressing Properties (MAPs) as SOAP headers.

20.3. ENABLING WS-ADDRESSING

Overview

To enable WS-Addressing the WS-Addressing interceptors must be added to the inbound and outbound interceptor chains. This is done in one of the following ways:

- [Apache CXF Features](#)
- [RMAssertion and WS-Policy Framework](#)
- [Using Policy Assertion in a WS-Addressing Feature](#)

Adding WS-Addressing as a Feature

WS-Addressing can be enabled by adding the WS-Addressing feature to the client and the server configuration as shown in [Example 20.1, "client.xml and Adding WS-Addressing Feature to Client Configuration"](#) and [Example 20.2, "server.xml and Adding WS-Addressing Feature to Server Configuration"](#) respectively.

Example 20.1. client.xml and Adding WS-Addressing Feature to Client Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://cxf.apache.org/ws/addressing
           http://cxf.apache.org/schemas/ws-addr-conf.xsd">

    <jaxws:client ...>
        <jaxws:features>
            <wsa:addressing/>
        </jaxws:features>
    </jaxws:client>
</beans>
```

Example 20.2. server.xml and Adding WS-Addressing Feature to Server Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jaxws="http://cxf.apache.org/jaxws"
xmlns:wsa="http://cx.f.apache.org/ws/addressing"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

<jaxws:endpoint ...>
  <jaxws:features>
    <wsa:addressing/>
  </jaxws:features>
</jaxws:endpoint>
</beans>

```

20.4. CONFIGURING WS-ADDRESSING ATTRIBUTES

Overview

The Apache CXF WS-Addressing feature element is defined in the namespace <http://cx.f.apache.org/ws/addressing>. It supports the two attributes described in [Table 20.2, “WS-Addressing Attributes”](#).

Table 20.2. WS-Addressing Attributes

Attribute Name	Value
allowDuplicates	A boolean that determines if duplicate MessageIDs are tolerated. The default setting is true .
usingAddressingAdvisory	A boolean that indicates if the presence of the UsingAddressing element in the WSDL is advisory only; that is, its absence does not prevent the encoding of WS-Addressing headers.

Configuring WS-Addressing attributes

Configure WS-Addressing attributes by adding the attribute and the value you want to set it to the WS-Addressing feature in your server or client configuration file. For example, the following configuration extract sets the **allowDuplicates** attribute to **false** on the server endpoint:

```

<beans ... xmlns:wsa="http://cx.f.apache.org/ws/addressing" ...>
  <jaxws:endpoint ...>
    <jaxws:features>
      <wsa:addressing allowDuplicates="false"/>
    </jaxws:features>
  </jaxws:endpoint>
</beans>

```

Using a WS-Policy assertion embedded in a feature

In Example 20.3, "Using the Policies to Configure WS-Addressing" an addressing policy assertion to enable non-anonymous responses is embedded in the **policies** element.

Example 20.3. Using the Policies to Configure WS-Addressing

```
<?xml version="1.0" encoding="UTF-8"?>
<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:wsp="http://www.w3.org/2006/07/ws-policy"
       xmlns:policy="http://cxf.apache.org/policy-config"
       xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="
http://www.w3.org/2006/07/ws-policy http://www.w3.org/2006/07/ws-policy.xsd
http://cxf.apache.org/ws/addressing http://cxf.apache.org/schema/ws/addressing.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

<jaxws:endpoint name="{http://cxf.apache.org/greeter_control}GreeterPort"
                 createdFromAPI="true">
    <jaxws:features>
        <policy:policies>
            <wsp:Policy xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                <wsam:Addressing>
                    <wsp:Policy>
                        <wsam:NonAnonymousResponses/>
                    </wsp:Policy>
                </wsam:Addressing>
            </wsp:Policy>
        <policy:policies>
    </jaxws:features>
</jaxws:endpoint>
</beans>
```

CHAPTER 21. ENABLING RELIABLE MESSAGING

Abstract

Apache CXF supports WS-Reliable Messaging(WS-RM). This chapter explains how to enable and configure WS-RM in Apache CXF.

21.1. INTRODUCTION TO WS-RM

Overview

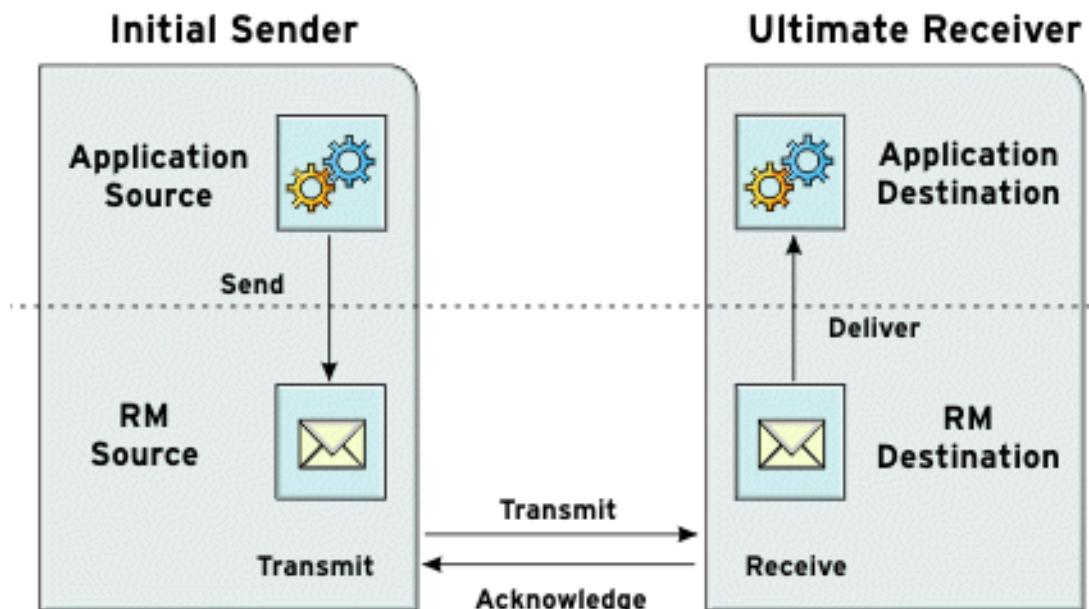
WS-ReliableMessaging (WS-RM) is a protocol that ensures the reliable delivery of messages in a distributed environment. It enables messages to be delivered reliably between distributed applications in the presence of software, system, or network failures.

For example, WS-RM can be used to ensure that the correct messages have been delivered across a network exactly once, and in the correct order.

How WS-RM works

WS-RM ensures the reliable delivery of messages between a source and a destination endpoint. The source is the initial sender of the message and the destination is the ultimate receiver, as shown in Figure 21.1, "Web Services Reliable Messaging".

Figure 21.1. Web Services Reliable Messaging



The flow of WS-RM messages can be described as follows:

1. The RM source sends a **CreateSequence** protocol message to the RM destination. This contains a reference for the endpoint that receives acknowledgements (the **wsrm:AcksTo** endpoint).
2. The RM destination sends a **CreateSequenceResponse** protocol message back to the RM source. This message contains the sequence ID for the RM sequence session.

3. The RM source adds an RM **Sequence** header to each message sent by the application source. This header contains the sequence ID and a unique message ID.
4. The RM source transmits each message to the RM destination.
5. The RM destination acknowledges the receipt of the message from the RM source by sending messages that contain the RM **SequenceAcknowledgement** header.
6. The RM destination delivers the message to the application destination in an exactly-once-in-order fashion.
7. The RM source retransmits a message that it has not yet received an acknowledgement. The first retransmission attempt is made after a base retransmission interval. Successive retransmission attempts are made, by default, at exponential back-off intervals or, alternatively, at fixed intervals. For more details, see [Section 21.5, "Configuring WS-RM"](#).

This entire process occurs symmetrically for both the request and the response message; that is, in the case of the response message, the server acts as the RM source and the client acts as the RM destination.

WS-RM delivery assurances

WS-RM guarantees reliable message delivery in a distributed environment, regardless of the transport protocol used. Either the source or the destination endpoint logs an error if reliable delivery can not be assured.

Supported specifications

Apache CXF supports the following versions of the WS-RM specification:

WS-ReliableMessaging 1.0

(Default) Corresponds to the [February 2005 submission version](#), which is now out of date. For reasons of backward compatibility, however, this version is used as the default.

Version 1.0 of WS-RM uses the following namespace:

`http://schemas.xmlsoap.org/ws/2005/02/rm/`

This version of WS-RM can be used with either of the following WS-Addressing versions:

`http://schemas.xmlsoap.org/ws/2004/08/addressing` (default)
`http://www.w3.org/2005/08/addressing`

Strictly speaking, in order to comply with the February 2005 submission version of WS-RM, you ought to use the first of these WS-Addressing versions (which is the default in Apache CXF). But most other Web service implementations have switched to the more recent WS-Addressing specification, so Apache CXF allows you to choose the WS-A version, to facilitate interoperability (see [Section 21.4, "Runtime Control"](#)).

WS-ReliableMessaging 1.1/1.2

Corresponds to the official [1.1/1.2 Web Services Reliable Messaging](#) specification.

Versions 1.1 and 1.2 of WS-RM uses the following namespace:

`http://docs.oasis-open.org/ws-rx/wsrm/200702`

The 1.1 and 1.2 versions of WS-RM use the following WS-Addressing version:

<http://www.w3.org/2005/08/addressing>

Selecting the WS-RM version

You can select which WS-RM specification version to use, as follows:

Server side

On the provider side, Apache CXF adapts to whichever version of WS-ReliableMessaging is used by the client and responds appropriately.

Client side

On the client side, the WS-RM version is determined either by the namespace that you use in the client configuration (see [Section 21.5, “Configuring WS-RM”](#)) or by overriding the WS-RM version at run time, using the runtime control options (see [Section 21.4, “Runtime Control”](#)).

21.2. WS-RM INTERCEPTORS

Overview

In Apache CXF, WS-RM functionality is implemented as interceptors. The Apache CXF runtime uses interceptors to intercept and work with the raw messages that are being sent and received. When a transport receives a message, it creates a message object and sends that message through an interceptor chain. If the application’s interceptor chain includes the WS-RM interceptors, the application can participate in reliable messaging sessions. The WS-RM interceptors handle the collection and aggregation of the message chunks. They also handle all of the acknowledgement and retransmission logic.

Apache CXF WS-RM Interceptors

The Apache CXF WS-RM implementation consists of four interceptors, which are described in [Table 21.1, “Apache CXF WS-ReliableMessaging Interceptors”](#).

Table 21.1. Apache CXF WS-ReliableMessaging Interceptors

Interceptor	Description
<code>org.apache.cxf.ws.rm.RMOutInterceptor</code>	<p>Deals with the logical aspects of providing reliability guarantees for outgoing messages.</p> <p>Responsible for sending the CreateSequence requests and waiting for their CreateSequenceResponse responses.</p> <p>Also responsible for aggregating the sequence properties—ID and message number—for an application message.</p>

Interceptor	Description
org.apache.cxf.ws.rm.RMInInterceptor	Responsible for intercepting and processing RM protocol messages and SequenceAcknowledgement messages that are piggybacked on application messages.
org.apache.cxf.ws.rm.RMCaptureInInterceptor	Caching incoming messages for persistent storage.
org.apache.cxf.ws.rm.RMDeliveryInterceptor	Assuring InOrder delivery of messages to the application.
org.apache.cxf.ws.rm.soap.RMSoapInterceptor	Responsible for encoding and decoding the reliability properties as SOAP headers.
org.apache.cxf.ws.rm.RetransmissionInterceptor	Responsible for creating copies of application messages for future resending.

Enabling WS-RM

The presence of the WS-RM interceptors on the interceptor chains ensures that WS-RM protocol messages are exchanged when necessary. For example, when intercepting the first application message on the outbound interceptor chain, the **RMOOutInterceptor** sends a **CreateSequence** request and waits to process the original application message until it receives the **CreateSequenceResponse** response. In addition, the WS-RM interceptors add the sequence headers to the application messages and, on the destination side, extract them from the messages. It is not necessary to make any changes to your application code to make the exchange of messages reliable.

For more information on how to enable WS-RM, see [Section 21.3, “Enabling WS-RM”](#).

Configuring WS-RM Attributes

You control sequence demarcation and other aspects of the reliable exchange through configuration. For example, by default Apache CXF attempts to maximize the lifetime of a sequence, thus reducing the overhead incurred by the out-of-band WS-RM protocol messages. To enforce the use of a separate sequence per application message configure the WS-RM source’s sequence termination policy (setting the maximum sequence length to **1**).

For more information on configuring WS-RM behavior, see [Section 21.5, “Configuring WS-RM”](#).

21.3. ENABLING WS-RM

Overview

To enable reliable messaging, the WS-RM interceptors must be added to the interceptor chains for both inbound and outbound messages and faults. Because the WS-RM interceptors use WS-Addressing, the WS-Addressing interceptors must also be present on the interceptor chains.

You can ensure the presence of these interceptors in one of two ways:

- [Explicitly](#), by adding them to the dispatch chains using Spring beans
- [Implicitly](#), using WS-Policy assertions, which cause the Apache CXF runtime to transparently add the interceptors on your behalf.

Spring beans: explicitly adding interceptors

To enable WS-RM add the WS-RM and WS-Addressing interceptors to the Apache CXF bus, or to a consumer or service endpoint using Spring bean configuration. This is the approach taken in the WS-RM sample that is found in the **InstallDir/samples/ws_rm** directory. The configuration file, **ws-rm.cxf**, shows the WS-RM and WS-Addressing interceptors being added one-by-one as Spring beans (see [Example 21.1, "Enabling WS-RM Using Spring Beans"](#)).

Example 21.1. Enabling WS-RM Using Spring Beans

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="mapAggregator" class="org.apache.cxf.ws.addressing.MAPAggregator"/>
    <bean id="mapCodec" class="org.apache.cxf.ws.addressing.soap.MAPCodec"/>
    <bean id="rmLogicalOut" class="org.apache.cxf.ws.rm.RMOutInterceptor">
        <property name="bus" ref="cxf"/>
    </bean>
    <bean id="rmLogicalIn" class="org.apache.cxf.ws.rm.RMInInterceptor">
        <property name="bus" ref="cxf"/>
    </bean>
    <bean id="rmCodec" class="org.apache.cxf.ws.rm.soap.RMSoapInterceptor"/>
    <bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl">
        <property name="inInterceptors">
            <list>
                <ref bean="mapAggregator"/>
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalIn"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
        <property name="inFaultInterceptors">
            <list>
                <ref bean="mapAggregator"/>
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalIn"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
        <property name="outInterceptors">
            <list>
                <ref bean="mapAggregator"/>
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalOut"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
        <property name="outFaultInterceptors">
```

```

<list>
    <ref bean="mapAggregator">
    <ref bean="mapCodec"/>
    <ref bean="rmLogicalOut"/>
    <ref bean="rmCodec"/>
</list>
</property>
</bean>
</beans>

```

The code shown in [Example 21.1, “Enabling WS-RM Using Spring Beans”](#) can be explained as follows:

A Apache CXF configuration file is a Spring XML file. You must include an opening Spring **beans** element that declares the namespaces and schema files for the child elements that are encapsulated by the **beans** element.

Configures each of the WS-Addressing interceptors—**MAPAggregator** and **MAPCodec**. For more information on WS-Addressing, see [Chapter 20, Deploying WS-Addressing](#).

Configures each of the WS-RM interceptors—**RMOutInterceptor**, **RMIInInterceptor**, and **RMSoapInterceptor**.

Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for inbound messages.

Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for inbound faults.

Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for outbound messages.

Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for outbound faults.

WS-Policy framework: implicitly adding interceptors

The WS-Policy framework provides the infrastructure and APIs that allow you to use WS-Policy. It is compliant with the November 2006 draft publications of the [Web Services Policy 1.5–Framework](#) and [Web Services Policy 1.5–Attachment](#) specifications.

To enable WS-RM using the Apache CXF WS-Policy framework, do the following:

1. Add the policy feature to your client and server endpoint. [Example 21.2, “Configuring WS-RM using WS-Policy”](#) shows a reference bean nested within a **jaxws:feature** element. The reference bean specifies the **AddressingPolicy**, which is defined as a separate element within the same configuration file.

Example 21.2. Configuring WS-RM using WS-Policy

```

<jaxws:client>
    <jaxws:features>
        <ref bean="AddressingPolicy"/>
    </jaxws:features>
</jaxws:client>
<wsp:Policy wsu:id="AddressingPolicy"
    xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
    <wsam:Addressing>
        <wsp:Policy>
            <wsam:NonAnonymousResponses/>

```

```
</wsp:Policy>
</wsam:Addressing>
</wsp:Policy>
```

2. Add a reliable messaging policy to the **wsdl:service** element—or any other WSDL element that can be used as an attachment point for policy or policy reference elements—to your WSDL file, as shown in [Example 21.3, “Adding an RM Policy to Your WSDL File”](#).

Example 21.3. Adding an RM Policy to Your WSDL File

```
<wsp:Policy wsu:id="RM"
  xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd">
  <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
    <wsp:Policy/>
  </wsam:Addressing>
  <wsrmp:RMAssertion
    xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
    <wsrmp:BaseRetransmissionInterval Milliseconds="10000"/>
  </wsrmp:RMAssertion>
</wsp:Policy>
...
<wsdl:service name="ReliableGreeterService">
  <wsdl:port binding="tns:GreeterSOAPBinding" name="GreeterPort">
    <soap:address location="http://localhost:9020/SoapContext/GreeterPort"/>
    <wsp:PolicyReference URI="#RM" xmlns:wsp="http://www.w3.org/2006/07/ws-
policy"/>
  </wsdl:port>
</wsdl:service>
```

21.4. RUNTIME CONTROL

Overview

Several message context property values can be set in client code to control WS-RM at runtime, with key values defined by public constants in the **org.apache.cxf.ws.rm.RMManager** class.

Runtime control options

The following table lists the keys defined by the **org.apache.cxf.ws.rm.RMManager** class.

Key	Description
WSRM_VERSION_PROPERTY	String WS-RM version namespace (http://schemas.xmlsoap.org/ws/2005/02/rm/ or http://docs.oasis-open.org/ws-rx/wsrm/200702).

Key	Description
WSRM_WSA_VERSION_PROPERTY	String WS-Addressing version namespace (http://schemas.xmlsoap.org/ws/2004/08/addressing or http://www.w3.org/2005/08/addressing) - this property is ignored unless you're using the http://schemas.xmlsoap.org/ws/2005/02/rm/ RM namespace).
WSRM_LAST_MESSAGE_PROPERTY	Boolean value true to tell the WS-RM code that the last message is being sent, allowing the code to close the WS-RM sequence and release resources (as of the 3.0.0 version of CXF, the WS-RM will close the RM sequence by default, when you close your client).
WSRM_INACTIVITY_TIMEOUT_PROPERTY	Long inactivity timeout in milliseconds.
WSRM_RETRANSMISSION_INTERVAL_PROPERTY	Long base retransmission interval in milliseconds.
WSRM_EXPONENTIAL_BACKOFF_PROPERTY	Boolean exponential back-off flag.
WSRM_ACKNOWLEDGEMENT_INTERVAL_PROPERTY	Long acknowledgement interval in milliseconds.

Controlling WS-RM through JMX

You can also monitor and control many aspects of WS-RM using the JMX Management features of Apache CXF. The full list of JMX operations is defined by **org.apache.cxf.ws.rm.ManagedRMMManager** and **org.apache.cxf.ws.rm.ManagedRMEndpoint**, but these operations include viewing the current RM state down to the individual message level. You can also use JMX to close or terminate a WS-RM sequence, and to receive notification of when previously-sent messages are acknowledged by the remote RM endpoint.

Example of JMX control

For example, if you have the JMX server enabled in your client configuration, you could use the following code to track the last acknowledgement number received:

```
// Java
private static class AcknowledgementListener implements NotificationListener {
    private volatile long lastAcknowledgement;

    @Override
    public void handleNotification(Notification notification, Object handback) {
        if (notification instanceof AcknowledgementNotification) {
            AcknowledgementNotification ack = (AcknowledgementNotification)notification;
            lastAcknowledgement = ack.getMessageNumber();
        }
    }
}
```

```

// initialize client
...
// attach to JMX bean for notifications
// NOTE: you must have sent at least one message to initialize RM before executing this code
Endpoint ep = ClientProxy.getClient(client).getEndpoint();
InstrumentationManager im = busgetExtension(InstrumentationManager.class);
MBeanServer mbs = im.getMBeanServer();
RMManager clientManager = bus.getExtension(RMManager.class);
ObjectName name = RMUtils.getManagedObjectName(clientManager, ep);
System.out.println("Looking for endpoint name " + name);
AcknowledgementListener listener = new AcknowledgementListener();
mbs.addNotificationListener(name, listener, null, null);

// send messages using RM with acknowledgement status reported to listener
...

```

21.5. CONFIGURING WS-RM

21.5.1. Configuring Apache CXF-Specific WS-RM Attributes

Overview

To configure the Apache CXF-specific attributes, use the **rmManager** Spring bean. Add the following to your configuration file:

- The <http://cxf.apache.org/ws/rm/manager> namespace to your list of namespaces.
- An **rmManager** Spring bean for the specific attribute that you want to configure.

[Example 21.4, “Configuring Apache CXF-Specific WS-RM Attributes”](#) shows a simple example.

Example 21.4. Configuring Apache CXF-Specific WS-RM Attributes

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/ws/rm/manager http://cxf.apache.org/schemas/configuration/wsrm-
manager.xsd">
...
<wsrm-mgr:rmManager>
<!--
...Your configuration goes here
-->
</wsrm-mgr:rmManager>

```

Children of the rmManager Spring bean

[Table 21.2, “Children of the rmManager Spring Bean”](#) shows the child elements of the **rmManager** Spring bean, defined in the <http://cxf.apache.org/ws/rm/manager> namespace.

Table 21.2. Children of the rmManager Spring Bean

Element	Description
RMAssertion	An element of type RMAssertion
deliveryAssurance	An element of type DeliveryAssuranceType that describes the delivery assurance that should apply
sourcePolicy	An element of type SourcePolicyType that allows you to configure details of the RM source
destinationPolicy	An element of type DestinationPolicyType that allows you to configure details of the RM destination

Example

For an example, see [the section called “Maximum unacknowledged messages threshold”](#).

21.5.2. Configuring Standard WS-RM Policy Attributes

Overview

You can configure standard WS-RM policy attributes in one of the following ways:

- [the section called “RMAssertion in rmManager Spring bean”](#)
- [the section called “Policy within a feature”](#)
- [the section called “WSDL file”](#)
- [the section called “External attachment”](#)

WS-Policy RMAssertion Children

Table 21.3, “Children of the WS-Policy RMAssertion Element” shows the elements defined in the <http://schemas.xmlsoap.org/ws/2005/02/rm/policy> namespace:

Table 21.3. Children of the WS-Policy RMAssertion Element

Name	Description
InactivityTimeout	Specifies the amount of time that must pass without receiving a message before an endpoint can consider an RM sequence to have been terminated due to inactivity.

Name	Description
BaseRetransmissionInterval	Sets the interval within which an acknowledgement must be received by the RM Source for a given message. If an acknowledgement is not received within the time set by the BaseRetransmissionInterval , the RM Source will retransmit the message.
ExponentialBackoff	Indicates the retransmission interval will be adjusted using the commonly known exponential backoff algorithm (Tanenbaum). For more information, see Computer Networks , Andrew S. Tanenbaum, Prentice Hall PTR, 2003.
AcknowledgementInterval	In WS-RM, acknowledgements are sent on return messages or sent stand-alone. If a return message is not available to send an acknowledgement, an RM Destination can wait for up to the acknowledgement interval before sending a stand-alone acknowledgement. If there are no unacknowledged messages, the RM Destination can choose not to send an acknowledgement.

More detailed reference information

For more detailed reference information, including descriptions of each element's sub-elements and attributes, please refer to <http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd>.

RMAssertion in rmManager Spring bean

You can configure standard WS-RM policy attributes by adding an **RMAssertion** within a Apache CXF **rmManager** Spring bean. This is the best approach if you want to keep all of your WS-RM configuration in the same configuration file; that is, if you want to configure Apache CXF-specific attributes and standard WS-RM policy attributes in the same file.

For example, the configuration in [Example 21.5, “Configuring WS-RM Attributes Using an RMAssertion in an rmManager Spring Bean”](#) shows:

- A standard WS-RM policy attribute, **BaseRetransmissionInterval**, configured using an **RMAssertion** within an **rmManager** Spring bean.
- An Apache CXF-specific RM attribute, **intraMessageThreshold**, configured in the same configuration file.

Example 21.5. Configuring WS-RM Attributes Using an RMAssertion in an rmManager Spring Bean

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
       xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
...
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
```

```

<wsrm-policy:RMAssertion>
    <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
</wsrm-policy:RMAssertion>
<wsrm-mgr:destinationPolicy>
    <wsrm-mgr:acksPolicy intraMessageThreshold="0" />
</wsrm-mgr:destinationPolicy>
</wsrm-mgr:rmManager>
</beans>

```

Policy within a feature

You can configure standard WS-RM policy attributes within features, as shown in [Example 21.6, "Configuring WS-RM Attributes as a Policy within a Feature"](#).

Example 21.6. Configuring WS-RM Attributes as a Policy within a Feature

```

<xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:wsa="http://cxfr.apache.org/ws/addressing"
       xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
       xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
       xmlns:jaxws="http://cxfr.apache.org/jaxws"
       xsi:schemaLocation="
           http://www.w3.org/2006/07/ws-policy http://www.w3.org/2006/07/ws-policy.xsd
           http://cxfr.apache.org/ws/addressing http://cxfr.apache.org/schema/ws/addressing.xsd
           http://cxfr.apache.org/jaxws http://cxfr.apache.org/schemas/jaxws.xsd
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">
    <jaxws:endpoint name="{http://cxfr.apache.org/greeter_control}GreeterPort"
        createdFromAPI="true">
        <jaxws:features>
            <wsp:Policy>
                <wsrm:RMAssertion
                    xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
                        <wsrm:AcknowledgementInterval Milliseconds="200" />
                    </wsrm:RMAssertion>
                    <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                        <wsp:Policy>
                            <wsam:NonAnonymousResponses/>
                        </wsp:Policy>
                    </wsam:Addressing>
                </wsp:Policy>
            </jaxws:features>
        </jaxws:endpoint>
    </beans>

```

WSDL file

If you use the WS-Policy framework to enable WS-RM, you can configure standard WS-RM policy attributes in a WSDL file. This is a good approach if you want your service to interoperate and use WS-RM seamlessly with consumers deployed to other policy-aware Web services stacks.

For an example, see [the section called “WS-Policy framework: implicitly adding interceptors”](#) where the base retransmission interval is configured in the WSDL file.

External attachment

You can configure standard WS-RM policy attributes in an external attachment file. This is a good approach if you cannot, or do not want to, change your WSDL file.

[Example 21.7, “Configuring WS-RM in an External Attachment”](#) shows an external attachment that enables both WS-A and WS-RM (base retransmission interval of 30 seconds) for a specific EPR.

Example 21.7. Configuring WS-RM in an External Attachment

```
<attachments xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
  xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <wsp:PolicyAttachment>
    <wsp:AppliesTo>
      <wsa:EndpointReference>
        <wsa:Address>http://localhost:9020/SoapContext/GreeterPort</wsa:Address>
      </wsa:EndpointReference>
    </wsp:AppliesTo>
    <wsp:Policy>
      <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
        <wsp:Policy/>
      </wsam:Addressing>
      <wsrmp:RMAssertion xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
        <wsrmp:BaseRetransmissionInterval Milliseconds="30000"/>
      </wsrmp:RMAssertion>
    </wsp:Policy>
  </wsp:PolicyAttachment>
</attachments>
```

21.5.3. WS-RM Configuration Use Cases

Overview

This subsection focuses on configuring WS-RM attributes from a use case point of view. Where an attribute is a standard WS-RM policy attribute, defined in the <http://schemas.xmlsoap.org/ws/2005/02/rm/policy/> namespace, only the example of setting it in an **RMAssertion** within an **rmManager** Spring bean is shown. For details of how to set such attributes as a policy within a feature; in a WSDL file, or in an external attachment, see [Section 21.5.2, “Configuring Standard WS-RM Policy Attributes”](#).

The following use cases are covered:

- [the section called “Base retransmission interval”](#)
- [the section called “Exponential backoff for retransmission”](#)
- [the section called “Acknowledgement interval”](#)

- the section called “Maximum unacknowledged messages threshold”
- the section called “Maximum length of an RM sequence”
- the section called “Message delivery assurance policies”

Base retransmission interval

The **BaseRetransmissionInterval** element specifies the interval at which an RM source retransmits a message that has not yet been acknowledged. It is defined in the <http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd> schema file. The default value is 3000 milliseconds.

[Example 21.8, “Setting the WS-RM Base Retransmission Interval”](#) shows how to set the WS-RM base retransmission interval.

Example 21.8. Setting the WS-RM Base Retransmission Interval

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

Exponential backoff for retransmission

The **ExponentialBackoff** element determines if successive retransmission attempts for an unacknowledged message are performed at exponential intervals.

The presence of the **ExponentialBackoff** element enables this feature. An exponential backoff ratio of **2** is used by default. **ExponentialBackoff** is a flag. When the element is present, exponential backoff is enabled. When the element is absent, exponential backoff is disabled. No value is required.

[Example 21.9, “Setting the WS-RM Exponential Backoff Property”](#) shows how to set the WS-RM exponential backoff for retransmission.

Example 21.9. Setting the WS-RM Exponential Backoff Property

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:ExponentialBackoff/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

Acknowledgement interval

The **AcknowledgementInterval** element specifies the interval at which the WS-RM destination sends asynchronous acknowledgements. These are in addition to the synchronous acknowledgements that it sends on receipt of an incoming message. The default asynchronous acknowledgement interval is **0** milliseconds. This means that if the **AcknowledgementInterval** is not configured to a specific value, acknowledgements are sent immediately (that is, at the first available opportunity).

Asynchronous acknowledgements are sent by the RM destination only if both of the following conditions are met:

- The RM destination is using a non-anonymous **wsrm:acksTo** endpoint.
- The opportunity to piggyback an acknowledgement on a response message does not occur before the expiry of the acknowledgement interval.

[Example 21.10, "Setting the WS-RM Acknowledgement Interval"](#) shows how to set the WS-RM acknowledgement interval.

Example 21.10. Setting the WS-RM Acknowledgement Interval

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:AcknowledgementInterval Milliseconds="2000"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

Maximum unacknowledged messages threshold

The **maxUnacknowledged** attribute sets the maximum number of unacknowledged messages that can accrue per sequence before the sequence is terminated.

[Example 21.11, "Setting the WS-RM Maximum Unacknowledged Message Threshold"](#) shows how to set the WS-RM maximum unacknowledged messages threshold.

Example 21.11. Setting the WS-RM Maximum Unacknowledged Message Threshold

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:sourcePolicy>
        <wsrm-mgr:sequenceTerminationPolicy maxUnacknowledged="20" />
    </wsrm-mgr:sourcePolicy>
</wsrm-mgr:reliableMessaging>
</beans>
```

Maximum length of an RM sequence

The **maxLength** attribute sets the maximum length of a WS-RM sequence. The default value is **0**, which means that the length of a WS-RM sequence is unbound.

When this attribute is set, the RM endpoint creates a new RM sequence when the limit is reached, and after receiving all of the acknowledgements for the previously sent messages. The new message is sent using a newsequence.

[Example 21.12, "Setting the Maximum Length of a WS-RM Message Sequence"](#) shows how to set the maximum length of an RM sequence.

Example 21.12. Setting the Maximum Length of a WS-RM Message Sequence

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:sourcePolicy>
        <wsrm-mgr:sequenceTerminationPolicy maxLength="100" />
    </wsrm-mgr:sourcePolicy>
</wsrm-mgr:reliableMessaging>
</beans>
```

Message delivery assurance policies

You can configure the RM destination to use the following delivery assurance policies:

- **AtMostOnce** – The RM destination delivers the messages to the application destination only once. If a message is delivered more than once an error is raised. It is possible that some messages in a sequence may not be delivered.
- **AtLeastOnce** – The RM destination delivers the messages to the application destination at least once. Every message sent will be delivered or an error will be raised. Some messages might be delivered more than once.
- **InOrder** – The RM destination delivers the messages to the application destination in the order that they are sent. This delivery assurance can be combined with the **AtMostOnce** or **AtLeastOnce** assurances.

[Example 21.13, "Setting the WS-RM Message Delivery Assurance Policy"](#) shows how to set the WS-RM message delivery assurance.

Example 21.13. Setting the WS-RM Message Delivery Assurance Policy

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:deliveryAssurance>
        <wsrm-mgr:AtLeastOnce />
    </wsrm-mgr:deliveryAssurance>
</wsrm-mgr:reliableMessaging>
</beans>
```

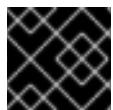
21.6. CONFIGURING WS-RM PERSISTENCE

Overview

The Apache CXF WS-RM features already described in this chapter provide reliability for cases such as network failures. WS-RM persistence provides reliability across other types of failure such as an RM source or an RM destination crash.

WS-RM persistence involves storing the state of the various RM endpoints in persistent storage. This enables the endpoints to continue sending and receiving messages when they are reincarnated.

Apache CXF enables WS-RM persistence in a configuration file. The default WS-RM persistence store is JDBC-based. For convenience, Apache CXF includes Derby for out-of-the-box deployment. In addition, the persistent store is also exposed using a Java API. To implement your own persistence mechanism, you can implement one using this API with your preferred DB.



IMPORTANT

WS-RM persistence is supported for oneway calls only, and it is disabled by default.

How it works

Apache CXF WS-RM persistence works as follows:

- At the RM source endpoint, an outgoing message is persisted before transmission. It is evicted from the persistent store after the acknowledgement is received.
- After a recovery from crash, it recovers the persisted messages and retransmits until all the messages have been acknowledged. At that point, the RM sequence is closed.
- At the RM destination endpoint, an incoming message is persisted, and upon a successful store, the acknowledgement is sent. When a message is successfully dispatched, it is evicted from the persistent store.
- After a recovery from a crash, it recovers the persisted messages and dispatches them. It also brings the RM sequence to a state where new messages are accepted, acknowledged, and delivered.

Enabling WS-persistence

To enable WS-RM persistence, you must specify the object implementing the persistent store for WS-RM. You can develop your own or you can use the JDBC based store that comes with Apache CXF.

The configuration shown in [Example 21.14, "Configuration for the Default WS-RM Persistence Store"](#) enables the JDBC-based store that comes with Apache CXF.

Example 21.14. Configuration for the Default WS-RM Persistence Store

```
<bean id="RMTxStore" class="org.apache.cxf.ws.rm.persistence.jdbc.RMTxStore"/>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <property name="store" ref="RMTxStore"/>
</wsrm-mgr:rmManager>
```

Configuring WS-persistence

The JDBC-based store that comes with Apache CXF supports the properties shown in [Table 21.4, "JDBC Store Properties"](#).

Table 21.4. JDBC Store Properties

Attribute Name	Type	Default Setting
driverClassName	String	org.apache.derby.jdbc.EmbeddedDriver
userName	String	null
passWord	String	null
url	String	jdbc:derby:rmdb;create=true

The configuration shown in [Example 21.15, "Configuring the JDBC Store for WS-RM Persistence"](#) enables the JDBC-based store that comes with Apache CXF, while setting the **driverClassName** and **url** to non-default values.

Example 21.15. Configuring the JDBC Store for WS-RM Persistence

```
<bean id="RMTxStore" class="org.apache.cxf.ws.rm.persistence.jdbc.RMTxStore">
    <property name="driverClassName" value="com.acme.jdbc.Driver"/>
    <property name="url" value="jdbc:acme:rmdb;create=true"/>
</bean>
```

CHAPTER 22. ENABLING HIGH AVAILABILITY

Abstract

This chapter explains how to enable and configure high availability in the Apache CXF runtime.

22.1. INTRODUCTION TO HIGH AVAILABILITY

Overview

Scalable and reliable applications require high availability to avoid any single point of failure in a distributed system. You can protect your system from single points of failure using *replicated services*.

A replicated service is comprised of multiple instances, or *replicas*, of the same service. Together these act as a single logical service. Clients invoke requests on the replicated service, and Apache CXF delivers the requests to one of the member replicas. The routing to a replica is transparent to the client.

HA with static failover

Apache CXF supports high availability (HA) with static failover in which replica details are encoded in the service WSDL file. The WSDL file contains multiple ports, and can contain multiple hosts, for the same service. The number of replicas in the cluster remains static as long as the WSDL file remains unchanged. Changing the cluster size involves editing the WSDL file.

22.2. ENABLING HA WITH STATIC FAILOVER

Overview

To enable HA with static failover, you must do the following:

1. [the section called “Encode replica details in your service WSDL file”](#)
2. [the section called “Add the clustering feature to your client configuration”](#)

Encode replica details in your service WSDL file

You must encode the details of the replicas in your cluster in your service WSDL file. [Example 22.1, “Enabling HA with Static Failover: WSDL File”](#) shows a WSDL file extract that defines a service cluster of three replicas.

Example 22.1. Enabling HA with Static Failover: WSDL File

```
<wsdl:service name="ClusteredService">
    <wsdl:port binding="tns:Greeter_SOAPBinding" name="Replica1">
        <soap:address location="http://localhost:9001/SoapContext/Replica1"/>
    </wsdl:port>

    <wsdl:port binding="tns:Greeter_SOAPBinding" name="Replica2">
        <soap:address location="http://localhost:9002/SoapContext/Replica2"/>
    </wsdl:port>

    <wsdl:port binding="tns:Greeter_SOAPBinding" name="Replica3">
```

```

<soap:address location="http://localhost:9003/SoapContext/Replica3"/>
</wsdl:port>

</wsdl:service>

```

The WSDL extract shown in [Example 22.1, “Enabling HA with Static Failover: WSDL File”](#) can be explained as follows:

Defines a service, **ClusterService**, which is exposed on three ports:

1. **Replica1**
2. **Replica2**
3. **Replica3**

Defines **Replica1** to expose the **ClusterService** as a SOAP over HTTP endpoint on port **9001**.

Defines **Replica2** to expose the **ClusterService** as a SOAP over HTTP endpoint on port **9002**.

Defines **Replica3** to expose the **ClusterService** as a SOAP over HTTP endpoint on port **9003**.

Add the clustering feature to your client configuration

In your client configuration file, add the clustering feature as shown in [Example 22.2, “Enabling HA with Static Failover: Client Configuration”](#).

Example 22.2. Enabling HA with Static Failover: Client Configuration

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:clustering="http://cxf.apache.org/clustering"
       xsi:schemaLocation="http://cxf.apache.org/jaxws
                           http://cxf.apache.org/schemas/jaxws.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <jaxws:client name="{http://apache.org/hello_world_soap_http}Replica1"
                  createdFromAPI="true">
        <jaxws:features>
            <clustering:failover/>
        </jaxws:features>
    </jaxws:client>

    <jaxws:client name="{http://apache.org/hello_world_soap_http}Replica2"
                  createdFromAPI="true">
        <jaxws:features>
            <clustering:failover/>
        </jaxws:features>
    </jaxws:client>

    <jaxws:client name="{http://apache.org/hello_world_soap_http}Replica3"
                  createdFromAPI="true">
        <jaxws:features>
            <clustering:failover/>
        </jaxws:features>
    </jaxws:client>

```

```

    createdFromAPI="true">
<jaxws:features>
    <clustering:failover/>
</jaxws:features>
</jaxws:client>

</beans>

```

22.3. CONFIGURING HA WITH STATIC FAILOVER

Overview

By default, HA with static failover uses a sequential strategy when selecting a replica service if the original service with which a client is communicating becomes unavailable, or fails. The sequential strategy selects a replica service in the same sequential order every time it is used. Selection is determined by Apache CXF's internal service model and results in a deterministic failover pattern.

Configuring a random strategy

You can configure HA with static failover to use a random strategy instead of the sequential strategy when selecting a replica. The random strategy selects a random replica service each time a service becomes unavailable, or fails. The choice of failover target from the surviving members in a cluster is entirely random.

To configure the random strategy, add the configuration shown in [Example 22.3, “Configuring a Random Strategy for Static Failover”](#) to your client configuration file.

Example 22.3. Configuring a Random Strategy for Static Failover

```

<beans ...>
    <bean id="Random" class="org.apache.cxf.clustering.RandomStrategy"/>

    <jaxws:client name="{http://apache.org/hello_world_soap_http}Replica3"
        createdFromAPI="true">
        <jaxws:features>
            <clustering:failover>
                <clustering:strategy>
                    <ref bean="Random"/>
                </clustering:strategy>
            </clustering:failover>
        </jaxws:features>
    </jaxws:client>
</beans>

```

The configuration shown in [Example 22.3, “Configuring a Random Strategy for Static Failover”](#) can be explained as follows:

Defines a **Random** bean and implementation class that implements the random strategy.

Specifies that the random strategy is used when selecting a replica.

CHAPTER 23. APACHE CXF BINDING IDS

TABLE OF BINDING IDS

Table 23.1. Binding IDs for Message Bindings

Binding	ID
CORBA	http://cxf.apache.org/bindings/corba
HTTP/REST	http://apache.org/cxf/binding/http
SOAP 1.1	http://schemas.xmlsoap.org/wsdl/soap/http
SOAP 1.1 w/ MTOM	http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true
SOAP 1.2	http://www.w3.org/2003/05/soap/bindings/HTTP/
SOAP 1.2 w/ MTOM	http://www.w3.org/2003/05/soap/bindings/HTTP/?mtom=true
XML	http://cxf.apache.org/bindings/xformat

APPENDIX A. USING THE MAVEN OSGI TOOLING

Abstract

Manually creating a bundle, or a collection of bundles, for a large project can be cumbersome. The Maven bundle plug-in makes the job easier by automating the process and providing a number of shortcuts for specifying the contents of the bundle manifest.

A.1. THE MAVEN BUNDLE PLUG-IN

The Red Hat Fuse OSGi tooling uses the [Maven bundle plug-in](#) from Apache Felix. The bundle plug-in is based on the `bnd` tool from Peter Kriens. It automates the construction of OSGi bundle manifests by introspecting the contents of the classes being packaged in the bundle. Using the knowledge of the classes contained in the bundle, the plug-in can calculate the proper values to populate the **Import-Packages** and the **Export-Package** properties in the bundle manifest. The plug-in also has default values that are used for other required properties in the bundle manifest.

To use the bundle plug-in, do the following:

1. [Section A.2, “Setting up a Red Hat Fuse OSGi project”](#) the bundle plug-in to your project’s POM file.
2. [Section A.3, “Configuring the Bundle Plug-In”](#) the plug-in to correctly populate your bundle’s manifest.

A.2. SETTING UP A RED HAT FUSE OSGI PROJECT

Overview

A Maven project for building an OSGi bundle can be a simple single level project. It does not require any sub-projects. However, it does require that you do the following:

1. [Add](#) the bundle plug-in to your POM.
2. [Instruct](#) Maven to package the results as an OSGi bundle.



NOTE

There are several Maven archetypes you can use to set up your project with the appropriate settings.

Directory structure

A project that constructs an OSGi bundle can be a single level project. It only requires that you have a top-level POM file and a **src** folder. As in all Maven projects, you place all Java source code in the **src/java** folder, and you place any non-Java resources in the **src/resources** folder.

Non-Java resources include Spring configuration files, JBI endpoint configuration files, and WSDL contracts.



NOTE

Red Hat Fuse OSGi projects that use Apache CXF, Apache Camel, or another Spring configured bean also include a **beans.xml** file located in the **src/resources/META-INF/spring** folder.

Adding a bundle plug-in

Before you can use the bundle plug-in you must add a dependency on Apache Felix. After you add the dependency, you can add the bundle plug-in to the plug-in portion of the POM.

[Example A.1, “Adding an OSGi bundle plug-in to a POM”](#) shows the POM entries required to add the bundle plug-in to your project.

Example A.1. Adding an OSGi bundle plug-in to a POM

```

...
<dependencies>
  <dependency>
    <groupId>org.apache.felix</groupId>
    <artifactId>org.osgi.core</artifactId>
    <version>1.0.0</version>
  </dependency>
...
</dependencies>
...
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <configuration>
        <instructions>
          <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName>
          <Import-Package>*,org.apache.camel.osgi</Import-Package>
          <Private-Package>org.apache.servicemix.examples.camel</Private-Package>
        </instructions>
      </configuration>
    </plugin>
  </plugins>
</build>
...

```

The entries in [Example A.1, “Adding an OSGi bundle plug-in to a POM”](#) do the following:

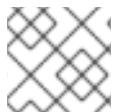
Adds the dependency on Apache Felix

Adds the bundle plug-in to your project

Configures the plug-in to use the project’s artifact ID as the bundle’s symbolic name

Configures the plug-in to include all Java packages imported by the bundled classes; also imports the **org.apache.camel.osgi** package

Configures the plug-in to bundle the listed class, but not to include them in the list of exported packages



NOTE

Edit the configuration to meet the requirements of your project.

For more information on configuring the bundle plug-in, see [Section A.3, “Configuring the Bundle Plug-In”](#).

Activating a bundle plug-in

To have Maven use the bundle plug-in, instruct it to package the results of the project as a bundle. Do this by setting the POM file’s **packaging** element to **bundle**.

Useful Maven archetypes

There are several Maven archetypes available to generate a project that is preconfigured to use the bundle plug-in:

- [the section called “Spring OSGi archetype”](#)
- [the section called “Apache CXF code-first archetype”](#)
- [the section called “Apache CXF wsdl-first archetype”](#)
- [the section called “Apache Camel archetype”](#)

Spring OSGi archetype

The Spring OSGi archetype creates a generic project for building an OSGi project using Spring DM, as shown:

```
org.springframework.osgi/spring-bundle-osgi-archetype/1.1.2
```

You invoke the archetype using the following command:

```
mvn archetype:generate -DarchetypeGroupId=org.springframework.osgi -  
DarchetypeArtifactId=spring-osgi-bundle-archetype -DarchetypeVersion=1.1.2 -DgroupId=groupId -  
DartifactId=artifactId -Dversion=version
```

Apache CXF code-first archetype

The Apache CXF code-first archetype creates a project for building a service from Java, as shown:

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-code-first-archetype/2010.02.0-fuse-02-00
```

You invoke the archetype using the following command:

```
mvn archetype:generate -DarchetypeGroupId=org.apache.servicemix.tooling -  
DarchetypeArtifactId=servicemix-osgi-cxf-code-first-archetype -DarchetypeVersion=2010.02.0-fuse-  
02-00 -DgroupId=groupId -DartifactId=artifactId -Dversion=version
```

Apache CXF wsdl-first archetype

The Apache CXF wsdl-first archetype creates a project for creating a service from WSDL, as shown:

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-wsdl-first-archetype/2010.02.0-fuse-02-00
```

You invoke the archetype using the following command:

```
mvn archetype:generate -DarchetypeGroupId=org.apache.servicemix.tooling -  
DarchetypeArtifactId=servicemix-osgi-cxf-wsdl-first-archetype -DarchetypeVersion=2010.02.0-fuse-  
02-00 -DgroupId=groupId -DartifactId=artifactId -Dversion=version
```

Apache Camel archetype

The Apache Camel archetype creates a project for building a route that is deployed into Red Hat Fuse, as shown:

```
org.apache.servicemix.tooling/servicemix-osgi-camel-archetype/2010.02.0-fuse-02-00
```

You invoke the archetype using the following command:

```
mvn archetype:generate -DarchetypeGroupId=org.apache.servicemix.tooling -  
DarchetypeArtifactId=servicemix-osgi-camel-archetype -DarchetypeVersion=2010.02.0-fuse-02-00 -  
DgroupId=groupId -DartifactId=artifactId -Dversion=version
```

A.3. CONFIGURING THE BUNDLE PLUG-IN

Overview

A bundle plug-in requires very little information to function. All of the required properties use default settings to generate a valid OSGi bundle.

While you can create a valid bundle using just the default values, you will probably want to modify some of the values. You can specify most of the properties inside the plug-in's **instructions** element.

Configuration properties

Some of the commonly used configuration properties are:

- [Bundle-SymbolicName](#)
- [Bundle-Name](#)
- [Bundle-Version](#)
- [Export-Package](#)
- [Private-Package](#)
- [Import-Package](#)

Setting a bundle's symbolic name

By default, the bundle plug-in sets the value for the **Bundle-SymbolicName** property to *groupId* + "." + *artifactId*, with the following exceptions:

- If *groupId* has only one section (no dots), the first package name with classes is returned.
For example, if the group Id is **commons-logging:commons-logging**, the bundle's symbolic name is **org.apache.commons.logging**.
- If *artifactId* is equal to the last section of *groupId*, then *groupId* is used.
For example, if the POM specifies the group ID and artifact ID as **org.apache.maven:maven**, the bundle's symbolic name is **org.apache.maven**.
- If *artifactId* starts with the last section of *groupId*, that portion is removed.
For example, if the POM specifies the group ID and artifact ID as **org.apache.maven:maven-core**, the bundle's symbolic name is **org.apache.maven.core**.

To specify your own value for the bundle's symbolic name, add a **Bundle-SymbolicName** child in the plug-in's **instructions** element, as shown in [Example A.2, "Setting a bundle's symbolic name"](#).

Example A.2. Setting a bundle's symbolic name

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
    <instructions>
      <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
      ...
    </instructions>
  </configuration>
</plugin>
```

Setting a bundle's name

By default, a bundle's name is set to **\${project.name}**.

To specify your own value for the bundle's name, add a **Bundle-Name** child to the plug-in's **instructions** element, as shown in [Example A.3, "Setting a bundle's name"](#).

Example A.3. Setting a bundle's name

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
    <instructions>
      <Bundle-Name>JoeFred</Bundle-Name>
      ...
    </instructions>
  </configuration>
</plugin>
```

Setting a bundle's version

By default, a bundle's version is set to `${project.version}`. Any dashes (-) are replaced with dots (.) and the number is padded up to four digits. For example, **4.2-SNAPSHOT** becomes **4.2.0.SNAPSHOT**.

To specify your own value for the bundle's version, add a **Bundle-Version** child to the plug-in's **instructions** element, as shown in [Example A.4, "Setting a bundle's version"](#).

Example A.4. Setting a bundle's version

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
    <instructions>
      <Bundle-Version>1.0.3.1</Bundle-Version>
      ...
    </instructions>
  </configuration>
</plugin>
```

Specifying exported packages

By default, the OSGi manifest's **Export-Package** list is populated by all of the packages in your local Java source code (under `src/main/java`), except for the default package, `.`, and any packages containing `.impl` or `.internal`.



IMPORTANT

If you use a **Private-Package** element in your plug-in configuration and you do not specify a list of packages to export, the default behavior includes only the packages listed in the **Private-Package** element in the bundle. No packages are exported.

The default behavior can result in very large packages and in exporting packages that should be kept private. To change the list of exported packages you can add an **Export-Package** child to the plug-in's **instructions** element.

The **Export-Package** element specifies a list of packages that are to be included in the bundle and that are to be exported. The package names can be specified using the `*` wildcard symbol. For example, the entry `com.fuse.demo.*` includes all packages on the project's classpath that start with `com.fuse.demo`.

You can specify packages to be excluded by prefixing the entry with `!`. For example, the entry `!com.fuse.demo.private` excludes the package `com.fuse.demo.private`.

When excluding packages, the order of entries in the list is important. The list is processed in order from the beginning and any subsequent contradicting entries are ignored.

For example, to include all packages starting with `com.fuse.demo` except the package `com.fuse.demo.private`, list the packages using:

```
!com.fuse.demo.private,com.fuse.demo.*
```

However, if you list the packages using `com.fuse.demo.*,!com.fuse.demo.private`, then `com.fuse.demo.private` is included in the bundle because it matches the first pattern.

Specifying private packages

If you want to specify a list of packages to include in a bundle **without** exporting them, you can add a **Private-Package** instruction to the bundle plug-in configuration. By default, if you do not specify a **Private-Package** instruction, all packages in your local Java source are included in the bundle.



IMPORTANT

If a package matches an entry in both the **Private-Package** element and the **Export-Package** element, the **Export-Package** element takes precedence. The package is added to the bundle and exported.

The **Private-Package** element works similarly to the **Export-Package** element in that you specify a list of packages to be included in the bundle. The bundle plug-in uses the list to find all classes on the project's classpath that are to be included in the bundle. These packages are packaged in the bundle, but not exported (unless they are also selected by the **Export-Package** instruction).

[Example A.5, "Including a private package in a bundle"](#) shows the configuration for including a private package in a bundle

Example A.5. Including a private package in a bundle

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
    <instructions>
      <Private-Package>org.apache.cxf.wsdlFirst.impl</Private-Package>
      ...
    </instructions>
  </configuration>
</plugin>
```

Specifying imported packages

By default, the bundle plug-in populates the OSGi manifest's **Import-Package** property with a list of all the packages referred to by the contents of the bundle.

While the default behavior is typically sufficient for most projects, you might find instances where you want to import packages that are not automatically added to the list. The default behavior can also result in unwanted packages being imported.

To specify a list of packages to be imported by the bundle, add an **Import-Package** child to the plug-in's **instructions** element. The syntax for the package list is the same as for the **Export-Package** element and the **Private-Package** element.



IMPORTANT

When you use the **Import-Package** element, the plug-in does not automatically scan the bundle's contents to determine if there are any required imports. To ensure that the contents of the bundle are scanned, you must place an * as the last entry in the package list.

[Example A.6, "Specifying the packages imported by a bundle"](#) shows the configuration for specifying the packages imported by a bundle

Example A.6. Specifying the packages imported by a bundle

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
    <instructions>
      <Import-Package>javax.jws, javax.wsdl, org.apache.cxf.bus, org.apache.cxf.bus.spring,
      org.apache.cxf.bus.resource, org.apache.cxf.configuration.spring, org.apache.cxf.resource,
      org.springframework.beans.factory.config, * </Import-Package>
      ...
    </instructions>
  </configuration>
</plugin>
```

More information

For more information on configuring a bundle plug-in, see:

- [olink:OsgiDependencies/OsgiDependencies](#)
- [Apache Felix documentation](#)
- [Peter Kriens' aQute Software Consultancy web site](#)

PART V. DEVELOPING APPLICATIONS USING JAX-WS

This guide describes how to develop Web services using the standard JAX-WS APIs.

CHAPTER 24. BOTTOM-UP SERVICE DEVELOPMENT

Abstract

There are many instances where you have Java code that already implements a set of functionality that you want to expose as part of a service oriented application. You may also simply want to avoid using WSDL to define your interface. Using JAX-WS annotations, you can add the information required to service enable a Java class. You can also create a *Service Endpoint Interface* (SEI) that can be used in place of a WSDL contract. If you want a WSDL contract, Apache CXF provides tools to generate a contract from annotated Java code.

24.1. INTRODUCTION TO JAX-WS SERVICE DEVELOPMENT

To create a service starting from Java you must do the following:

1. [Section 24.2, "Creating the SEI"](#) a Service Endpoint Interface (SEI) that defines the methods you want to expose as a service.



NOTE

You can work directly from a Java class, but working from an interface is the recommended approach. Interfaces are better suited for sharing with the developers who are responsible for developing the applications consuming your service. The interface is smaller and does not provide any of the service's implementation details.

2. [Section 24.3, "Annotating the Code"](#) the required annotations to your code.
3. [Section 24.4, "Generating WSDL"](#) the WSDL contract for your service.



NOTE

If you intend to use the SEI as the service's contract, it is not necessary to generate a WSDL contract.

4. [Chapter 31, Publishing a Service](#) the service as a service provider.

24.2. CREATING THE SEI

Overview

The *service endpoint interface* (SEI) is the piece of Java code that is shared between a service implementation and the consumers that make requests on that service. The SEI defines the methods implemented by the service and provides details about how the service will be exposed as an endpoint. When starting with a WSDL contract, the SEI is generated by the code generators. However, when starting from Java, it is the developer's responsibility to create the SEI. There are two basic patterns for creating an SEI:

- Green field development – In this pattern, you are developing a new service without any existing Java code or WSDL. It is best to start by creating the SEI. You can then distribute the SEI to any developers that are responsible for implementing the service providers and consumers that use the SEI.

**NOTE**

The recommended way to do green field service development is to start by creating a WSDL contract that defines the service and its interfaces. See [Chapter 26, A Starting Point WSDL Contract](#).

- Service enablement – In this pattern, you typically have an existing set of functionality that is implemented as a Java class, and you want to service enable it. This means that you must do two things:
 - a. Create an SEI that contains **only** the operations that are going to be exposed as part of the service.
 - b. Modify the existing Java class so that it implements the SEI.

**NOTE**

Although you can add the JAX-WS annotations to a Java class, it is not recommended.

Writing the interface

The SEI is a standard Java interface. It defines a set of methods that a class implements. It can also define a number of member fields and constants to which the implementing class has access.

In the case of an SEI the methods defined are intended to be mapped to operations exposed by a service. The SEI corresponds to a **wsdl:portType** element. The methods defined by the SEI correspond to **wsdl:operation** elements in the **wsdl:portType** element.

**NOTE**

JAX-WS defines an annotation that allows you to specify methods that are not exposed as part of a service. However, the best practice is to leave those methods out of the SEI.

[Example 24.1, “Simple SEI”](#) shows a simple SEI for a stock updating service.

Example 24.1. Simple SEI

```
package com.fusesource.demo;

public interface quoteReporter
{
    public Quote getQuote(String ticker);
}
```

Implementing the interface

Because the SEI is a standard Java interface, the class that implements it is a standard Java class. If you start with a Java class you must modify it to implement the interface. If you start with the SEI, the implementation class implements the SEI.

[Example 24.2, "Simple Implementation Class"](#) shows a class for implementing the interface in [Example 24.1, "Simple SEI"](#).

Example 24.2. Simple Implementation Class

```
package com.fusesource.demo;

import java.util.*;

public class stockQuoteReporter implements quoteReporter
{
    ...
    public Quote getQuote(String ticker)
    {
        Quote retVal = new Quote();
        retVal.setID(ticker);
        retVal.setVal(Board.check(ticker));[1]
        Date retDate = new Date();
        retVal.setTime(retDate.toString());
        return(retVal);
    }
}
```

24.3. ANNOTATING THE CODE

24.3.1. Overview of JAX-WS Annotations

The JAX-WS annotations specify the metadata used to map the SEI to a fully specified service definition. Among the information provided in the annotations are the following:

- The target namespace for the service.
- The name of the class used to hold the request message
- The name of the class used to hold the response message
- If an operation is a one way operation
- The binding style the service uses
- The name of the class used for any custom exceptions
- The namespaces under which the types used by the service are defined



NOTE

Most of the annotations have sensible defaults and it is not necessary to provide values for them. However, the more information you provide in the annotations, the better your service definition is specified. A well-specified service definition increases the likelihood that all parts of a distributed application will work together.

24.3.2. Required Annotations

Overview

In order to create a service from Java code you are only required to add one annotation to your code. You must add the **@WebService** annotation on both the SEI and the implementation class.

The @WebService annotation

The **@WebService** annotation is defined by the javax.jws.WebService interface and it is placed on an interface or a class that is intended to be used as a service. **@WebService** has the properties described in Table 24.1, “[@WebService Properties](#)”

Table 24.1. @WebService Properties

Property	Description
name	Specifies the name of the service interface. This property is mapped to the name attribute of the wsdl:portType element that defines the service’s interface in a WSDL contract. The default is to append PortType to the name of the implementation class. [a]
targetNamespace	Specifies the target namespace where the service is defined. If this property is not specified, the target namespace is derived from the package name.
serviceName	Specifies the name of the published service. This property is mapped to the name attribute of the wsdl:service element that defines the published service. The default is to use the name of the service’s implementation class.
wsdlLocation	Specifies the URL where the service’s WSDL contract is stored. This must be specified using a relative URL. The default is the URL where the service is deployed.
endpointInterface	Specifies the full name of the SEI that the implementation class implements. This property is only specified when the attribute is used on a service implementation class.
portName	Specifies the name of the endpoint at which the service is published. This property is mapped to the name attribute of the wsdl:port element that specifies the endpoint details for a published service. The default is the append Port to the name of the service’s implementation class.

[a] When you generate WSDL from an SEI the interface’s name is used in place of the implementation class’ name.



NOTE

It is not necessary to provide values for any of the **@WebService** annotation's properties. However, it is recommended that you provide as much information as you can.

Annotating the SEI

The SEI requires that you add the **@WebService** annotation. Because the SEI is the contract that defines the service, you should specify as much detail as possible about the service in the **@WebService** annotation's properties.

[Example 24.3, "Interface with the @WebService Annotation"](#) shows the interface defined in [Example 24.1, "Simple SEI"](#) with the **@WebService** annotation.

Example 24.3. Interface with the @WebService Annotation

```
package com.fusesource.demo;

import javax.jws.*;

@WebService(name="quoteUpdater",
            targetNamespace="http:\\demos.redhat.com",
            serviceName="updateQuoteService",
            wsdlLocation="http:\\demos.redhat.com\\quoteExampleService?wsdl",
            portName="updateQuotePort")
public interface quoteReporter
{
    public Quote getQuote(String ticker);
}
```

The **@WebService** annotation in [Example 24.3, "Interface with the @WebService Annotation"](#) does the following:

Specifies that the value of the **name** attribute of the **wsdl:portType** element defining the service interface is **quoteUpdater**.

Specifies that the target namespace of the service is <http://demos.redhat.com>.

Specifies that the value of the **name** of the **wsdl:service** element defining the published service is **updateQuoteService**.

Specifies that the service will publish its WSDL contract at <http://demos.redhat.com/quoteExampleService?wsdl>.

Specifies that the value of the **name** attribute of the **wsdl:port** element defining the endpoint exposing the service is **updateQuotePort**.

Annotating the service implementation

In addition to annotating the SEI with the **@WebService** annotation, you also must annotate the service implementation class with the **@WebService** annotation. When adding the annotation to the service implementation class you only need to specify the **endpointInterface** property. As shown in [Example 24.4, "Annotated Service Implementation Class"](#) the property must be set to the full name of the SEI.

Example 24.4. Annotated Service Implementation Class

```
package org.eric.demo;

import javax.jws.*;

@WebService(endpointInterface="com.fusesource.demo.quoteReporter")
public class stockQuoteReporter implements quoteReporter
{
    public Quote getQuote(String ticker)
    {
        ...
    }
}
```

24.3.3. Optional Annotations

Abstract

While the **@WebService** annotation is sufficient for service enabling a Java interface or a Java class, it does not fully describe how the service will be exposed as a service provider. The JAX-WS programming model uses a number of optional annotations for adding details about your service, such as the binding it uses, to the Java code. You add these annotations to the service's SEI.

The more details you provide in the SEI the easier it is for developers to implement applications that can use the functionality it defines. It also makes the WSDL documents generated by the tools more specific.

Overview

Defining the Binding Properties with Annotations

If you are using a SOAP binding for your service, you can use JAX-WS annotations to specify a number of the bindings properties. These properties correspond directly to the properties you can specify in a service's WSDL contract. Some of the settings, such as the parameter style, can restrict how you implement a method. These settings can also effect which annotations can be used when annotating method parameters.

The **@SOAPBinding** annotation

The **@SOAPBinding** annotation is defined by the javax.jws.soap.SOAPBinding interface. It provides details about the SOAP binding used by the service when it is deployed. If the **@SOAPBinding** annotation is not specified, a service is published using a wrapped doc/literal SOAP binding.

You can put the **@SOAPBinding** annotation on the SEI and any of the SEI's methods. When it is used on a method, setting of the method's **@SOAPBinding** annotation take precedence.

[Table 24.2, “@SOAPBinding Properties”](#) shows the properties for the **@SOAPBinding** annotation.

Table 24.2. @SOAPBinding Properties

Property	Values	Description
style	Style.DOCUMENT (default) Style.RPC	Specifies the style of the SOAP message. If RPC style is specified, each message part within the SOAP body is a parameter or return value and appears inside a wrapper element within the soap:body element. The message parts within the wrapper element correspond to operation parameters and must appear in the same order as the parameters in the operation. If DOCUMENT style is specified, the contents of the SOAP body must be a valid XML document, but its form is not as tightly constrained.
use	Use.LITERAL (default) Use.ENCODED ^[a]	Specifies how the data of the SOAP message is streamed.
parameterStyle ^[b]	ParameterStyle.BARE ParameterStyle.WRAPPED (default)	Specifies how the method parameters, which correspond to message parts in a WSDL contract, are placed into the SOAP message body. If BARE is specified, each parameter is placed into the message body as a child element of the message root. If WRAPPED is specified, all of the input parameters are wrapped into a single element on a request message and all of the output parameters are wrapped into a single element in the response message.

[a] Use.ENCODED is not currently supported.

[b] If you set the **style** to RPC you must use the WRAPPED parameter style.

Document bare style parameters

Document bare style is the most direct mapping between Java code and the resulting XML representation of the service. When using this style, the schema types are generated directly from the input and output parameters defined in the operation's parameter list.

You specify you want to use bare document\literal style by using the **@SOAPBinding** annotation with its **style** property set to Style.DOCUMENT, and its **parameterStyle** property set to ParameterStyle.BARE.

To ensure that an operation does not violate the restrictions of using document style when using bare parameters, your operations must adhere to the following conditions:

- The operation must have no more than one input or input/output parameter.
- If the operation has a return type other than **void**, it must not have any output or input/output parameters.
- If the operation has a return type of **void**, it must have no more than one output or input/output parameter.



NOTE

Any parameters that are placed in the SOAP header using the **@WebParam** annotation or the **@WebResult** annotation are not counted against the number of allowed parameters.

Document wrapped parameters

Document wrapped style allows a more RPC like mapping between the Java code and the resulting XML representation of the service. When using this style, the parameters in the method's parameter list are wrapped into a single element by the binding. The disadvantage of this is that it introduces an extra-layer of indirection between the Java implementation and how the messages are placed on the wire.

To specify that you want to use wrapped document\literal style use the **@SOAPBinding** annotation with its **style** property set to `Style.DOCUMENT`, and its **parameterStyle** property set to `ParameterStyle.WRAPPED`.

You have some control over how the wrappers are generated by using the [the section called “The @RequestWrapper annotation”](#) annotation and the [the section called “The @ResponseWrapper annotation”](#) annotation.

Example

[Example 24.5, “Specifying a Document Bare SOAP Binding with the SOAP Binding Annotation”](#) shows an SEI that uses document bare SOAP messages.

Example 24.5. Specifying a Document Bare SOAP Binding with the SOAP Binding Annotation

```
package org.eric.demo;

import javax.jws.*;
import javax.jws.soap.*;
import javax.jws.soap.SOAPBinding.*;

@WebService(name="quoteReporter")
@SOAPBinding(parameterStyle=ParameterStyle.BARE)
public interface quoteReporter
{
    ...
}
```

Overview

Defining Operation Properties with Annotations

When the runtime maps your Java method definitions into XML operation definitions it provides details such as:

- What the exchanged messages look like in XML
- If the message can be optimized as a one way message
- The namespaces where the messages are defined

The @WebMethod annotation

The **@WebMethod** annotation is defined by the javax.jws.WebMethod interface. It is placed on the methods in the SEI. The **@WebMethod** annotation provides the information that is normally represented in the **wsdl:operation** element describing the operation to which the method is associated.

[Table 24.3, “@WebMethod Properties”](#) describes the properties of the **@WebMethod** annotation.

Table 24.3. @WebMethod Properties

Property	Description
operationName	Specifies the value of the associated wsdl:operation element’s name . The default value is the name of the method.
action	Specifies the value of the soapAction attribute of the soap:operation element generated for the method. The default value is an empty string.
exclude	Specifies if the method should be excluded from the service interface. The default is false.

The @RequestWrapper annotation

The **@RequestWrapper** annotation is defined by the javax.xml.ws.RequestWrapper interface. It is placed on the methods in the SEI. The **@RequestWrapper** annotation specifies the Java class implementing the wrapper bean for the method parameters of the request message starting a message exchange. It also specifies the element names, and namespaces, used by the runtime when marshalling and unmarshalling the request messages.

[Table 24.4, “@RequestWrapper Properties”](#) describes the properties of the **@RequestWrapper** annotation.

Table 24.4. @RequestWrapper Properties

Property	Description
----------	-------------

Property	Description
localName	Specifies the local name of the wrapper element in the XML representation of the request message. The default value is either the name of the method, or the value of the the section called “The @WebMethod annotation” annotation’s operationName property.
targetNamespace	Specifies the namespace under which the XML wrapper element is defined. The default value is the target namespace of the SEI.
className	Specifies the full name of the Java class that implements the wrapper element.

**NOTE**

Only the **className** property is required.

**IMPORTANT**

If the method is also annotated with the **@SOAPBinding** annotation, and its **parameterStyle** property is set to **ParameterStyle.BARE**, this annotation is ignored.

The @ResponseWrapper annotation

The **@ResponseWrapper** annotation is defined by the javax.xml.ws.ResponseWrapper interface. It is placed on the methods in the SEI. The **@ResponseWrapper** specifies the Java class implementing the wrapper bean for the method parameters in the response message in the message exchange. It also specifies the element names, and namespaces, used by the runtime when marshaling and unmarshalling the response messages.

[Table 24.5, “@ResponseWrapper Properties”](#) describes the properties of the **@ResponseWrapper** annotation.

Table 24.5. @ResponseWrapper Properties

Property	Description
localName	Specifies the local name of the wrapper element in the XML representation of the response message. The default value is either the name of the method with Response appended, or the value of the the section called “The @WebMethod annotation” annotation’s operationName property with Response appended.
targetNamespace	Specifies the namespace where the XML wrapper element is defined. The default value is the target namespace of the SEI.

Property	Description
className	Specifies the full name of the Java class that implements the wrapper element.

**NOTE**

Only the **className** property is required.

**IMPORTANT**

If the method is also annotated with the **@SOAPBinding** annotation and its **parameterStyle** property is set to **ParameterStyle.BARE**, this annotation is ignored.

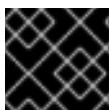
The @WebFault annotation

The **@WebFault** annotation is defined by the javax.xml.ws.WebFault interface. It is placed on exceptions that are thrown by your SEI. The **@WebFault** annotation is used to map the Java exception to a **wsdl:fault** element. This information is used to marshall the exceptions into a representation that can be processed by both the service and its consumers.

[Table 24.6, “@WebFault Properties”](#) describes the properties of the **@WebFault** annotation.

Table 24.6. @WebFault Properties

Property	Description
name	Specifies the local name of the fault element.
targetNamespace	Specifies the namespace under which the fault element is defined. The default value is the target namespace of the SEI.
faultName	Specifies the full name of the Java class that implements the exception.

**IMPORTANT**

The **name** property is required.

The @Oneway annotation

The **@Oneway** annotation is defined by the javax.jws.Oneway interface. It is placed on the methods in the SEI that will not require a response from the service. The **@Oneway** annotation tells the run time that it can optimize the execution of the method by not waiting for a response and by not reserving any resources to process a response.

This annotation can only be used on methods that meet the following criteria:

- They return **void**

- They have no parameters that implement the Holder interface
- They do not throw any exceptions that can be passed back to a consumer

Example

[Example 24.6, "SEI with Annotated Methods"](#) shows an SEI with its methods annotated.

Example 24.6. SEI with Annotated Methods

```
package com.fusesource.demo;

import javax.jws.*;
import javax.xml.ws.*;

@WebService(name="quoteReporter")
public interface quoteReporter
{
    @WebMethod(operationName="getStockQuote")
    @RequestWrapper(targetNamespace="http://demo.redhat.com/types",
                    className="java.lang.String")
    @ResponseWrapper(targetNamespace="http://demo.redhat.com/types",
                     className="org.eric.demo.Quote")
    public Quote getQuote(String ticker);
}
```

Overview

Defining Parameter Properties with Annotations

The method parameters in the SEI correspond to the **wsdl:message** elements and their **wsdl:part** elements. JAX-WS provides annotations that allow you to describe the **wsdl:part** elements that are generated for the method parameters.

The **@WebParam** annotation

The **@WebParam** annotation is defined by the javax.jws.WebParam interface. It is placed on the parameters of the methods defined in the SEI. The **@WebParam** annotation allows you to specify the direction of the parameter, if the parameter will be placed in the SOAP header, and other properties of the generated **wsdl:part**.

[Table 24.7, "@WebParam Properties"](#) describes the properties of the **@WebParam** annotation.

Table 24.7. @WebParam Properties

Property	Values	Description
----------	--------	-------------

Property	Values	Description
name		Specifies the name of the parameter as it appears in the generated WSDL document. For RPC bindings, this is the name of the wsdl:part representing the parameter. For document bindings, this is the local name of the XML element representing the parameter. Per the JAX-WS specification, the default is argN , where <i>N</i> is replaced with the zero-based argument index (i.e., arg0, arg1, etc.).
targetNamespace		Specifies the namespace for the parameter. It is only used with document bindings where the parameter maps to an XML element. The default is to use the service's namespace.
mode	Mode.IN (default) ^[a] Mode.OUT Mode.INOUT	Specifies the direction of the parameter.
header	false (default) true	Specifies if the parameter is passed as part of the SOAP header.
partName		Specifies the value of the name attribute of the wsdl:part element for the parameter. This property is used for document style SOAP bindings.

[a] Any parameter that implements the Holder interface is mapped to Mode.INOUT by default.

The @WebResult annotation

The **@WebResult** annotation is defined by the javax.jws.WebResult interface. It is placed on the methods defined in the SEI. The **@WebResult** annotation allows you to specify the properties of the **wsdl:part** that is generated for the method's return value.

Table 24.8, “[@WebResult Properties](#)” describes the properties of the **@WebResult** annotation.

Table 24.8. **@WebResult Properties**

Property	Description
name	Specifies the name of the return value as it appears in the generated WSDL document. For RPC bindings, this is the name of the wsdl:part representing the return value. For document bindings, this is the local name of the XML element representing the return value. The default value is <code>return</code> .
targetNamespace	Specifies the namespace for the return value. It is only used with document bindings where the return value maps to an XML element. The default is to use the service's namespace.
header	Specifies if the return value is passed as part of the SOAP header.
partName	Specifies the value of the name attribute of the wsdl:part element for the return value. This property is used for document style SOAP bindings.

Example

[Example 24.7, "Fully Annotated SEI"](#) shows an SEI that is fully annotated.

Example 24.7. Fully Annotated SEI

```
package com.fusesource.demo;

import javax.jws.*;
import javax.xml.ws.*;
import javax.jws.soap.*;
import javax.jws.soap.SOAPBinding.*;
import javax.jws.WebParam.*;

@WebService(targetNamespace="http://demo.redhat.com",
           name="quoteReporter")
@SOAPBinding(style=Style.RPC, use=Use.LITERAL)
public interface quoteReporter
{
    @WebMethod(operationName="getStockQuote")
    @RequestWrapper(targetNamespace="http://demo.redhat.com/types",
                    className="java.lang.String")
    @ResponseWrapper(targetNamespace="http://demo.redhat.com/types",
                     className="org.eric.demo.Quote")
    @WebResult(targetNamespace="http://demo.redhat.com/types",
               name="updatedQuote")
    public Quote getQuote(
        @WebParam(targetNamespace="http://demo.redhat.com/types",
                  name="stockTicker",
                  mode=Mode.IN)
```

```

    String ticker
);
}

```

24.3.4. Apache CXF Annotations

24.3.4.1. WSDL Documentation

@WSDLDocumentation annotation

The **@WSDLDocumentation** annotation is defined by the `org.apache.cxf.annotations.WSDLDocumentation` interface. It can be placed on the SEI or the SEI methods.

This annotation enables you to add documentation, which will then appear within **wsdl:documentation** elements after the SEI is converted to WSDL. By default, the documentation elements appear inside the port type, but you can specify the placement property to make the documentation appear at other locations in the WSDL file. [Section 24.3.4.2, “@WSDLDocumentation properties”](#) shows the properties supported by the **@WSDLDocumentation** annotation.

24.3.4.2. @WSDLDocumentation properties

Property	Description
value	(Required) A string containing the documentation text.
placement	(Optional) Specifies where in the WSDL file this documentation is to appear. For the list of possible placement values, see the section called “Placement in the WSDL contract” .
faultClass	(Optional) If the placement is set to be FAULT_MESSAGE , PORT_TYPE_OPERATION_FAULT , or BINDING_OPERATION_FAULT , you must also set this property to the Java class that represents the fault.

@WSDLDocumentationCollection annotation

The **@WSDLDocumentationCollection** annotation is defined by the `org.apache.cxf.annotations.WSDLDocumentationCollection` interface. It can be placed on the SEI or the SEI methods.

This annotation is used to insert multiple documentation elements at a single placement location or at various placement locations.

Placement in the WSDL contract

To specify where the documentation should appear in the WSDL contract, you can specify the **placement** property, which is of type **WSDLDocumentation.Placement**. The placement can have one of the following values:

- **WSDLDocumentation.Placement.BINDING**
- **WSDLDocumentation.Placement.BINDING_OPERATION**
- **WSDLDocumentation.Placement.BINDING_OPERATION_FAULT**
- **WSDLDocumentation.Placement.BINDING_OPERATION_INPUT**
- **WSDLDocumentation.Placement.BINDING_OPERATION_OUTPUT**
- **WSDLDocumentation.Placement.DEFAULT**
- **WSDLDocumentation.PlacementFAULT_MESSAGE**
- **WSDLDocumentation.Placement.INPUT_MESSAGE**
- **WSDLDocumentation.Placement.OUTPUT_MESSAGE**
- **WSDLDocumentation.Placement.PORT_TYPE**
- **WSDLDocumentation.Placement.PORT_TYPE_OPERATION**
- **WSDLDocumentation.Placement.PORT_TYPE_OPERATION_FAULT**
- **WSDLDocumentation.Placement.PORT_TYPE_OPERATION_INPUT**
- **WSDLDocumentation.Placement.PORT_TYPE_OPERATION_OUTPUT**
- **WSDLDocumentation.Placement.SERVICE**
- **WSDLDocumentation.Placement.SERVICE_PORT**
- **WSDLDocumentation.Placement.TOP**

Example of @WSDLDocumentation

[Section 24.3.4.3, “Using @WSDLDocumentation”](#) shows how to add a **@WSDLDocumentation** annotation to the SEI and to one of its methods.

24.3.4.3. Using @WSDLDocumentation

```

@WebService
@WSDLDocumentation("A very simple example of an SEI")
public interface HelloWorld {
    @WSDLDocumentation("A traditional form of greeting")
    String sayHi(@WebParam(name = "text") String text);
}

```

When WSDL, shown in [Section 24.3.4.4, “WSDL generated with documentation”](#), is generated from the SEI in [Section 24.3.4.3, “Using @WSDLDocumentation”](#), the default placements of the **documentation** elements are, respectively, **PORT_TYPE** and **PORT_TYPE_OPERATION**.

24.3.4.4. WSDL generated with documentation

```
<wsdl:definitions ... >
...
<wsdl:portType name="HelloWorld">
  <wsdl:documentation>A very simple example of an SEI</wsdl:documentation>
  <wsdl:operation name="sayHi">
    <wsdl:documentation>A traditional form of greeting</wsdl:documentation>
    <wsdl:input name="sayHi" message="tns:sayHi">
      </wsdl:input>
    <wsdl:output name="sayHiResponse" message="tns:sayHiResponse">
      </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
...
</wsdl:definitions>
```

Example of @WSDLDocumentationCollection

[Section 24.3.4.5, “Using @WSDLDocumentationCollection”](#) shows how to add a **@WSDLDocumentationCollection** annotation to an SEI.

24.3.4.5. Using @WSDLDocumentationCollection

```
@WebService
@WSDLDocumentationCollection(
{
  @WSDLDocumentation("A very simple example of an SEI"),
  @WSDLDocumentation(value = "My top level documentation",
                     placement = WSDLDocumentation.Placement.TOP),
  @WSDLDocumentation(value = "Binding documentation",
                     placement = WSDLDocumentation.Placement.BINDING)
}
)
public interface HelloWorld {
  @WSDLDocumentation("A traditional form of Geeky greeting")
  String sayHi(@WebParam(name = "text") String text);
}
```

24.3.4.6. Schema Validation of Messages

@SchemaValidation annotation

The **@SchemaValidation** annotation is defined by the org.apache.cxf.annotations.SchemaValidation interface. It can be placed on the SEI and on individual SEI methods.

This annotation turns on schema validation of the XML messages sent to this endpoint. This can be useful for testing purposes, when you suspect there is a problem with the format of incoming XML messages. By default, validation is disabled, because it has a significant impact on performance.

Schema validation type

The schema validation behaviour is controlled by the **type** parameter, whose value is an enumeration of **org.apache.cxf.annotations.SchemaValidation.SchemaValidationType** type. [Section 24.3.4.7, "Schema Validation Type Values"](#) shows the list of available validation types.

24.3.4.7. Schema Validation Type Values

Type	Description
IN	Apply schema validation to incoming messages on client and server.
OUT	Apply schema validation to outgoing messages on client and server.
BOTH	Apply schema validation to both incoming and outgoing messages on client and server.
NONE	All schema validation is disabled.
REQUEST	Apply schema validation to Request messages—that is, causing validation to be applied to outgoing client messages and to incoming server messages.
RESPONSE	Apply schema validation to Response messages—that is, causing validation to be applied to incoming client messages, and outgoing server messages.

Example

The following example shows how to enable schema validation of messages for endpoints based on the MyService SEI. Note how the annotation can be applied to the SEI as a whole, as well as to individual methods in the SEI.

```

@WebService
@SchemaValidation(type = SchemaValidationType.BOTH)
public interface MyService {
    Foo validateBoth(Bar data);

    @SchemaValidation(type = SchemaValidationType.NONE)
    Foo validateNone(Bar data);

    @SchemaValidation(type = SchemaValidationType.IN)
    Foo validateIn(Bar data);

    @SchemaValidation(type = SchemaValidationType.OUT)
    Foo validateOut(Bar data);

    @SchemaValidation(type = SchemaValidationType.REQUEST)
    Foo validateRequest(Bar data);

```

```

    @SchemaValidation(type = SchemaValidationType.RESPONSE)
    Foo validateResponse(Bar data);
}

```

24.3.4.8. Specifying the Data Binding

@DataBinding annotation

The **@DataBinding** annotation is defined by the org.apache.cxf.annotations.DataBinding interface. It is placed on the SEI.

This annotation is used to associate a data binding with the SEI, replacing the default JAXB data binding. The value of the **@DataBinding** annotation must be the class that provides the data binding, **ClassName.class**.

Supported data bindings

The following data bindings are currently supported by Apache CXF:

- **org.apache.cxf.jaxb.JAXBDataBinding**
 (Default) The standard [JAXB](#) data binding.
- **org.apache.cxf.sdo.SDODataBinding**
 The Service Data Objects (SDO) data binding is based on the [Apache Tuscany SDO](#) implementation. If you want to use this data binding in the context of a Maven build, you need to add a dependency on the **cxf-rt-databinding-sdo** artifact.
- **org.apache.cxf.aegis.databinding.AegisDatabinding**
 If you want to use this data binding in the context of a Maven build, you need to add a dependency on the **cxf-rt-databinding-aegis** artifact.
- **org.apache.cxf.xmlbeans.XmlBeansDataBinding**
 If you want to use this data binding in the context of a Maven build, you need to add a dependency on the **cxf-rt-databinding-xmlbeans** artifact.
- **org.apache.cxf.databinding.source.SourceDataBinding**
 This data binding belongs to the Apache CXF core.
- **org.apache.cxf.databinding.stax.StaxDataBinding**
 This data binding belongs to the Apache CXF core.

Example

[Section 24.3.4.9, "Setting the data binding"](#) shows how to associate the SDO binding with the **HelloWorld** SEI

24.3.4.9. Setting the data binding

```

@WebService
@DataBinding(org.apache.cxf.sdo.SDODataBinding.class)
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}

```

24.3.4.10. Compressing Messages

@GZIP annotation

The **@GZIP** annotation is defined by the org.apache.cxf.annotations.GZIP interface. It is placed on the SEI.

Enables GZIP compression of messages. GZIP is a negotiated enhancement. That is, an initial request from a client will not be gzipped, but an **Accept** header will be added and, if the server supports GZIP compression, the response will be gzipped and any subsequent requests will be also.

[Section 24.3.4.11, “@GZIP Properties”](#) shows the optional properties supported by the **@GZIP** annotation.

24.3.4.11. @GZIP Properties

Property	Description
threshold	Messages smaller than the size specified by this property are not gzipped. Default is -1 (no limit).

@FastInfoSet

The **@FastInfoSet** annotation is defined by the org.apache.cxf.annotations.FastInfoSet interface. It is placed on the SEI.

Enables the use of FastInfoSet format for messages. FastInfoSet is a binary encoding format for XML, which aims to optimize both the message size and the processing performance of XML messages. For more details, see the following Sun article on [Fast InfoSet](#).

FastInfoSet is a negotiated enhancement. That is, an initial request from a client will not be in FastInfoSet format, but an **Accept** header will be added and, if the server supports FastInfoSet, the response will be in FastInfoSet and any subsequent requests will be also.

[Section 24.3.4.12, “@FastInfoSet Properties”](#) shows the optional properties supported by the **@FastInfoSet** annotation.

24.3.4.12. @FastInfoSet Properties

Property	Description
force	A boolean property that forces the use of FastInfoSet format, instead of negotiating. When true , force the use of FastInfoSet format; otherwise, negotiate. Default is false .

Example of @GZIP

[Section 24.3.4.13, “Enabling GZIP”](#) shows how to enable GZIP compression for the **HelloWorld** SEI.

24.3.4.13. Enabling GZIP

```

@WebService
@GZIP
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}

```

Example of @FastInfoSet

[Section 24.3.4.14, “Enabling FastInfoSet”](#) shows how to enable the FastInfoSet format for the **HelloWorld** SEI.

24.3.4.14. Enabling FastInfoSet

```

@WebService
@FastInfoSet
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}

```

24.3.4.15. Enable Logging on an Endpoint

@Logging annotation

The **@Logging** annotation is defined by the org.apache.cxf.annotations.Logging interface. It is placed on the SEI.

This annotation enables logging for all endpoints associated with the SEI. [Section 24.3.4.16, “@Logging Properties”](#) shows the optional properties you can set in this annotation.

24.3.4.16. @Logging Properties

Property	Description
limit	Specifies the size limit, beyond which the message is truncated in the logs. Default is 64K.
inLocation	Specifies the location to log incoming messages. Can be either <stderr> , <stdout> , <logger> , or a filename. Default is <logger> .
outLocation	Specifies the location to log outgoing messages. Can be either <stderr> , <stdout> , <logger> , or a filename. Default is <logger> .

Example

[Section 24.3.4.17, “Logging configuration using annotations”](#) shows how to enable logging for the **HelloWorld** SEI, where incoming messages are sent to **<stdout>** and outgoing messages are sent to **<logger>**.

24.3.4.17. Logging configuration using annotations

```
@WebService
@Logging(limit=16000, inLocation="<stdout>")
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

24.3.4.18. Adding Properties and Policies to an Endpoint

Abstract

Both properties and policies can be used to associate configuration data with an endpoint. The essential difference between them is that **properties** are a Apache CXF specific configuration mechanism whereas **policies** are a standard WSDL configuration mechanism. Policies typically originate from WS specifications and standards and they are normally set by defining **wsdl:policy** elements that appear in the WSDL contract. By contrast, properties are Apache CXF-specific and they are normally set by defining **jaxws:properties** elements in the Apache CXF Spring configuration file.

It is also possible, however, to define property settings and WSDL policy settings in Java using annotations, as described here.

24.3.4.19. Adding properties

@EndpointProperty annotation

The **@EndpointProperty** annotation is defined by the **org.apache.cxf.annotations.EndpointProperty** interface. It is placed on the SEI.

This annotation adds Apache CXF-specific configuration settings to an endpoint. Endpoint properties can also be specified in a Spring configuration file. For example, to configure WS-Security on an endpoint, you could add endpoint properties using the **jaxws:properties** element in a Spring configuration file as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       ... >

    <jaxws:endpoint
        id="MyService"
        address="https://localhost:9001/MyService"
        serviceName="interop:MyService"
        endpointName="interop:MyServiceEndpoint"
        implementor="com.foo.MyService">

        <jaxws:properties>
            <entry key="ws-security.callback-handler" value="interop.client.UTPasswordCallback"/>
            <entry key="ws-security.signature.properties" value="etc/keystore.properties"/>
            <entry key="ws-security.encryption.properties" value="etc/truststore.properties"/>
            <entry key="ws-security.encryption.username" value="useReqSigCert"/>
        </jaxws:properties>
    
```

```
</jaxws:endpoint>
</beans>
```

Alternatively, you could specify the preceding configuration settings in Java by adding **@EndpointProperty** annotations to the SEI, as shown in [Section 24.3.4.20, "Configuring WS-Security Using @EndpointProperty Annotations"](#).

24.3.4.20. Configuring WS-Security Using @EndpointProperty Annotations

```
@WebService
@EndpointProperty(name="ws-security.callback-handler"
value="interop.client.UTPasswordCallback")
@EndpointProperty(name="ws-security.signature.properties" value="etc/keystore.properties")
@EndpointProperty(name="ws-security.encryption.properties" value="etc/truststore.properties")
@EndpointProperty(name="ws-security.encryption.username" value="useReqSigCert")
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

@EndpointProperties annotation

The **@EndpointProperties** annotation is defined by the **org.apache.cxf.annotations.EndpointProperties** interface. It is placed on the SEI.

This annotation provides a way of grouping multiple **@EndpointProperty** annotations into a list. Using **@EndpointProperties**, it is possible to re-write [Section 24.3.4.20, "Configuring WS-Security Using @EndpointProperty Annotations"](#) as shown in [Section 24.3.4.21, "Configuring WS-Security Using an @EndpointProperties Annotation"](#).

24.3.4.21. Configuring WS-Security Using an @EndpointProperties Annotation

```
@WebService
@EndpointProperties(
{
    @EndpointProperty(name="ws-security.callback-handler"
value="interop.client.UTPasswordCallback"),
    @EndpointProperty(name="ws-security.signature.properties" value="etc/keystore.properties"),
    @EndpointProperty(name="ws-security.encryption.properties" value="etc/truststore.properties"),
    @EndpointProperty(name="ws-security.encryption.username" value="useReqSigCert")
})
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

24.3.4.22. Adding policies

@Policy annotation

The **@Policy** annotation is defined by the **org.apache.cxf.annotations.Policy** interface. It can be placed on the SEI or the SEI methods.

This annotation is used to associate a WSDL policy with an SEI or an SEI method. The policy is specified

by providing a URI that references an XML file containing a standard **wsdl:policy** element. If a WSDL contract is to be generated from the SEI (for example, using the **java2ws** command-line tool), you can specify whether or not you want to include this policy in the WSDL.

Section 24.3.4.23, “[@Policy Properties](#)” shows the properties supported by the **@Policy** annotation.

24.3.4.23. @Policy Properties

Property	Description
uri	(Required) The location of the file containing the policy definition.
includeInWSDL	(Optional) Whether to include the policy in the generated contract, when generating WSDL. Default is true .
placement	(Optional) Specifies where in the WSDL file this documentation is to appear. For the list of possible placement values, see the section called “Placement in the WSDL contract” .
faultClass	(Optional) If the placement is set to be BINDING_OPERATION_FAULT or PORT_TYPE_OPERATION_FAULT , you must also set this property to specify which fault this policy applies to. The value is the Java class that represents the fault.

@Policies annotation

The **@Policies** annotation is defined by the **org.apache.cxf.annotations.Policies** interface. It can be placed on the SEI or these SEI methods.

This annotation provides a way of grouping multiple **@Policy** annotations into a list.

Placement in the WSDL contract

To specify where the policy should appear in the WSDL contract, you can specify the **placement** property, which is of type **Policy.Placement**. The placement can have one of the following values:

```
Policy.Placement.BINDING
Policy.Placement.BINDING_OPERATION
Policy.Placement.BINDING_OPERATION_FAULT
Policy.Placement.BINDING_OPERATION_INPUT
Policy.Placement.BINDING_OPERATION_OUTPUT
Policy.Placement.DEFAULT
Policy.Placement.PORT_TYPE
Policy.Placement.PORT_TYPE_OPERATION
Policy.Placement.PORT_TYPE_OPERATION_FAULT
Policy.Placement.PORT_TYPE_OPERATION_INPUT
```

```
Policy.Placement.PORT_TYPE_OPERATION_OUTPUT
Policy.Placement.SERVICE
Policy.Placement.SERVICE_PORT
```

Example of @Policy

The following example shows how to associate WSDL policies with the **HelloWorld** SEI and how to associate a policy with the **sayHi** method. The policies themselves are stored in XML files in the file system, under the **annotationpolicies** directory.

```
@WebService
@Policy(uri = "annotationpolicies/TestImplPolicy.xml",
    placement = Policy.Placement.SERVICE_PORT),
@Policy(uri = "annotationpolicies/TestPortTypePolicy.xml",
    placement = Policy.Placement.PORT_TYPE)
public interface HelloWorld {
    @Policy(uri = "annotationpolicies/TestOperationPTPolicy.xml",
        placement = Policy.Placement.PORT_TYPE_OPERATION),
    String sayHi(@WebParam(name = "text") String text);
}
```

Example of @Policies

You can use the **@Policies** annotation to group multiple **@Policy** annotations into a list, as shown in the following example:

```
@WebService
@Policies({
    @Policy(uri = "annotationpolicies/TestImplPolicy.xml",
        placement = Policy.Placement.SERVICE_PORT),
    @Policy(uri = "annotationpolicies/TestPortTypePolicy.xml",
        placement = Policy.Placement.PORT_TYPE)
})
public interface HelloWorld {
    @Policy(uri = "annotationpolicies/TestOperationPTPolicy.xml",
        placement = Policy.Placement.PORT_TYPE_OPERATION),
    String sayHi(@WebParam(name = "text") String text);
}
```

24.4. GENERATING WSDL

Using Maven

Once your code is annotated, you can generate a WSDL contract for your service using the **java2ws** Maven plug-in's **-wsdl** option. For a detailed listing of options for the **java2ws** Maven plug-in see [Section 44.3, “java2ws”](#).

[Example 24.8, “Generating WSDL from Java”](#) shows how to set up the **java2ws** Maven plug-in to generate WSDL.

Example 24.8. Generating WSDL from Java

```
<plugin>
```

```

<groupId>org.apache.cxf</groupId>
<artifactId>cxf-java2ws-plugin</artifactId>
<version>${cxf.version}</version>
<executions>
  <execution>
    <id>process-classes</id>
    <phase>process-classes</phase>
    <configuration>
      <className>className</className>
      <genWSDL>true</genWSDL>
    </configuration>
    <goals>
      <goal>java2ws</goal>
    </goals>
  </execution>
</executions>
</plugin>

```



NOTE

Replace the value of className with the qualified className.

Example

[Example 24.9, “Generated WSDL from an SEI”](#) shows the WSDL contract that is generated for the SEI shown in [Example 24.7, “Fully Annotated SEI”](#).

Example 24.9. Generated WSDL from anSEI

```

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://demo.eric.org/">
  xmlns:tns="http://demo.eric.org/"
  xmlns:ns1=""
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns2="http://demo.eric.org/types"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
<wsdl:types>
  <xsd:schema>
    <xs:complexType name="quote">
      <xs:sequence>
        <xs:element name="ID" type="xs:string" minOccurs="0"/>
        <xs:element name="time" type="xs:string" minOccurs="0"/>
        <xs:element name="val" type="xs:float"/>
      </xs:sequence>
    </xs:complexType>
  </xsd:schema>
</wsdl:types>
<wsdl:message name="getStockQuote">
  <wsdl:part name="stockTicker" type="xsd:string">
  </wsdl:part>
</wsdl:message>
<wsdl:message name="getStockQuoteResponse">

```

```
<wsdl:part name="updatedQuote" type="tns:quote">
</wsdl:part>
</wsdl:message>
<wsdl:portType name="quoteReporter">
<wsdl:operation name="getStockQuote">
<wsdl:input name="getQuote" message="tns:getStockQuote">
</wsdl:input>
<wsdl:output name="getQuoteResponse" message="tns:getStockQuoteResponse">
</wsdl:output>
</wsdl:operation>
</wsdl:portType>
<wsdl:binding name="quoteReporterBinding" type="tns:quoteReporter">
<soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
<wsdl:operation name="getStockQuote">
<soap:operation style="rpc" />
<wsdl:input name="getQuote">
<soap:body use="literal" />
</wsdl:input>
<wsdl:output name="getQuoteResponse">
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="quoteReporterService">
<wsdl:port name="quoteReporterPort" binding="tns:quoteReporterBinding">
<soap:address location="http://localhost:9000/quoteReporterService" />
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

[1] **Board** is an assumed class whose implementation is left to the reader.

CHAPTER 25. DEVELOPING A CONSUMER WITHOUT A WSDL CONTRACT

Abstract

You do not need a WSDL contract to develop a service consumer. You can create a service consumer from an annotated SEI. Along with the SEI you need to know the address at which the endpoint exposing the service is published, the QName of the service element that defines the endpoint exposing the service, and the QName of the port element defining the endpoint on which your consumer makes requests. This information can be specified in the SEI's annotations or provided separately.

25.1. JAVA-FIRST CONSUMER DEVELOPMENT

To create a consumer without a WSDL contract you must do the following:

1. [Create a **Service** object](#) for the service on which the consumer will invoke operations.
2. [Add a port](#) to the **Service** object.
3. [Get a proxy](#) for the service using the **Service** object's **getPort()** method.
4. [Implement the consumer's business logic.](#)

25.2. CREATING A SERVICE OBJECT

Overview

The **javax.xml.ws.Service** class represents the **wsdl:service** element which contains the definition of all of the endpoints that expose a service. As such, it provides methods that allow you to get endpoints, defined by **wsdl:port** elements, that are proxies for making remote invocations on a service.



NOTE

The **Service** class provides the abstractions that allow the client code to work with Java types as opposed to working with XML documents.

The **create()** methods

The **Service** class has two static **create()** methods that can be used to create a new **Service** object. As shown in [Example 25.1, "Service create\(\) Methods"](#), both of the **create()** methods take the QName of the **wsdl:service** element the **Service** object will represent, and one takes a URI specifying the location of the WSDL contract.



NOTE

All services publish their WSDL contracts. For SOAP/HTTP services the URI is usually the URI for the service appended with **?wsdl**.

Example 25.1. Service create() Methods

```
public static Service createURL(wsdlLocation QName serviceName) throws WebServiceException
public static Service createQName(serviceName) throws WebServiceException
```

The value of the **serviceName** parameter is a QName. The value of its namespace part is the target namespace of the service. The service's target namespace is specified in the **targetNamespace** property of the **@WebService** annotation. The value of the QName's local part is the value of **wsdl:service** element's **name** attribute. You can determine this value in one of the following ways: . It is specified in the **serviceName** property of the **@WebService** annotation.

1. You append **Service** to the value of the **name** property of the **@WebService** annotation.
2. You append **Service** to the name of the SEI.



IMPORTANT

Programmatically-created CXF consumers deployed in OSGi environments require special handling to avoid the likelihood of incurring **ClassNotFoundExceptions**. For each bundle that contains programmatically-created CXF consumers, you need to create a singleton CXF default bus and ensure that all of the bundle's CXF consumers use it. Without this safeguard, one bundle could be assigned the CXF default bus created in another bundle, which could cause the inheriting bundle to fail.

For example, suppose bundle A did not explicitly set a CXF default bus and was assigned the CXF default bus created in bundle B. If the CXF bus in bundle A needed to be configured with additional features (such as SSL or WS-Security) or needed to load certain classes or resources from the application in bundle A, it would fail. This is so because the CXF bus instance sets a thread context class loader (TCCL) as the bundle class loader of the bundle that created it (in this case bundle B). Furthermore, certain frameworks, such as wss4j (implements WS-Security in CXF) use the TCCL to load resources, such as callback handler classes or other property files, from inside the bundle. Because bundle A is assigned bundle B's default CXF bus and it's TCCL, the wss4j layer cannot load the required resources from bundle A, which results in **ClassNotFoundException** errors.

To create the singleton CXF default bus, insert this code:

```
BusFactory.setThreadDefaultBus(BusFactory.newInstance().createBus());
```

at the beginning of the **main** method that creates the service object, as shown in [the section called "Example"](#).

Example

[Example 25.2, "Creating a Service Object"](#) shows code for creating a **Service** object for the SEI shown in [Example 24.7, "Fully Annotated SEI"](#).

Example 25.2. Creating a Service Object

```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
    public static void main(String args[])
    {
        QName serviceName = new QName("http://com.fusesource/demo", "Client");
        Service service = new Service(serviceName);
        Client client = service.getPort(Client.class);
        System.out.println(client.getHello("World"));
    }
}
```

```

    }
    BusFactory.setThreadDefaultBus(BusFactory.newInstance().createBus());
    QName serviceName = new QName("http://demo.redhat.com", "stockQuoteReporter");
    Service s = Service.create(serviceName);
    ...
}
}

```

The code in [Example 25.2, “Creating a **Service** Object”](#) does the following:

Creates a singleton CXF default bus that is available to all CXF consumers of the service.

Builds the QName for the service using the **targetNamespace** property and the **name** property of the **@WebService** annotation.

Calls the single parameter **create()** method to create a new **Service** object.



NOTE

Using the single parameter **create()** frees you from having any dependencies on accessing a WSDL contract.

25.3. ADDING A PORT TO A SERVICE

Overview

The endpoint information for a service is defined in a **wsdl:port** element, and the **Service** object creates a proxy instance for each of the endpoints defined in a WSDL contract, if one is specified. If you do not specify a WSDL contract when you create your **Service** object, the **Service** object has no information about the endpoints that implement your service, and therefore cannot create any proxy instances. In this case, you must provide the **Service** object with the information needed to represent a **wsdl:port** element using the **addPort()** method.

The **addPort()** method

The **Service** class defines an **addPort()** method, shown in [Example 25.3, “The **addPort\(\)** Method”](#), that is used in cases where there is no WSDL contract available to the consumer implementation. The **addPort()** method allows you to give a **Service** object the information, which is typically stored in a **wsdl:port** element, necessary to create a proxy for a service implementation.

Example 25.3. The **addPort()** Method

```
addPort(QName portName, String bindingId, String endpointAddress) WebServiceException
```

The value of the **portName** is a QName. The value of its namespace part is the target namespace of the service. The service’s target namespace is specified in the **targetNamespace** property of the **@WebService** annotation. The value of the QName’s local part is the value of **wsdl:port** element’s **name** attribute. You can determine this value in one of the following ways:

1. Specify it in the **portName** property of the **@WebService** annotation.
2. Append **Port** to the value of the **name** property of the **@WebService** annotation.

3. Append **Port** to the name of the SEI.

The value of the **bindingId** parameter is a string that uniquely identifies the type of binding used by the endpoint. For a SOAP binding you use the standard SOAP namespace:

<http://schemas.xmlsoap.org/soap/>. If the endpoint is not using a SOAP binding, the value of the **bindingId** parameter is determined by the binding developer. The value of the **endpointAddress** parameter is the address where the endpoint is published. For a SOAP/HTTP endpoint, the address is an HTTP address. Transports other than HTTP use different address schemes.

Example

[Example 25.4, "Adding a Port to a Service Object"](#) shows code for adding a port to the **Service** object created in [Example 25.2, "Creating a Service Object"](#).

Example 25.4. Adding a Port to a Service Object

```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
    public static void main(String args[])
    {
        ...
        QName portName = new QName("http://demo.redhat.com", "stockQuoteReporterPort");
        s.addPort(portName,
                  "http://schemas.xmlsoap.org/soap/",
                  "http://localhost:9000/StockQuote");
        ...
    }
}
```

The code in [Example 25.4, "Adding a Port to a Service Object"](#) does the following:

Creates the QName for the **portName** parameter.

Calls the **addPort()** method.

Specifies that the endpoint uses a SOAP binding.

Specifies the address where the endpoint is published.

25.4. GETTING A PROXY FOR AN ENDPOINT

Overview

A service proxy is an object that provides all of the methods exposed by a remote service and handles all of the details required to make the remote invocations. The **Service** object provides service proxies for all of the endpoints it is aware of through the **getPort()** method. Once you have a service proxy, you can invoke its methods. The proxy forwards the invocation to the remote service endpoint using the connection details specified in the service's contract.

The `getPort()` method

The `getPort()` method, shown in [Example 25.5, “The `getPort\(\)` Method”](#), returns a service proxy for the specified endpoint. The returned proxy is of the same class as the SEI.

Example 25.5. The `getPort()` Method

```
public<T> T getPort(QName portName, Class<T> serviceEndpointInterface) throws WebServiceException
```

The value of the `portName` parameter is a QName that identifies the `wsdl:port` element that defines the endpoint for which the proxy is created. The value of the `serviceEndpointInterface` parameter is the fully qualified name of the SEI.



NOTE

When you are working without a WSDL contract the value of the `portName` parameter is typically the same as the value used for the `portName` parameter when calling `addPort()`.

Example

[Example 25.6, “Getting a Service Proxy”](#) shows code for getting a service proxy for the endpoint added in [Example 25.4, “Adding a Port to a `Service` Object”](#).

Example 25.6. Getting a Service Proxy

```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
    public static void main(String args[])
    {
        ...
        quoteReporter proxy = s.getPort(portName, quoteReporter.class);
        ...
    }
}
```

25.5. IMPLEMENTING THE CONSUMER’S BUSINESS LOGIC

Overview

Once you instantiate a service proxy for a remote endpoint, you can invoke its methods as if it were a local object. The calls block until the remote method completes.



NOTE

If a method is annotated with the `@OneWay` annotation, the call returns immediately.

Example

[Example 25.7, "Consumer Implemented without a WSDL Contract"](#) shows a consumer for the service defined in [Example 24.7, "Fully Annotated SEI"](#).

Example 25.7. Consumer Implemented without a WSDL Contract

```
package com.fusesource.demo;

import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
    public static void main(String args[])
    {
        QName serviceName = new QName("http://demo.eric.org", "stockQuoteReporter");
        Service s = Service.create(serviceName);

        QName portName = new QName("http://demo.eric.org", "stockQuoteReporterPort");
        s.addPort(portName, "http://schemas.xmlsoap.org/soap/",
                  "http://localhost:9000/EricStockQuote");

        quoteReporter proxy = s.getPort(portName, quoteReporter.class);

        Quote quote = proxy.getQuote("ALPHA");
        System.out.println("Stock "+quote.getID()+" is worth "+quote.getVal()+" as of
                           "+quote.getTime());
    }
}
```

The code in [Example 25.7, "Consumer Implemented without a WSDL Contract"](#) does the following:

Creates a **Service** object.

Adds an endpoint definition to the **Service** object.

Gets a service proxy from the **Service** object.

Invokes an operation on the service proxy.

CHAPTER 26. A STARTING POINT WSDL CONTRACT

26.1. SAMPLE WSDL CONTRACT

Example 26.1, “HelloWorld WSDL Contract” shows the HelloWorld WSDL contract. This contract defines a single interface, Greeter, in the **wsdl:portType** element. The contract also defines the endpoint which will implement the service in the **wsdl:port** element.

Example 26.1. HelloWorld WSDL Contract

```
<?xml version="1.0" encoding=";UTF-8"?>
<wsdl:definitions name="HelloWorld"
    targetNamespace="http://apache.org/hello_world_soap_http"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://apache.org/hello_world_soap_http"
    xmlns:x1="http://apache.org/hello_world_soap_http/types"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <wsdl:types>
        <schema targetNamespace="http://apache.org/hello_world_soap_http/types"
            xmlns="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified">
            <element name="sayHiResponse">
                <complexType>
                    <sequence>
                        <element name="responseType" type="string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="greetMe">
                <complexType>
                    <sequence>
                        <element name="requestType" type="string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="greetMeResponse">
                <complexType>
                    <sequence>
                        <element name="responseType" type="string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="greetMeOneWay">
                <complexType>
                    <sequence>
                        <element name="requestType" type="string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="pingMe">
                <complexType/>
            </element>
            <element name="pingMeResponse">
```

```
<complexType/>
</element>
<element name="faultDetail">
<complexType>
<sequence>
<element name="minor" type="short"/>
<element name="major" type="short"/>
</sequence>
</complexType>
</element>
</schema>
</wsdl:types>

<wsdl:message name="sayHiRequest">
<wsdl:part element="x1:sayHi" name="in"/>
</wsdl:message>
<wsdl:message name="sayHiResponse">
<wsdl:part element="x1:sayHiResponse" name="out"/>
</wsdl:message>
<wsdl:message name="greetMeRequest">
<wsdl:part element="x1:greetMe" name="in"/>
</wsdl:message>
<wsdl:message name="greetMeResponse">
<wsdl:part element="x1:greetMeResponse" name="out"/>
</wsdl:message>
<wsdl:message name="greetMeOneWayRequest">
<wsdl:part element="x1:greetMeOneWay" name="in"/>
</wsdl:message>
<wsdl:message name="pingMeRequest">
<wsdl:part name="in" element="x1:pingMe"/>
</wsdl:message>
<wsdl:message name="pingMeResponse">
<wsdl:part name="out" element="x1:pingMeResponse"/>
</wsdl:message>
<wsdl:message name="pingMeFault">
<wsdl:part name="faultDetail" element="x1:faultDetail"/>
</wsdl:message>

<wsdl:portType name="Greeter">
<wsdl:operation name="sayHi">
<wsdl:input message="tns:sayHiRequest" name="sayHiRequest"/>
<wsdl:output message="tns:sayHiResponse" name="sayHiResponse"/>
</wsdl:operation>

<wsdl:operation name="greetMe">
<wsdl:input message="tns:greetMeRequest" name="greetMeRequest"/>
<wsdl:output message="tns:greetMeResponse" name="greetMeResponse"/>
</wsdl:operation>

<wsdl:operation name="greetMeOneWay">
<wsdl:input message="tns:greetMeOneWayRequest" name="greetMeOneWayRequest"/>
</wsdl:operation>

<wsdl:operation name="pingMe">
<wsdl:input name="pingMeRequest" message="tns:pingMeRequest"/>
<wsdl:output name="pingMeResponse" message="tns:pingMeResponse"/>
```

```
<wsdl:fault name="pingMeFault" message="tns:pingMeFault"/>
</wsdl:operation>
</wsdl:portType>

<wsdl:binding name="Greeter_SOAPBinding" type="tns:Greeter">
  ...
</wsdl:binding>

<wsdl:service name="SOAPService">
  <wsdl:port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <soap:address location="http://localhost:9000/SOAPContext/SoapPort"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

The Greeter interface defined in [Example 26.1, “HelloWorld WSDL Contract”](#) defines the following operations:

sayHi – Has a single output parameter, of **xsd:string**.

greetMe – Has an input parameter, of **xsd:string**, and an output parameter, of **xsd:string**.

greetMeOneWay – Has a single input parameter, of **xsd:string**. Because this operation has no output parameters, it is optimized to be a oneway invocation (that is, the consumer does not wait for a response from the server).

pingMe – Has no input parameters and no output parameters, but it can raise a fault exception.

CHAPTER 27. TOP-DOWN SERVICE DEVELOPMENT

Abstract

In the top-down method of developing a service provider you start from a WSDL document that defines the operations and methods the service provider will implement. Using the WSDL document, you generate starting point code for the service provider. Adding the business logic to the generated code is done using normal Java programming APIs.

27.1. OVERVIEW OF JAX-WS SERVICE PROVIDER DEVELOPMENT

Once you have a WSDL document, the process for developing a JAX-WS service provider is as follows:

1. [Section 27.2, “Generating the Starting Point Code”](#) starting point code.
2. [Implement](#) the service provider’s operations.
3. [Chapter 31, Publishing a Service](#) the implemented service.

27.2. GENERATING THE STARTING POINT CODE

Overview

JAX-WS specifies a detailed mapping from a service defined in WSDL to the Java classes that will implement that service as a service provider. The logical interface, defined by the **wsdl:portType** element, is mapped to a service endpoint interface (SEI). Any complex types defined in the WSDL are mapped into Java classes following the mapping defined by the Java Architecture for XML Binding (JAXB) specification. The endpoint defined by the **wsdl:service** element is also generated into a Java class that is used by consumers to access service providers implementing the service.

The **cxf-codegen-plugin** Maven plug-in generates this code. It also provides options for generating starting point code for your implementation. The code generator provides a number of options for controlling the generated code.

Running the code generator

[Example 27.1, “Service Code Generation”](#) shows how to use the code generator to generate starting point code for a service.

Example 27.1. Service Code Generation

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>${cxf.version}</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <sourceRoot>outputDir</sourceRoot>
        <wsdlOptions>
          <wsdlOption>
```

```

<wsdl>wsdl</wsdl>
<extraargs>
    <extraarg>-server</extraarg>
    <extraarg>-impl</extraarg>
</extraargs>
</wsdlOption>
</wsdlOptions>
</configuration>
<goals>
    <goal>wsdl2java</goal>
</goals>
</execution>
</executions>
</plugin>

```

This does the following:

- The **-impl** option generates a shell implementation class for each **wsdl:portType** element in the WSDL contract.
- The **-server** option generates a simple **main()** to run your service provider as a stand alone application.
- The **sourceRoot** specifies that the generated code is written to a directory called **outputDir**.
- **wsdl** element specifies the WSDL contract from which code is generated.

For a complete list of the options for the code generator see [Section 44.2, “cxfrs-codegen-plugin”](#).

Generated code

[Table 27.1, “Generated Classes for a Service Provider”](#) describes the files generated for creating a service provider.

Table 27.1. Generated Classes for a Service Provider

File	Description
portTypeName.java	The SEI. This file contains the interface your service provider implements. You should not edit this file.
serviceName.java	The endpoint. This file contains the Java class consumers use to make requests on the service.
portTypeNamesImpl.java	The skeleton implementation class. Modify this file to build your service provider.
portTypeNamesServer.java	A basic server mainline that allows you to deploy your service provider as a stand alone process. For more information see Chapter 31, Publishing a Service .

In addition, the code generator will generate Java classes for all of the types defined in the WSDL contract.

Generated packages

The generated code is placed into packages based on the namespaces used in the WSDL contract. The classes generated to support the service (based on the **wsdl:portType** element, the **wsdl:service** element, and the **wsdl:port** element) are placed in a package based on the target namespace of the WSDL contract. The classes generated to implement the types defined in the **types** element of the contract are placed in a package based on the **targetNamespace** attribute of the **types** element.

The mapping algorithm is as follows:

1. The leading **http://** or **urn://** are stripped off the namespace.
2. If the first string in the namespace is a valid Internet domain, for example it ends in **.com** or **.gov**, then the leading **www.** is stripped off the string, and the two remaining components are flipped.
3. If the final string in the namespace ends with a file extension of the pattern **.xxx** or **.xx**, then the extension is stripped.
4. The remaining strings in the namespace are appended to the resulting string and separated by dots.
5. All letters are made lowercase.

27.3. IMPLEMENTING THE SERVICE PROVIDER

Generating the implementation code

You generate the implementation class used to build your service provider with the code generator's **-impl** flag.



NOTE

If your service's contract includes any custom types defined in XML Schema, you must ensure that the classes for the types are generated and available.

For more information on using the code generator see [Section 44.2, “cxfrs-codegen-plugin”](#).

Generated code

The implementation code consists of two files:

- **portTypeName.java** – The service interface(SEI) for the service.
- **portTypeNamImpl.java** – The class you will use to implement the operations defined by the service.

Implement the operation's logic

To provide the business logic for your service's operations complete the stub methods in **portTypeNamImpl.java**. You usually use standard Java to implement the business logic. If your service

uses custom XML Schema types, you must use the generated classes for each type to manipulate them. There are also some Apache CXF specific APIs that can be used to access some advanced features.

Example

For example, an implementation class for the service defined in [Example 26.1, “HelloWorld WSDL Contract”](#) may look like [Example 27.2, “Implementation of the Greeter Service”](#). Only the code portions highlighted in bold must be inserted by the programmer.

Example 27.2. Implementation of the Greeter Service

```
package demo.hw.server;

import org.apache.hello_world_soap_http.Greeter;

@javax.jws.WebService(portName = "SoapPort", serviceName = "SOAPService",
                      targetNamespace = "http://apache.org/hello_world_soap_http",
                      endpointInterface = "org.apache.hello_world_soap_http.Greeter")

public class GreeterImpl implements Greeter {

    public String greetMe(String me) {
        System.out.println("Executing operation greetMe"); System.out.println("Message received: " +
me + "\n"); return "Hello " + me;
    }

    public void greetMeOneWay(String me) {
        System.out.println("Executing operation greetMeOneWay\n"); System.out.println("Hello there
" + me);
    }

    public String sayHi() {
        System.out.println("Executing operation sayHi\n"); return "Bonjour";
    }

    public void pingMe() throws PingMeFault {
        FaultDetail faultDetail = new FaultDetail(); faultDetail.setMajor((short)2);
        faultDetail.setMinor((short)1); System.out.println("Executing operation pingMe, throwing
PingMeFault exception\n"); throw new PingMeFault("PingMeFault raised by server", faultDetail);
    }
}
```

CHAPTER 28. DEVELOPING A CONSUMER FROM A WSDL CONTRACT

Abstract

One way of creating a consumer is to start from a WSDL contract. The contract defines the operations, messages, and transport details of the service on which a consumer makes requests. The starting point code for the consumer is generated from the WSDL contract. The functionality required by the consumer is added to the generated code.

28.1. GENERATING THE STUB CODE

Overview

The **cxfrs-codegen-plugin** Maven plug-in generates the stub code from the WSDL contract. The stub code provides the supporting code that is required to invoke operations on the remote service.

For consumers, the **cxfrs-codegen-plugin** Maven plug-in generates the following types of code:

- Stub code – Supporting files for implementing a consumer.
- Starting point code – Sample code that connects to the remote service and invokes every operation on the remote service.

Generating the consumer code

To generate consumer code use the **cxfrs-codegen-plugin** Maven plug-in. [Example 28.1, “Consumer Code Generation”](#) shows how to use the code generator to generate consumer code.

Example 28.1. Consumer Code Generation

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxfrs-codegen-plugin</artifactId>
  <version>${cxfrs.version}</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <sourceRoot>outputDir</sourceRoot>
        <wsdlOptions>
          <wsdlOption>
            <wsdl>wsdl</wsdl>
            <extraargs>
              <extraarg>-client</extraarg>
            </extraargs>
          </wsdlOption>
        </wsdlOptions>
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

```

</execution>
</executions>
</plugin>
```

Where `outputDir` is the location of a directory where the generated files are placed and `wsdl` specifies the WSDL contract's location. The **-client** option generates starting point code for the consumer's `main()` method.

For a complete list of the arguments available for the **cxfrs-codegen-plugin** Maven plug-in see [Section 44.2, "cxfrs-codegen-plugin"](#).

Generated code

The code generation plug-in generates the following Java packages for the contract shown in [Example 26.1, "HelloWorld WSDL Contract"](#):

- **org.apache.hello_world_soap_http** – This package is generated from the http://apache.org/hello_world_soap_http target namespace. All of the WSDL entities defined in this namespace (for example, the Greeter port type and the SOAPService service) map to Java classes this Java package.
- **org.apache.hello_world_soap_http.types** – This package is generated from the http://apache.org/hello_world_soap_http/types target namespace. All of the XML types defined in this namespace (that is, everything defined in the **wsdl:types** element of the HelloWorld contract) map to Java classes in this Java package.

The stub files generated by the **cxfrs-codegen-plugin** Maven plug-in fall into the following categories:

- Classes representing WSDL entities in the **org.apache.hello_world_soap_http** package. The following classes are generated to represent WSDL entities:
 - Greeter – A Java interface that represents the Greeter **wsdl:portType** element. In JAX-WS terminology, this Java interface is the service endpoint interface (SEI).
 - **SOAPService** – A Java service class (extending **javax.xml.ws.Service**) that represents the SOAPService **wsdl:service** element.
 - PingMeFault – A Java exception class (extending **java.lang.Exception**) that represents the pingMeFault **wsdl:fault** element.
- Classes representing XML types in the **org.objectweb.hello_world_soap_http.types** package. In the HelloWorld example, the only generated types are the various wrappers for the request and reply messages. Some of these data types are useful for the asynchronous invocation model.

28.2. IMPLEMENTING A CONSUMER

Overview

To implement a consumer when starting from a WSDL contract, you must use the following stubs:

- Service class
- SEI

Using these stubs, the consumer code instantiates a service proxy to make requests on the remote service. It also implements the consumer's business logic.

Generated service class

[Example 28.2, “Outline of a Generated Service Class”](#) shows the typical outline of a generated service class, **ServiceName_Service**^[2], which extends the **javax.xml.ws.Service** base class.

Example 28.2. Outline of a Generated Service Class

```
@WebServiceClient(name="..." targetNamespace="..."
    wsdlLocation="...")
public class ServiceName extends javax.xml.ws.Service
{
    ...
    public ServiceName(URL wsdlLocation, QName serviceName) { }

    public ServiceName() { }

    // Available only if you specify '-fe cxf' option in wsdl2java
    public ServiceName(Bus bus) { }

    @WebEndpoint(name="...")
    public SEI getPortName() { }
    .
    .
}
}
```

The **ServiceName** class in [Example 28.2, “Outline of a Generated Service Class”](#) defines the following methods:

- **ServiceName(URL wsdlLocation, QName serviceName)** – Constructs a service object based on the data in the **wsdl:service** element with the QName *ServiceName* service in the WSDL contract that is obtainable from *wsdlLocation*.
- **ServiceName()** – The default constructor. It constructs a service object based on the service name and the WSDL contract that were provided at the time the stub code was generated (for example, when running the **wsdl2java** tool). Using this constructor presupposes that the WSDL contract remains available at a specified location.
- **ServiceName(Bus bus)** – (CXF specific) An additional constructor that enables you to specify the Bus instance used to configure the Service. This can be useful in the context of a multi-threaded application, where multiple Bus instances can be associated with different threads. This constructor provides a simple way of ensuring that the Bus that you specify is the one that is used with this Service. Only available if you specify the **-fe cxf** option when invoking the **wsdl2java** tool.
- **getPortName()** – Returns a proxy for the endpoint defined by the **wsdl:port** element with the **name** attribute equal to *PortName*. A getter method is generated for every **wsdl:port** element defined by the *ServiceName* service. A **wsdl:service** element that contains multiple endpoint definitions results in a generated service class with multiple **getPortName()** methods.

Service endpoint interface

For every interface defined in the original WSDL contract, you can generate a corresponding SEI. A service endpoint interface is the Java mapping of a **wsdl:portType** element. Each operation defined in the original **wsdl:portType** element maps to a corresponding method in the SEI. The operation's parameters are mapped as follows: . The input parameters are mapped to method arguments.

1. The first output parameter is mapped to a return value.
2. If there is more than one output parameter, the second and subsequent output parameters map to method arguments (moreover, the values of these arguments must be passed using **Holder** types).

For example, [Example 28.3, "The Greeter Service Endpoint Interface"](#) shows the Greeter SEI, which is generated from the **wsdl:portType** element defined in [Example 26.1, "HelloWorld WSDL Contract"](#). For simplicity, [Example 28.3, "The Greeter Service Endpoint Interface"](#) omits the standard JAXB and JAX-WS annotations.

Example 28.3. The Greeter Service Endpoint Interface

```
package org.apache.hello_world_soap_http;
...
public interface Greeter
{
    public String sayHi();
    public String greetMe(String requestType);
    public void greetMeOneWay(String requestType);
    public void pingMe() throws PingMeFault;
}
```

Consumer main function

[Example 28.4, "Consumer Implementation Code"](#) shows the code that implements the HelloWorld consumer. The consumer connects to the SoapPort port on the SOAPService service and then proceeds to invoke each of the operations supported by the Greeter port type.

Example 28.4. Consumer Implementation Code

```
package demo.hw.client;

import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import org.apache.hello_world_soap_http.Greeter;
import org.apache.hello_world_soap_http.PingMeFault;
import org.apache.hello_world_soap_http.SOAPService;

public final class Client {

    private static final QName SERVICE_NAME =
        new QName("http://apache.org/hello_world_soap_http",
                  "SOAPService");

    private Client()
```

```

{
}

public static void main(String args[]) throws Exception
{
if (args.length == 0)
{
    System.out.println("please specify wsdl");
    System.exit(1);
}

URL wsdlURL;
File wsdlFile = new File(args[0]);
if (wsdlFile.exists())
{
    wsdlURL = wsdlFile.toURL();
}
else
{
    wsdlURL = new URL(args[0]);
}

System.out.println(wsdlURL);
SOAPService ss = new SOAPService(wsdlURL,SERVICE_NAME);
Greeter port = ss.getSoapPort();
String resp;

System.out.println("Invoking sayHi...");
resp = port.sayHi();
System.out.println("Server responded with: " + resp);
System.out.println();

System.out.println("Invoking greetMe...");
resp = port.greetMe(System.getProperty("user.name"));
System.out.println("Server responded with: " + resp);
System.out.println();

System.out.println("Invoking greetMeOneWay...");
port.greetMeOneWay(System.getProperty("user.name"));
System.out.println("No response from server as method is OneWay");
System.out.println();

try {
    System.out.println("Invoking pingMe, expecting exception...");
    port.pingMe();
} catch (PingMeFault ex) {
    System.out.println("Expected exception: PingMeFault has occurred.");
    System.out.println(ex.toString());
}
System.exit(0);
}
}

```

The **Client.main()** method from [Example 28.4, “Consumer Implementation Code”](#) proceeds as follows:

Provided that the Apache CXF runtime classes are on your classpath, the runtime is implicitly initialized. There is no need to call a special function to initialize Apache CXF.

The consumer expects a single string argument that gives the location of the WSDL contract for `HelloWorld`. The WSDL contract's location is stored in **wsdlURL**.

You create a service object using the constructor that requires the WSDL contract's location and service name. Call the appropriate **getPortName()** method to obtain an instance of the required port. In this case, the `SOAPService` service supports only the `SoapPort` port, which implements the **Greeter** service endpoint interface.

The consumer invokes each of the methods supported by the Greeter service endpoint interface.

In the case of the **pingMe()** method, the example code shows how to catch the `PingMeFault` fault exception.

Client proxy generated with -fe cxf option

If you generate your client proxy by specifying the **-fe cxf** option in `wsdl2java` (thereby selecting the **cxf** frontend), the generated client proxy code is better integrated with Java 7. In this case, when you call a **getServiceNamePort()** method, you get back a type that is a sub-interface of the SEI and implements the following additional interfaces:

- **java.lang.AutoCloseable**
- **javax.xml.ws.BindingProvider** (JAX-WS 2.0)
- **org.apache.cxf.endpoint.Client**

To see how this simplifies working with a client proxy, consider the following Java code sample, written using a standard JAX-WS proxy object:

```
// Programming with standard JAX-WS proxy object
//
(ServiceNamePortType port = service.getServiceNamePort();
((BindingProvider)port).getRequestContext()
    .put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, address);
port.serviceMethod(...);
((Closeable)port).close();
```

And compare the preceding code with the following equivalent code sample, written using code generated by the **cxf** frontend:

```
// Programming with proxy generated using '-fe cxf' option
//
try (ServiceNamePortTypeProxy port = service.getServiceNamePort()) {
    port.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, address);
    port.serviceMethod(...);
}
```

[2] If the **name** attribute of the **wsdl:service** element ends in `_Service` the `_Service` is not used.

CHAPTER 29. FINDING WSDL AT RUNTIME

Abstract

Hard coding the location of WSDL documents into an application is not scalable. In real deployment environments, you will want to allow the WSDL document's location be resolved at runtime. Apache CXF provides a number of tools to make this possible.

29.1. MECHANISMS FOR LOCATING THE WSDL DOCUMENT

When developing consumers using the JAX-WS APIs you must provide a hard coded path to the WSDL document that defines your service. While this is OK in a small environment, using hard coded paths does not work well in enterprise deployments.

To address this issue, Apache CXF provides three mechanisms for removing the requirement of using hard coded paths:

- [Section 29.2, "Instantiating a Proxy by Injection"](#)
- [Section 29.3, "Using a JAX-WS Catalog"](#)
- [Section 29.4, "Using a contract resolver"](#)



NOTE

Injecting the proxy into your implementation code is generally the best option because it is the easiest to implement. It requires only a client endpoint and a configuration file for injecting and instantiating the service proxy.

29.2. INSTANTIATING A PROXY BY INJECTION

Overview

Apache CXF's use of the Spring Framework allows you to avoid the hassle of using the JAX-WS APIs to create service proxies. It allows you to define a client endpoint in a configuration file and then inject a proxy directly into the implementation code. When the runtime instantiates the implementation object, it will also instantiate a proxy for the external service based on the configuration. The implementation is handed by reference to the instantiated proxy.

Because the proxy is instantiated using information in the configuration file, the WSDL location does not need to be hard coded. It can be changed at deployment time. You can also specify that the runtime should search the application's classpath for the WSDL.

Procedure

To inject a proxy for an external service into a service provider's implementation do the following:

1. Deploy the required WSDL documents in a well known location that all parts of the application can access.

**NOTE**

If you are deploying the application as a WAR file, it is recommended that you place all of the WSDL documents and XML Schema documents in the **WEB-INF/wsdl** folder of the WAR.

**NOTE**

If you are deploying the application as a JAR file, it is recommended that you place all of the WSDL documents and XML Schema documents in the **META-INF/wsdl** folder of the JAR.

2. [Configure](#) a JAX-WS client endpoint for the proxy that is being injected.
3. [Inject](#) the proxy into your service provider using the **@Resource** annotation.

Configuring the proxy

You configure a JAX-WS client endpoint using the **jaxws:client** element in your application's configuration file. This tells the runtime to instantiate a **org.apache.cxf.jaxws.JaxWsClientProxy** object with the specified properties. This object is the proxy that will be injected into the service provider.

At a minimum you need to provide values for the following attributes:

- **id**—Specifies the ID used to identify the client to be injected.
- **serviceClass**—Specifies the SEI of the service on which the proxy makes requests.

[Example 29.1, "Configuration for a Proxy to be Injected into a Service Implementation"](#) shows the configuration for a JAX-WS client endpoint.

Example 29.1. Configuration for a Proxy to be Injected into a Service Implementation

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
  ...">
  <jaxws:client id="bookClient"
    serviceClass="org.apache.cxf.demo.BookService"
    wsdlLocation="classpath:books.wsdl"/>
  ...
</beans>
```

**NOTE**

In [Example 29.1, "Configuration for a Proxy to be Injected into a Service Implementation"](#) the **wsdlLocation** attribute instructs the runtime to load the WSDL from the classpath. If **books.wsdl** is on the classpath, the runtime will be able to find it.

For more information on configuring a JAX-WS client see [Section 17.2, “Configuring Consumer Endpoints”](#).

Coding the provider implementation

You inject the configured proxy into a service implementation as a resource using the **@Resource** as shown in [Example 29.2, “Injecting a Proxy into a Service Implementation”](#).

Example 29.2. Injecting a Proxy into a Service Implementation

```
package demo.hw.server;

import org.apache.hello_world_soap_http.Greeter;

@javax.jws.WebService(portName = "SoapPort", serviceName = "SOAPService",
                      targetNamespace = "http://apache.org/hello_world_soap_http",
                      endpointInterface = "org.apache.hello_world_soap_http.Greeter")
public class StoreImpl implements Store {

    @Resource(name="bookClient") private BookService proxy;

}
```

The annotation's **name** property corresponds to the value of the JAX-WS client's **id** attribute. The configured proxy is injected into the **BookService** object declared immediately after the annotation. You can use this object to make invocations on the proxy's external service.

29.3. USING A JAX-WS CATALOG

Overview

The JAX-WS specification mandates that all implementations support:

a standard catalog facility to be used when resolving any Web service document that is part of the description of a Web service, specifically WSDL and XML Schema documents.

This catalog facility uses the XML catalog facility specified by OASIS. All of the JAX-WS APIs and annotation that take a WSDL URI use the catalog to resolve the WSDL document's location.

This means that you can provide an XML catalog file that rewrites the locations of your WSDL documents to suite specific deployment environments.

Writing the catalog

JAX-WS catalogs are standard XML catalogs as defined by the [OASIS XML Catalogs 1.1](#) specification. They allow you to specify mapping:

- a document's public identifier and/or a system identifier to a URI.
- the URI of a resource to another URI.

[Table 29.1, “Common JAX-WS Catalog Elements”](#) lists some common elements used for WSDL location resolution.

Table 29.1. Common JAX-WS Catalog Elements

Element	Description
uri	Maps a URI to an alternate URI.
rewriteURI	Rewrites the beginning of a URI. For example, this element allows you to map all URLs that start with http://cxf.apache.org to URLs that start with <code>classpath:</code> .
uriSuffix	Maps a URI to an alternate URI based on the suffix of the original URI. For example you could map all URLs that end in <code>foo.xsd</code> to <code>classpath:foo.xsd</code> .

Packaging the catalog

The JAX-WS specification mandates that the catalog used to resolve WSDL and XML Schema documents is assembled using all available resources named **META-INF/jax-ws-catalog.xml**. If your application is packaged into a single JAR, or WAR, you can place the catalog into a single file.

If your application is packaged as multiple JARs, you can split the catalog into a number of files. Each catalog file could be modularized to only deal with WSDLs accessed by the code in the specific JARs.

29.4. USING A CONTRACT RESOLVER

Overview

The most involved mechanism for resolving WSDL document locations at runtime is to implement your own custom contract resolver. This requires that you provide an implementation of the Apache CXF specific ServiceContractResolver interface. You also need to register your custom resolver with the bus.

Once properly registered, the custom contract resolver will be used to resolve the location of any required WSDL and schema documents.

Implementing the contract resolver

A contract resolver is an implementation of the org.apache.cxf.endpoint.ServiceContractResolver interface. As shown in [Example 29.3, “ServiceContractResolver Interface”](#), this interface has a single method, **getContractLocation()**, that needs to be implemented. **getContractLocation()** takes the QName of a service and returns the URI for the service’s WSDL contract.

Example 29.3. ServiceContractResolver Interface

```
public interface ServiceContractResolver
{
    URI getContractLocation(QName qname);
}
```

The logic used to resolve the WSDL contract's location is application specific. You can add logic that resolves contract locations from a UDDI registry, a database, a custom location on a file system, or any other mechanism you choose.

Registering the contract resolver programmatically

Before the Apache CXF runtime will use your contract resolver, you must register it with a contract resolver registry. Contract resolver registries implement the `org.apache.cxf.endpoint.ServiceContractResolverRegistry` interface. However, you do not need to implement your own registry. Apache CXF provides a default implementation in the `org.apache.cxf.endpoint.ServiceContractResolverRegistryImpl` class.

To register a contract resolver with the default registry you do the following:

1. Get a reference to the default bus object.
2. Get the service contract registry from the bus using the bus' `getExtension()` method.
3. Create an instance of your contract resolver.
4. Register your contract resolver with the registry using the registry's `register()` method.

[Example 29.4, "Registering a Contract Resolver"](#) shows the code for registering a contract resolver with the default registry.

Example 29.4. Registering a Contract Resolver

```
BusFactory bf=BusFactory.newInstance();
Bus bus=bf.createBus();

ServiceContractResolverRegistry registry = busgetExtension(ServiceContractResolverRegistry);

JarServiceContractResolver resolver = new JarServiceContractResolver();

registry.register(resolver);
```

The code in [Example 29.4, "Registering a Contract Resolver"](#) does the following:

Gets a bus instance.

Gets the bus' contract resolver registry.

Creates an instance of a contract resolver.

Registers the contract resolver with the registry.

Registering a contract resolver using configuration

You can also implement a contract resolver so that it can be added to a client through configuration. The contract resolver is implemented in such a way that when the runtime reads the configuration and instantiates the resolver, the resolver registers itself. Because the runtime handles the initialization, you can decide at runtime if a client needs to use the contract resolver.

To implement a contract resolver so that it can be added to a client through configuration do the following:

1. Add an **init()** method to your contract resolver implementation.
2. Add logic to your **init()** method that registers the contract resolver with the contract resolver registry as shown in [Example 29.4, "Registering a Contract Resolver"](#).
3. Decorate the **init()** method with the **@PostConstruct** annotation.

[Example 29.5, "Service Contract Resolver that can be Registered Using Configuration"](#) shows a contract resolver implementation that can be added to a client using configuration.

Example 29.5. Service Contract Resolver that can be Registered Using Configuration

```
import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.xml.namespace.QName;

import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;

public class UddiResolver implements ServiceContractResolver
{
    private Bus bus;
    ...

    @PostConstruct
    public void init()
    {
        BusFactory bf=BusFactory.newInstance();
        Bus bus=bf.createBus();
        if (null != bus)
        {
            ServiceContractResolverRegistry resolverRegistry =
busgetExtension(ServiceContractResolverRegistry.class);
            if (resolverRegistry != null)
            {
                resolverRegistry.register(this);
            }
        }
    }

    public URI getContractLocation(QName serviceName)
    {
    ...
    }
}
```

To register the contract resolver with a client you need to add a **bean** element to the client's configuration. The **bean** element's **class** attribute is the name of the class implementing the contract resolver.

[Example 29.6, "Bean Configuring a Contract Resolver"](#) shows a bean for adding a configuration resolver implemented by the **org.apache.cxf.demos.myContractResolver** class.

Example 29.6. Bean Configuring a Contract Resolver

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    ...
    <bean id="myResolver" class="org.apache.cxf.demos.myContractResolver" />
    ...
</beans>
```

Contract resolution order

When a new proxy is created, the runtime uses the contract registry resolver to locate the remote service's WSDL contract. The contract resolver registry calls each contract resolver's **getContractLocation()** method in the order in which the resolvers were registered. It returns the first URI returned from one of the registered contract resolvers.

If you registered a contract resolver that attempted to resolve the WSDL contract at a well known shared file system, it would be the only contract resolver used. However, if you subsequently registered a contract resolver that resolved WSDL locations using a UDDI registry, the registry could use both resolvers to locate a service's WSDL contract. The registry would first attempt to locate the contract using the shared file system contract resolver. If that contract resolver failed, the registry would then attempt to locate it using the UDDI contract resolver.

CHAPTER 30. GENERIC FAULT HANDLING

Abstract

The JAX-WS specification defines two types of faults. One is a generic JAX-WS runtime exception. The other is a protocol specific class of exceptions that is thrown during message processing.

30.1. RUNTIME FAULTS

Overview

Most of the JAX-WS APIs throw a generic javax.xml.ws.WebServiceException exception.

APIs that throw WebServiceException

[Table 30.1, “APIs that Throw WebServiceException”](#) lists some of the JAX-WS APIs that can throw the generic WebServiceException exception.

Table 30.1. APIs that Throw WebServiceException

API	Reason
Binding.setHandlerChain()	There is an error in the handler chain configuration.
BindingProvider.getEndpointReference()	The specified class is not assigned from a W3CEndpointReference .
Dispatch.invoke()	There is an error in the Dispatch instance’s configuration or an error occurred while communicating with the service.
Dispatch.invokeAsync()	There is an error in the Dispatch instance’s configuration.
Dispatch.invokeOneWay()	There is an error in the Dispatch instance’s configuration or an error occurred while communicating with the service.
LogicalMessage.getPayload()	An error occurred when using a supplied JAXBContext to unmarshal the payload. The cause field of the WebServiceException contains the original JAXBException.
LogicalMessage.setPayload()	An error occurred when setting the payload of the message. If the exception is thrown when using a JAXBContext , the cause field of the WebServiceException contains the original JAXBException.

API	Reason
WebServiceContext.getEndpointReference()	The specified class is not assigned from a W3CEndpointReference .

30.2. PROTOCOL FAULTS

Overview

Protocol exceptions are thrown when an error occurs during the processing of a request. All synchronous remote invocations can throw a protocol exception. The underlying cause occurs either in the consumer's message handling chain or in the service provider.

The JAX-WS specification defines a generic protocol exception. It also specifies a SOAP-specific protocol exception and an HTTP-specific protocol exception.

Types of protocol exceptions

The JAX-WS specification defines three types of protocol exception. Which exception you catch depends on the transport and binding used by your application.

[Table 30.2, "Types of Generic Protocol Exceptions"](#) describes the three types of protocol exception and when they are thrown.

Table 30.2. Types of Generic Protocol Exceptions

Exception Class	When Thrown
javax.xml.ws.ProtocolException	This exception is the generic protocol exception. It can be caught regardless of the protocol in use. It can be cast into a specific fault type if you are using the SOAP binding or the HTTP binding. When using the XML binding in combination with the HTTP or JMS transports, the generic protocol exception cannot be cast into a more specific fault type.
javax.xml.ws.soap.SOAPFaultException	This exception is thrown by remote invocations when using the SOAP binding. For more information see the section called "Using the SOAP protocol exception" .
javax.xml.ws.http.HTTPException	This exception is thrown when using the Apache CXF HTTP binding to develop RESTful Web services. For more information see Part VI, "Developing RESTful Web Services" .

Using the SOAP protocol exception

The SOAPFaultException exception wraps a SOAP fault. The underlying SOAP fault is stored in the **fault** field as a javax.xml.soap.SOAPFault object.

If a service implementation needs to throw an exception that does not fit any of the custom exceptions created for the application, it can wrap the fault in a `SOAPFaultException` using the exceptions creator and throw it back to the consumer. [Example 30.1, “Throwing a SOAP Protocol Exception”](#) shows code for creating and throwing a `SOAPFaultException` if the method is passed an invalid parameter.

Example 30.1. Throwing a SOAP Protocol Exception

```
public Quote getQuote(String ticker)
{
    ...
    if(tickers.length()<3)
    {
        SOAPFault fault = SOAPFactory.newInstance().createFault();
        fault.setFaultString("Ticker too short");
        throw new SOAPFaultException(fault);
    }
    ...
}
```

When a consumer catches a `SOAPFaultException` exception they can retrieve the underlying cause of the exception by examining the wrapped `SOAPFault` exception. As shown in [Example 30.2, “Getting the Fault from a SOAP Protocol Exception”](#), the `SOAPFault` exception is retrieved using the `SOAPFaultException` exception’s `getFault()` method.

Example 30.2. Getting the Fault from a SOAP Protocol Exception

```
...
try
{
    proxy.getQuote(ticker);
}
catch (SOAPFaultException sfe)
{
    SOAPFault fault = sfe.getFault();
    ...
}
```

CHAPTER 31. PUBLISHING A SERVICE

Abstract

When you want to deploy a JAX-WS service as a standalone Java application, you must explicitly implement the code that publishes the service provider.

31.1. WHEN TO PUBLISH A SERVICE

Apache CXF provides a number of ways to publish a service as a service provider. How you publish a service depends on the deployment environment you are using. Many of the containers supported by Apache CXF do not require writing logic for publishing endpoints. There are two exceptions:

- deploying a server as a standalone Java application
- deploying a server into an OSGi container without Blueprint

For detailed information in deploying applications into the supported containers see [Part IV, "Configuring Web Service Endpoints"](#).

31.2. APIS USED TO PUBLISH A SERVICE

Overview

The **javax.xml.ws.Endpoint** class does the work of publishing a JAX-WS service provider. To publishing an endpoint do the following:

1. Create an **Endpoint** object for your service provider.
2. Publish the endpoint.
3. Stop the endpoint when application shuts down.

The **Endpoint** class provides methods for creating and publishing service providers. It also provides a method that can create and publish a service provider in a single method call.

Instantiating an service provider

A service provider is instantiated using an **Endpoint** object. You instantiate an **Endpoint** object for your service provider using one of the following methods:

- static**EndpointcreateObjectimplementor** This **create()** method returns an **Endpoint** for the specified service implementation. The **Endpoint** object is created using the information provided by the implementation class' **javax.xml.ws.BindingType** annotation, if it is present. If the annotation is not present, the **Endpoint** uses a default SOAP 1.1/HTTP binding.
- static**EndpointcreateURIBindingIDObjectimplementor** This **create()** method returns an **Endpoint** object for the specified implementation object using the specified binding. This method overrides the binding information provided by the **javax.xml.ws.BindingType** annotation, if it is present. If the **bindingID** cannot be resolved, or it is **null**, the binding specified in the **javax.xml.ws.BindingType** is used to create the **Endpoint**. If neither the **bindingID** or the **javax.xml.ws.BindingType** can be used, the **Endpoint** is created using a default SOAP 1.1/HTTP binding.

- static **Endpoint publish(String address, Object implementor)** The **publish()** method creates an **Endpoint** object for the specified implementation, and publishes it. The binding used for the **Endpoint** object is determined by the URL scheme of the provided **address**. The list of bindings available to the implementation are scanned for a binding that supports the URL scheme. If one is found the **Endpoint** object is created and published. If one is not found, the method fails.

Using **publish()** is the same as invoking one of the **create()** methods, and then invoking the **publish()** method used in [??TITLE???](#).



IMPORTANT

The implementation object passed to any of the **Endpoint** creation methods must either be an instance of a class annotated with **javax.jws.WebService** and meeting the requirements for being an SEI implementation or it must be an instance of a class annotated with **javax.xml.ws.WebServiceProvider** and implementing the Provider interface.

Publishing a service provider

You can publish a service provider using either of the following **Endpoint** methods:

- **publishStringAddress** This **publish()** method publishes the service provider at the address specified.



IMPORTANT

The **address**'s URL scheme must be compatible with one of the service provider's bindings.

- **publishObjectServerContext** This **publish()** method publishes the service provider based on the information provided in the specified server context. The server context must define an address for the endpoint, and the context must also be compatible with one of the service provider's available bindings.

Stopping a published service provider

When the service provider is no longer needed you should stop it using its **stop()** method. The **stop()** method, shown in [Example 31.1, “Method for Stopping a Published Endpoint”](#), shuts down the endpoint and cleans up any resources it is using.

Example 31.1. Method for Stopping a Published Endpoint

```
stop
```



IMPORTANT

Once the endpoint is stopped it cannot be republished.

31.3. PUBLISHING A SERVICE IN A PLAIN JAVA APPLICATION

Overview

When you want to deploy your application as a plain java application you need to implement the logic for publishing your endpoints in the application's **main()** method. Apache CXF provides you two options for writing your application's **main()** method.

- use the **main()** method generated by the **wsdl2java** tool
- write a custom **main()** method that publishes the endpoints

Generating a Server Mainline

The code generators **-server** flag makes the tool generate a simple server mainline. The generated server mainline, as shown in [Example 31.2, "Generated Server Mainline"](#), publishes one service provider for each **port** element in the specified WSDL contract.

For more information see [Section 44.2, "cxfrs-codegen-plugin"](#).

[Example 31.2, "Generated Server Mainline"](#) shows a generated server mainline.

Example 31.2. Generated Server Mainline

```
package org.apache.hello_world_soap_http;

import javax.xml.ws.Endpoint;

public class GreeterServer {

    protected GreeterServer() throws Exception {
        System.out.println("Starting Server");
        Object implementor = new GreeterImpl();
        String address = "http://localhost:9000/SoapContext/SoapPort";
        Endpoint.publish(address, implementor);
    }

    public static void main(String args[]) throws Exception {
        new GreeterServer();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

The code in [Example 31.2, "Generated Server Mainline"](#) does the following:

Instantiates a copy of the service implementation object.

Creates the address for the endpoint based on the contents of the **address** child of the **wsdl:port** element in the endpoint's contract.

Publishes the endpoint.

Writing a Server Mainline

If you used the Java first development model or you do not want to use the generated server mainline you can write your own. To write your server mainline you must do the following:

1. [the section called “Instantiating a service provider”](#) an **javax.xml.ws.Endpoint** object for the service provider.
2. Create an optional server context to use when publishing the service provider.
3. [the section called “Publishing a service provider”](#) the service provider using one of the **publish()** methods.
4. Stop the service provider when the application is ready to exit.

[Example 31.3, “Custom Server Mainline”](#) shows the code for publishing a service provider.

Example 31.3. Custom Server Mainline

```
package org.apache.hello_world_soap_http;

import javax.xml.ws.Endpoint;

public class GreeterServer
{
    protected GreeterServer() throws Exception
    {
    }

    public static void main(String args[]) throws Exception
    {
        GreeterImpl impl = new GreeterImpl();
        Endpoint endpt.create(impl);
        endpt.publish("http://localhost:9000/SoapContext/SoapPort");

        boolean done = false;
        while(!done)
        {
            ...
        }

        endpt.stop();
        System.exit(0);
    }
}
```

The code in [Example 31.3, “Custom Server Mainline”](#) does the following:

Instantiates a copy of the service’s implementation object.

Creates an unpublished **Endpoint** for the service implementation.

Publishes the service provider at <http://localhost:9000/SoapContext/SoapPort>.

Loops until the server should be shutdown.

Stops the published endpoint.

31.4. PUBLISHING A SERVICE IN AN OSGI CONTAINER

Overview

When you develop an application that will be deployed into an OSGi container, you need to coordinate the publishing and stopping of your endpoints with the life-cycle of the bundle in which it is packaged. You want your endpoints published when the bundle is started and you want the endpoints stopped when the bundle is stopped.

You tie your endpoints life-cycle to the bundle's life-cycle by implementing an OSGi bundle activator. A bundle activator is used by the OSGi container to create the resource for a bundle when it is started. The container also uses the bundle activator to clean up the bundles resources when it is stopped.

The bundle activator interface

You create a bundle activator for your application by implementing the org.osgi.framework.BundleActivator interface. The BundleActivator interface, shown in [Example 31.4, “Bundle Activator Interface”](#), it has two methods that need to be implemented.

Example 31.4. Bundle Activator Interface

```
interface BundleActivator
{
    public void start(BundleContext context)
    throws java.lang.Exception;

    public void stop(BundleContext context)
    throws java.lang.Exception;
}
```

The **start()** method is called by the container when it starts the bundle. This is where you instantiate and publish the endpoints.

The **stop()** method is called by the container when it stops the bundle. This is where you would stop the endpoints.

Implementing the start method

The bundle activator's start method is where you publish your endpoints. To publish your endpoints the start method must do the following:

1. [the section called “Instantiating a service provider”](#) an **javax.xml.ws.Endpoint** object for the service provider.
2. Create an optional server context to use when publishing the service provider.
3. [the section called “Publishing a service provider”](#) the service provider using one of the **publish()** methods.

[Example 31.5, “Bundle Activator Start Method for Publishing an Endpoint”](#) shows code for publishing a service provider.

Example 31.5. Bundle Activator Start Method for Publishing an Endpoint

```
package com.widgetvendor.osgi;

import javax.xml.ws.Endpoint;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class widgetActivator implements BundleActivator
{
    private Endpoint endpt;
    ...

    public void start(BundleContext context)
    {
        WidgetOrderImpl impl = new WidgetOrderImpl();
        endpt = Endpoint.create(impl);
        endpt.publish("http://localhost:9000/SapContext/SapPort");
    }

    ...
}

}
```

The code in [Example 31.5, “Bundle Activator Start Method for Publishing an Endpoint”](#) does the following:

Instantiates a copy of the service’s implementation object.

Creates an unpublished **Endpoint** for the service implementation.

Publish the service provider at <http://localhost:9000/SapContext/SapPort>.

Implementing the stop method

The bundle activator’s stop method is where you clean up the resources used by your application. Its implementation should include logic for stopping all of the endpoint’s published by the application.

[Example 31.6, “Bundle Activator Stop Method for Stopping an Endpoint”](#) shows a stop method for stopping a published endpoint.

Example 31.6. Bundle Activator Stop Method for Stopping an Endpoint

```
package com.widgetvendor.osgi;

import javax.xml.ws.Endpoint;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class widgetActivator implements BundleActivator
{
    private Endpoint endpt;
    ...

    public void stop(BundleContext context)
    {
```

```
    endpt.stop();
}
...
}
```

Informing the container

You must add inform the container that the application's bundle includes a bundle activator. You do this by adding the **Bundle-Activator** property to the bundle's manifest. This property tells the container which class in the bundle to use when activating the bundle. Its value is the fully qualified name of the class implementing the bundle activator.

[Example 31.7, "Bundle Activator Manifest Entry"](#) shows a manifest entry for a bundle whose activator is implemented by the class **com.widgetvendor.osgi.widgetActivator**.

Example 31.7. Bundle Activator Manifest Entry

```
Bundle-Activator: com.widgetvendor.osgi.widgetActivator
```

CHAPTER 32. BASIC DATA BINDING CONCEPTS

Abstract

There are a number of general topics that apply to how Apache CXF handles type mapping.

32.1. INCLUDING AND IMPORTING SCHEMA DEFINITIONS

Overview

Apache CXF supports the including and importing of schema definitions, using the **include** and **import** schema tags. These tags enable you to insert definitions from external files or resources into the scope of a schema element. The essential difference between including and importing is:

- Including brings in definitions that belong to the same target namespace as the enclosing schema element.
- Importing brings in definitions that belong to a different target namespace from the enclosing schema element.

xsd:include syntax

The include directive has the following syntax:

```
<include schemaLocation="anyURI" />
```

The referenced schema, given by *anyURI*, must either belong to the same target namespace as the enclosing schema, or not belong to any target namespace at all. If the referenced schema does not belong to any target namespace, it is automatically adopted into the enclosing schema's namespace when it is included.

[Example 32.1, “Example of a Schema that Includes Another Schema”](#) shows an example of an XML Schema document that includes another XML Schema document.

Example 32.1. Example of a Schema that Includes Another Schema

```
<definitions targetNamespace="http://schemas.redhat.com/tests/schema_parser"
    xmlns:tns="http://schemas.redhat.com/tests/schema_parser"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl"/>
<types>
    <schema targetNamespace="http://schemas.redhat.com/tests/schema_parser"
        xmlns="http://www.w3.org/2001/XMLSchema">
        <include schemaLocation="included.xsd"/>
        <complexType name="IncludingSequence">
            <sequence>
                <element name="includedSeq" type="tns:IncludedSequence"/>
            </sequence>
        </complexType>
    </schema>
</types>
...
</definitions>
```

[Example 32.2, "Example of an Included Schema"](#) shows the contents of the included schema file.

Example 32.2. Example of an Included Schema

```
<schema targetNamespace="http://schemas.redhat.com/tests/schema_parser"
       xmlns="http://www.w3.org/2001/XMLSchema">
  <!-- Included type definitions -->
  <complexType name="IncludedSequence">
    <sequence>
      <element name="varInt" type="int"/>
      <element name="varString" type="string"/>
    </sequence>
  </complexType>
</schema>
```

xsd:import syntax

The import directive has the following syntax:

```
<import namespace="namespaceAnyURI"
          schemaLocation="schemaAnyURI" />
```

The imported definitions must belong to the *namespaceAnyURI* target namespace. If *namespaceAnyURI* is blank or remains unspecified, the imported schema definitions are unqualified.

[Example 32.3, "Example of a Schema that Imports Another Schema"](#) shows an example of an XML Schema that imports another XML Schema.

Example 32.3. Example of a Schema that Imports Another Schema

```
<definitions targetNamespace="http://schemas.redhat.com/tests/schema_parser"
             xmlns:tns="http://schemas.redhat.com/tests/schema_parser"
             xmlns:imp="http://schemas.redhat.com/tests/imported_types"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <schema targetNamespace="http://schemas.redhat.com/tests/schema_parser"
           xmlns="http://www.w3.org/2001/XMLSchema">
      <import namespace="http://schemas.redhat.com/tests/imported_types"
              schemaLocation="included.xsd"/>
      <complexType name="IncludingSequence">
        <sequence>
          <element name="includedSeq" type="imp:IncludedSequence"/>
        </sequence>
      </complexType>
    </schema>
  </types>
  ...
</definitions>
```

[Example 32.4, "Example of an Imported Schema"](#) shows the contents of the imported schema file.

Example 32.4. Example of an Imported Schema

```
<schema targetNamespace="http://schemas.redhat.com/tests/imported_types"
       xmlns="http://www.w3.org/2001/XMLSchema">
  <!-- Included type definitions -->
  <complexType name="IncludedSequence">
    <sequence>
      <element name="varInt" type="int"/>
      <element name="varString" type="string"/>
    </sequence>
  </complexType>
</schema>
```

Using non-referenced schema documents

Using types defined in a schema document that is not referenced in the service's WSDL document is a three step process:

1. Convert the schema document to a WSDL document using the **xsd2wsdl** tool.
2. Generate Java for the types using the **wsdl2java** tool on the generated WSDL document.



IMPORTANT

You will get a warning from the **wsdl2java** tool stating that the WSDL document does not define any services. You can ignore this warning.

3. Add the generated classes to your classpath.

32.2. XML NAMESPACE MAPPING

Overview

XML Schema type, group, and element definitions are scoped using namespaces. The namespaces prevent possible naming clashes between entities that use the same name. Java packages serve a similar purpose. Therefore, Apache CXF maps the target namespace of a schema document into a package containing the classes necessary to implement the structures defined in the schema document.

Package naming

The name of the generated package is derived from a schema's target namespace using the following algorithm:

1. The URI scheme, if present, is stripped.



NOTE

Apache CXF will only strip the **http:**, **https:**, and **urn:** schemes.

For example, the namespace `http://www.widgetvendor.com/types/widgetTypes.xsd` becomes `\widgetvendor.com\types\widgetTypes.xsd`.

2. The trailing file type identifier, if present, is stripped.

For example, `\www.widgetvendor.com\types\widgetTypes.xsd` becomes `\widgetvendor.com\types\widgetTypes`.

3. The resulting string is broken into a list of strings using / and : as separators.

So, `\www.widgetvendor.com\types\widgetTypes` becomes the list `{"www.widgetvendor.com", "types", "widgetTypes"}`.

4. If the first string in the list is an internet domain name, it is decomposed as follows:

- a. The leading **www.** is stripped.

- b. The remaining string is split into its component parts using the . as the separator.

- c. The order of the list is reversed.

So, `{"www.widgetvendor.com", "types", "widgetTypes"}` becomes `{"com", "widgetvendor", "types", "widgetTypes"}`



NOTE

Internet domain names end in one of the following: **.com**, **.net**, **.edu**, **.org**, **.gov**, or in one of the two-letter country codes.

5. The strings are converted into all lower case.

So, `{"com", "widgetvendor", "types", "widgetTypes"}` becomes `{"com", "widgetvendor", "types", "widgetTypes"}`.

6. The strings are normalized into valid Java package name components as follows:

- a. If the strings contain any special characters, the special characters are converted to an underscore(_).

- b. If any of the strings are a Java keyword, the keyword is prefixed with an underscore(_).

- c. If any of the strings begin with a numeral, the string is prefixed with an underscore(_).

7. The strings are concatenated using . as a separator.

So, `{"com", "widgetvendor", "types", "widgetTypes"}` becomes the package name `com.widgetvendor.types.widgetTypes`.

The XML Schema constructs defined in the namespace

`http://www.widgetvendor.com/types/widgetTypes.xsd` are mapped to the Java package `com.widgetvendor.types.widgetTypes`.

Package contents

A JAXB generated package contains the following:

- A class implementing each complex type defined in the schema
For more information on complex type mapping see [Chapter 35, Using Complex Types](#).
- An enum type for any simple types defined using the **enumeration** facet
For more information on how enumerations are mapped see [Section 34.3, “Enumerations”](#).

- A public **ObjectFactory** class that contains methods for instantiating objects from the schema
For more information on the **ObjectFactory** class see [Section 32.3, "The Object Factory"](#).
- A **package-info.java** file that provides metadata about the classes in the package

32.3. THE OBJECT FACTORY

Overview

JAXB uses an object factory to provide a mechanism for instantiating instances of JAXB generated constructs. The object factory contains methods for instantiating all of the XML schema defined constructs in the package's scope. The only exception is that enumerations do not get a creation method in the object factory.

Complex type factory methods

For each Java class generated to implement an XML schema complex type, the object factory contains a method for creating an instance of the class. This method takes the form:

```
typeName createtypeName();
```

For example, if your schema contained a complex type named **wIDGETTYPE**, Apache CXF generates a class called **WidgetType** to implement it. [Example 32.5, "Complex Type Object Factory Entry"](#) shows the generated creation method in the object factory.

Example 32.5. Complex Type Object Factory Entry

```
public class ObjectFactory
{
    ...
    WidgetType createWidgetType()
    {
        return new WidgetType();
    }
    ...
}
```

Element factory methods

For elements that are declared in the schema's global scope, Apache CXF inserts a factory method into the object factory. As discussed in [Chapter 33, Using XML Elements](#), XML Schema elements are mapped to **JAXBELEMENT<T>** objects. The creation method takes the form:

```
public JAXBELEMENT<elementType> createElementName(elementType value);
```

For example if you have an element named **comment** of type **xsd:string**, Apache CXF generates the object factory method shown in [Example 32.6, "Element Object Factory Entry"](#)

Example 32.6. Element Object Factory Entry

```
public class ObjectFactory
```

```

    ...
    @XmlElementDecl(namespace = "...", name = "comment")
    public JAXBElement<String> createComment(String value) {
        return new JAXBElement<String>(_Comment_QNAME, String.class, null, value);
    }
    ...
}

```

32.4. ADDING CLASSES TO THE RUNTIME MARSHALLER

Overview

When the Apache CXF runtime reads and writes XML data it uses a map that associates the XML Schema types with their representative Java types. By default, the map contains all of the types defined in the target namespace of the WSDL contract's **schema** element. It also contains any types that are generated from the namespaces of any schemas that are imported into the WSDL contract.

The addition of types from namespaces other than the schema namespace used by an application's **schema** element is accomplished using the **@XmlSeeAlso** annotation. If your application needs to work with types that are generated outside the scope of your application's WSDL document, you can edit the **@XmlSeeAlso** annotation to add them to the JAXB map.

Using the **@XmlSeeAlso** annotation

The **@XmlSeeAlso** annotation can be added to the SEI of your service. It contains a comma separated list of classes to include in the JAXB context. [Example 32.7, "Syntax for Adding Classes to the JAXB Context"](#) shows the syntax for using the **@XmlSeeAlso** annotation.

Example 32.7. Syntax for Adding Classes to the JAXB Context

```

import javax.xml.bind.annotation.XmlSeeAlso;
@WebService()
@XmlSeeAlso({Class1.class, Class2.class, ..., ClassN.class})
public class GeneratedSEI {
    ...
}

```

In cases where you have access to the JAXB generated classes, it is more efficient to use the **ObjectFactory** classes generated to support the needed types. Including the **ObjectFactory** class includes all of the classes that are known to the object factory.

Example

[Example 32.8, "Adding Classes to the JAXB Context"](#) shows an SEI annotated with **@XmlSeeAlso**.

Example 32.8. Adding Classes to the JAXB Context

```

...
import javax.xml.bind.annotation.XmlSeeAlso;

```

```
...
    @WebService()
    @XmlSeeAlso({org.apache.schemas.types.test.ObjectFactory.class,
org.apache.schemas.tests.group_test.ObjectFactory.class})
    public interface Foo {
        ...
    }
```

CHAPTER 33. USING XML ELEMENTS

Abstract

XML Schema elements are used to define an instance of an element in an XML document. Elements are defined either in the global scope of an XML Schema document, or they are defined as a member of a complex type. When they are defined in the global scope, Apache CXF maps them to a JAXB element class that makes manipulating them easier.

OVERVIEW

An element instance in an XML document is defined by an XML Schema **element** element in the global scope of an XML Schema document. To make it easier for Java developers to work with elements, Apache CXF maps globally scoped elements to either a special JAXB element class or to a Java class that is generated to match its content type.

How the element is mapped depends on if the element is defined using a named type referenced by the **type** attribute or if the element is defined using an in-line type definition. Elements defined with in-line type definitions are mapped to Java classes.

It is recommended that elements are defined using a named type because in-line types are not reusable by other elements in the schema.

XML SCHEMA MAPPING

In XML Schema elements are defined using **element** elements. **element** elements has one required attribute. The **name** specifies the name of the element as it appears in an XML document.

In addition to the **name** attribute **element** elements have the optional attributes listed in [Table 33.1, "Attributes Used to Define an Element"](#).

Table 33.1. Attributes Used to Define an Element

Attribute	Description
type	Specifies the type of the element. The type can be any XML Schema primitive type or any named complex type defined in the contract. If this attribute is not specified, you will need to include an in-line type definition.
nillable	Specifies if an element can be left out of a document entirely. If nillable is set to true , the element can be omitted from any document generated using the schema.

Attribute	Description
abstract	Specifies if an element can be used in an instance document. true indicates that the element cannot appear in the instance document. Instead, another element whose substitutionGroup attribute contains the QName of this element must appear in this element's place. For information on how this attribute effects code generation see the section called "Java mapping of abstract elements".
substitutionGroup	Specifies the name of an element that can be substituted with this element. For more information on using type substitution see Chapter 37, Element Substitution.
default	Specifies a default value for an element. For information on how this attribute effects code generation see the section called "Java mapping of elements with a default value".
fixed	Specifies a fixed value for the element.

Example 33.1, “Simple XML Schema Element Definition” shows a simple element definition.

Example 33.1. Simple XML Schema Element Definition

```
<element name="joeFred" type="xsd:string" />
```

An element can also define its own type using an in-line type definition. In-line types are specified using either a **complexType** element or a **simpleType** element. Once you specify whether the type of data is complex or simple, you can define any type of data needed using the tools available for each type of data.

Example 33.2, “XML Schema Element Definition with an In-Line Type” shows an element definition with an in-line type definition.

Example 33.2. XML Schema Element Definition with an In-Line Type

```
<element name="skate">
<complexType>
<sequence>
<element name="numWheels" type="xsd:int" />
<element name="brand" type="xsd:string" />
</sequence>
</complexType>
</element>
```

JAVA MAPPING OF ELEMENTS WITH A NAMED TYPE

By default, globally defined elements are mapped to **JAXBElement<T>** objects where the template class is determined by the value of the **element** element's **type** attribute. For primitive types, the template class is derived using the wrapper class mapping described in [the section called "Wrapper classes"](#). For complex types, the Java class generated to support the complex type is used as the template class.

To support the mapping and to relieve the developer of unnecessary worry about an element's QName, an object factory method is generated for each globally defined element, as shown in [Example 33.3, "Object Factory Method for a Globally Scoped Element"](#).

Example 33.3. Object Factory Method for a Globally Scoped Element

```
public class ObjectFactory {

    private final static QName _name_QNAME = new QName("targetNamespace", "localName");

    ...

    @XmlElementDecl(namespace = "targetNamespace", name = "localName")
    public JAXBElement<type> createname(type value);

}
```

For example, the element defined in [Example 33.1, "Simple XML Schema Element Definition"](#) results in the object factory method shown in [Example 33.4, "Object Factory for a Simple Element"](#).

Example 33.4. Object Factory for a Simple Element

```
public class ObjectFactory {

    private final static QName _JoeFred_QNAME = new QName("...", "joeFred");

    ...

    @XmlElementDecl(namespace = "...", name = "joeFred")
    public JAXBElement<String> createJoeFred(String value);

}
```

[Example 33.5, "Using a Globally Scoped Element"](#) shows an example of using a globally scoped element in Java.

Example 33.5. Using a Globally Scoped Element

```
JAXBElement<String> element = createJoeFred("Green");
String color = element.getValue();
```

USING ELEMENTS WITH NAMED TYPES IN WSDL

If a globally scoped element is used to define a message part, the generated Java parameter is not an instance of **JAXBElement<T>**. Instead it is mapped to a regular Java type or class.

Given the WSDL fragment shown in [Example 33.6, "WSDL Using an Element as a Message Part"](#), the resulting method has a parameter of type **String**.

Example 33.6. WSDL Using an Element as a Message Part

```
<?xml version="1.0" encoding=";UTF-8"?>
<wsdl:definitions name="HelloWorld"
    targetNamespace="http://apache.org/hello_world_soap_http"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://apache.org/hello_world_soap_http"
    xmlns:x1="http://apache.org/hello_world_soap_http/types"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <wsdl:types>
        <schema targetNamespace="http://apache.org/hello_world_soap_http/types"
            xmlns="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"><element name="sayHi">
            <element name="sayHi" type="string"/>
            <element name="sayHiResponse" type="string"/>
        </schema>
    </wsdl:types>

    <wsdl:message name="sayHiRequest">
        <wsdl:part element="x1:sayHi" name="in"/>
    </wsdl:message>
    <wsdl:message name="sayHiResponse">
        <wsdl:part element="x1:sayHiResponse" name="out"/>
    </wsdl:message>

    <wsdl:portType name="Greeter">
        <wsdl:operation name="sayHi">
            <wsdl:input message="tns:sayHiRequest" name="sayHiRequest"/>
            <wsdl:output message="tns:sayHiResponse" name="sayHiResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    ...
</wsdl:definitions>
```

[Example 33.7, "Java Method Using a Global Element as a Part"](#) shows the generated method signature for the **sayHi** operation.

Example 33.7. Java Method Using a Global Element as a Part

String sayHiStringin

JAVA MAPPING OF ELEMENTS WITH AN IN-LINE TYPE

When an element is defined using an in-line type, it is mapped to Java following the same rules used for mapping other types to Java. The rules for simple types are described in [Chapter 34, Using Simple Types](#). The rules for complex types are described in [Chapter 35, Using Complex Types](#).

When a Java class is generated for an element with an in-line type definition, the generated class is decorated with the **@XmlRootElement** annotation. The **@XmlRootElement** annotation has two useful properties: **name** and **namespace**. These attributes are described in [Table 33.2, “Properties for the @XmlRootElement Annotation”](#).

Table 33.2. Properties for the @XmlRootElement Annotation

Property	Description
name	Specifies the value of the XML Schema element element’s name attribute.
namespace	Specifies the namespace in which the element is defined. If this element is defined in the target namespace, the property is not specified.

The **@XmlRootElement** annotation is not used if the element meets one or more of the following conditions:

- The element’s **nillable** attribute is set to **true**
- The element is the head element of a substitution group
For more information on substitution groups see [Chapter 37, Element Substitution](#).

JAVA MAPPING OF ABSTRACT ELEMENTS

When the element’s **abstract** attribute is set to **true** the object factory method for instantiating instances of the type is not generated. If the element is defined using an in-line type, the Java class supporting the in-line type is generated.

JAVA MAPPING OF ELEMENTS WITH A DEFAULT VALUE

When the element’s **default** attribute is used the **defaultValue** property is added to the generated **@XmlElementDecl** annotation. For example, the element defined in [Example 33.8, “XML Schema Element with a Default Value”](#) results in the object factory method shown in [Example 33.9, “Object Factory Method for an Element with a Default Value”](#).

Example 33.8. XML Schema Element with a Default Value

```
<element name="size" type="xsd:int" default="7"/>
```

Example 33.9. Object Factory Method for an Element with a Default Value

```
@XmlElementDecl(namespace = "...", name = "size", defaultValue = "7")
public JAXBElement<Integer> createUnionJoe(Integer value) {
    return new JAXBElement<Integer>(_Size_QNAME, Integer.class, null, value);
}
```



CHAPTER 34. USING SIMPLE TYPES

Abstract

XML Schema simple types are either XML Schema primitive types like **xsd:int**, or are defined using the **simpleType** element. They are used to specify elements that do not contain any children or attributes. They are generally mapped to native Java constructs and do not require the generation of special classes to implement them. Enumerated simple types do not result in generated code because they are mapped to Java **enum** types.

34.1. PRIMITIVE TYPES

Overview

When a message part is defined using one of the XML Schema primitive types, the generated parameter's type is mapped to a corresponding Java native type. The same pattern is used when mapping elements that are defined within the scope of a complex type. The resulting field is of the corresponding Java native type.

Mappings

[Table 34.1, "XML Schema Primitive Type to Java Native Type Mapping"](#) lists the mapping between XML Schema primitive types and Java native types.

Table 34.1. XML Schema Primitive Type to Java Native Type Mapping

XML Schema Type	Java Type
xsd:string	String
xsd:integer	BigInteger
xsd:int	int
xsd:long	long
xsd:short	short
xsd:decimal	BigDecimal
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte
xsd:QName	QName

XML Schema Type	Java Type
xsd:dateTime	XMLGregorianCalendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:time	XMLGregorianCalendar
xsd:date	XMLGregorianCalendar
xsd:g	XMLGregorianCalendar
xsd:anySimpleType ^[a]	Object
xsd:anySimpleType ^[b]	String
xsd:duration	Duration
xsd:NOTATION	QName

[a] For elements of this type.

[b] For attributes of this type.

Wrapper classes

Mapping XML Schema primitive types to Java primitive types does not work for all possible XML Schema constructs. Several cases require that an XML Schema primitive type is mapped to the Java primitive type's corresponding wrapper type. These cases include:

- An **element** element with its **nillable** attribute set to **true** as shown:

```
<element name="finned" type="xsd:boolean"
      nillable="true" />
```

- An **element** element with its **minOccurs** attribute set to **0** and its **maxOccurs** attribute set to **1**, or its **maxOccurs** attribute not specified, as shown :

```
<element name="plane" type="xsd:string" minOccurs="0" />
```

- An **attribute** element with its **use** attribute set to **optional**, or not specified, and having neither its **default** attribute nor its **fixed** attribute specified, as shown:

```
<element name="date">
  <complexType>
    <sequence/>
    <attribute name="calType" type="xsd:string"
      use="optional" />
  </complexType>
</element>
```

Table 34.2, “Primitive Schema Type to Java Wrapper Class Mapping” shows how XML Schema primitive types are mapped into Java wrapper classes in these cases.

Table 34.2. Primitive Schema Type to Java Wrapper Class Mapping

Schema Type	Java Type
xsd:int	java.lang.Integer
xsd:long	java.lang.Long
xsd:short	java.lang.Short
xsd:float	java.lang.Float
xsd:double	java.lang.Double
xsd:boolean	java.lang.Boolean
xsd:byte	java.lang.Byte
xsd:unsignedByte	java.lang.Short
xsd:unsignedShort	java.lang.Integer
xsd:unsignedInt	java.lang.Long
xsd:unsignedLong	java.math.BigInteger
xsd:duration	java.lang.String

34.2. SIMPLE TYPES DEFINED BY RESTRICTION

Overview

XML Schema allows you to create simple types by deriving a new type from another primitive type or simple type. Simple types are described using a **simpleType** element.

The new types are described by restricting the **base type** with one or more facets. These facets limit the possible valid values that can be stored in the new type. For example, you could define a simple type, **SSN**, which is a **string** of exactly 9 characters.

Each of the primitive XML Schema types has their own set of optional facets.

Procedure

To define your own simple type do the following:

1. Determine the base type for your new simple type.
2. Determine what restrictions define the new type based on the available facets for the chosen base type.
3. Using the syntax shown in this section, enter the appropriate **simpleType** element into the types section of your contract.

Defining a simple type in XML Schema

[Example 34.1, "Simple type syntax"](#) shows the syntax for describing a simple type.

Example 34.1. Simple type syntax

```
<simpleType name="typeName">
  <restriction base="baseType">
    <facet value="value" />
    <facet value="value" />
    ...
  </restriction>
</simpleType>
```

The type description is enclosed in a **simpleType** element and identified by the value of the **name** attribute. The base type from which the new simple type is being defined is specified by the **base** attribute of the **xsd:restriction** element. Each facet element is specified within the **restriction** element. The available facets and their valid settings depend on the base type. For example, **xsd:string** has a number of facets including:

- **length**
- **minLength**
- **maxLength**
- **pattern**
- **whiteSpace**

[Example 34.2, "Postal Code Simple Type"](#) shows the definition for a simple type that represents the two-letter postal code used for US states. It can only contain two, uppercase letters. **TX** is a valid value, but **tx** or **tX** are not valid values.

Example 34.2. Postal Code Simple Type

```
<xsd:simpleType name="postalCode">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[A-Z]{2}" />
  </xsd:restriction>
</xsd:simpleType>
```

Mapping to Java

Apache CXF maps user-defined simple types to the Java type of the simple type's base type. So, any message using the simple type **postalCode**, shown in [Example 34.2, "Postal Code Simple Type"](#), is mapped to a **String** because the base type of **postalCode** is **xsd:string**. For example, the WSDL fragment shown in [Example 34.3, "Credit Request with Simple Types"](#) results in a Java method, **state()**, that takes a parameter, **postalCode**, of **String**.

Example 34.3. Credit Request with Simple Types

```
<message name="stateRequest">
  <part name="postalCode" type="postalCode" />
</message>
...
<portType name="postalSupport">
  <operation name="state">
    <input message="tns:stateRequest" name="stateRec" />
    <output message="tns:stateResponse" name="credResp" />
  </operation>
</portType>
```

Enforcing facets

By default, Apache CXF does not enforce any of the facets that are used to restrict a simple type. However, you can configure Apache CXF endpoints to enforce the facets by enabling schema validation.

To configure Apache CXF endpoints to use schema validation set the **schema-validation-enabled** property to **true**. [Example 34.4, "Service Provider Configured to Use Schema Validation"](#) shows the configuration for a service provider that uses schema validation

Example 34.4. Service Provider Configured to Use Schema Validation

```
<jaxws:endpoint name="{http://apache.org/hello_world_soap_http}SoapPort"
  wsdlLocation="wsdl/hello_world.wsdl"
  createdFromAPI="true">
  <jaxws:properties>
    <entry key="schema-validation-enabled" value="BOTH" />
  </jaxws:properties>
</jaxws:endpoint>
```

For more information on configuring schema validation, see [Section 24.3.4.7, "Schema Validation Type Values"](#).

34.3. ENUMERATIONS

Overview

In XML Schema, enumerated types are simple types that are defined using the **xsd:enumeration** facet. Unlike atomic simple types, they are mapped to Java **enums**.

Defining an enumerated type in XML Schema

Enumerations are a simple type using the **xsd:enumeration** facet. Each **xsd:enumeration** facet defines one possible value for the enumerated type.

[Example 34.5, “XML Schema Defined Enumeration”](#) shows the definition for an enumerated type. It has the following possible values:

- **big**
- **large**
- **mungo**
- **gargantuan**

Example 34.5. XML Schema Defined Enumeration

```
<simpleType name="widgetSize">
  <restriction base="xsd:string">
    <enumeration value="big"/>
    <enumeration value="large"/>
    <enumeration value="mungo"/>
    <enumeration value="gargantuan"/>
  </restriction>
```

Mapping to Java

XML Schema enumerations where the base type is **xsd:string** are automatically mapped to Java **enum** type. You can instruct the code generator to map enumerations with other base types to Java **enum** types by using the customizations described in [Section 38.4, “Customizing Enumeration Mapping”](#).

The **enum** type is created as follows:

1. The name of the type is taken from the **name** attribute of the simple type definition and converted to a Java identifier.
In general, this means converting the first character of the XML Schema’s name to an uppercase letter. If the first character of the XML Schema’s name is an invalid character, an underscore (_) is prepended to the name.
2. For each **enumeration** facet, an enum constant is generated based on the value of the **value** attribute.
The constant’s name is derived by converting all of the lowercase letters in the value to their uppercase equivalent.
3. A constructor is generated that takes the Java type mapped from the enumeration’s base type.

4. A public method called **value()** is generated to access the facet value that is represented by an instance of the type.
The return type of the **value()** method is the base type of the XML Schema type.
5. A public method called **fromValue()** is generated to create an instance of the enum type based on a facet value.
The parameter type of the **value()** method is the base type of the XML Schema type.
6. The class is decorated with the **@XmlEnum** annotation.

The enumerated type defined in [Example 34.5, “XML Schema Defined Enumeration”](#) is mapped to the **enum** type shown in [Example 34.6, “Generated Enumerated Type for a String Bases XML Schema Enumeration”](#).

Example 34.6. Generated Enumerated Type for a String Bases XML Schema Enumeration

```
@XmlType(name = "widgetSize")
@XmlEnum
public enum WidgetSize {

    @XmlEnumValue("big")
    BIG("big"),
    @XmlEnumValue("large")
    LARGE("large"),
    @XmlEnumValue("mungo")
    MUNGO("mungo"),
    @XmlEnumValue("gargantuan")
    GARGANTUAN("gargantuan");
    private final String value;

    WidgetSize(String v) {
        value = v;
    }

    public String value() {
        return value;
    }

    public static WidgetSize fromValue(String v) {
        for (WidgetSize c: WidgetSize.values()) {
            if (c.value.equals(v)) {
                return c;
            }
        }
        throw new IllegalArgumentException(v);
    }

}
```

34.4. LISTS

Overview

XML Schema supports a mechanism for defining data types that are a list of space separated simple types. An example of an element, **primeList**, using a list type is shown in [Example 34.7, "List Type Example"](#).

Example 34.7. List Type Example

```
<primeList>1 3 5 7 9 11 13</primeList>
```

XML Schema list types are generally mapped to Java **List<T>** objects. The only variation to this pattern is when a message part is mapped directly to an instance of an XML Schema list type.

Defining list types in XML Schema

XML Schema list types are simple types and as such are defined using a **simpleType** element. The most common syntax used to define a list type is shown in [Example 34.8, "Syntax for XML Schema List Types"](#).

Example 34.8. Syntax for XML Schema List Types

```
<simpleType name="listType">
  <list itemType="atomicType">
    <facet value="value" />
    <facet value="value" />
    ...
  </list>
</simpleType>
```

The value given for *atomicType* defines the type of the elements in the list. It can only be one of the built-in XML Schema atomic types, like **xsd:int** or **xsd:string**, or a user-defined simple type that is not a list.

In addition to defining the type of elements listed in the list type, you can also use facets to further constrain the properties of the list type. [Table 34.3, "List Type Facets"](#) shows the facets used by list types.

Table 34.3. List Type Facets

Facet	Effect
length	Defines the number of elements in an instance of the list type.
minLength	Defines the minimum number of elements allowed in an instance of the list type.
maxLength	Defines the maximum number of elements allowed in an instance of the list type.
enumeration	Defines the allowable values for elements in an instance of the list type.

Facet	Effect
pattern	Defines the lexical form of the elements in an instance of the list type. Patterns are defined using regular expressions.

For example, the definition for the **simpleList** element shown in [Example 34.7, “List Type Example”](#), is shown in [Example 34.9, “Definition of a List Type”](#).

Example 34.9. Definition of a List Type

```
<simpleType name="primeListType">
  <list itemType="int"/>
</simpleType>
<element name="primeList" type="primeListType"/>
```

In addition to the syntax shown in [Example 34.8, “Syntax for XML Schema List Types”](#) you can also define a list type using the less common syntax shown in [Example 34.10, “Alternate Syntax for List Types”](#).

Example 34.10. Alternate Syntax for List Types

```
<simpleType name="listType">
  <list>
    <simpleType>
      <restriction base="atomicType">
        <facet value="value"/>
        <facet value="value"/>
        ...
      </restriction>
    </simpleType>
  </list>
</simpleType>
```

Mapping list type elements to Java

When an element is defined a list type, the list type is mapped to a collection property. A collection property is a Java **List<T>** object. The template class used by the **List<T>** is the wrapper class mapped from the list’s base type. For example, the list type defined in [Example 34.9, “Definition of a List Type”](#) is mapped to a **List<Integer>**.

For more information on wrapper type mapping see [the section called “Wrapper classes”](#).

Mapping list type parameters to Java

When a message part is defined as a list type, or is mapped to an element of a list type, the resulting method parameter is mapped to an array instead of a **List<T>** object. The base type of the array is the wrapper class of the list type’s base class.

For example, the WSDL fragment in [Example 34.11, “WSDL with a List Type Message Part”](#) results in the method signature shown in [Example 34.12, “Java Method with a List Type Parameter”](#).

Example 34.11. WSDL with a List Type Message Part

```
<definitions ...>
...
<types ...>
  <schema ... >
    <simpleType name="primeListType">
      <list itemType="int"/>
    </simpleType>
    <element name="primeList" type="primeListType"/>
  </schemas>
</types>
<message name="numRequest"> <part name="inputData" element="xsd1:primeList" />
</message>
<message name="numResponse">;
  <part name="outputData" type="xsd:int">
  ...
<portType name="numberService">
  <operation name="primeProcessor">
    <input name="numRequest" message="tns:numRequest" />
    <output name="numResponse" message="tns:numResponse" />
  </operation>
  ...
</portType>
...
</definitions>
```

Example 34.12. Java Method with a List Type Parameter

```
public interface NumberService {
  ...
  @XmlList
  @WebResult(name = "outputData", targetNamespace = "", partName = "outputData")
  @WebMethod
  public int primeProcessor(
    @WebParam(partName = "inputData", name = "primeList", targetNamespace = "... ")
    java.lang.Integer[] inputData
  );
}
```

34.5. UNIONS

Overview

In XML Schema, a union is a construct that allows you to describe a type whose data can be one of a number of simple types. For example, you can define a type whose value is either the integer **1** or the string **first**. Unions are mapped to Java **Strings**.

Defining in XML Schema

XML Schema unions are defined using a **simpleType** element. They contain at least one **union** element that defines the member types of the union. The member types of the union are the valid types of data that can be stored in an instance of the union. They are defined using the **union** element's **memberTypes** attribute. The value of the **memberTypes** attribute contains a list of one or more defined simple type names. [Example 34.13, "Simple Union Type"](#) shows the definition of a union that can store either an integer or a string.

Example 34.13. Simple Union Type

```
<simpleType name="orderNumUnion">
  <union memberTypes="xsd:string xsd:int" />
</simpleType>
```

In addition to specifying named types as a member type of a union, you can also define an anonymous simple type as a member type of a union. This is done by adding the anonymous type definition inside of the **union** element. [Example 34.14, "Union with an Anonymous Member Type"](#) shows an example of a union containing an anonymous member type that restricts the possible values of a valid integer to the range 1 through 10.

Example 34.14. Union with an Anonymous Member Type

```
<simpleType name="restrictedOrderNumUnion">
  <union memberTypes="xsd:string">
    <simpleType>
      <restriction base="xsd:int">
        <minInclusive value="1" />
        <maxInclusive value="10" />
      </restriction>
    </simpleType>
  </union>
</simpleType>
```

Mapping to Java

XML Schema union types are mapped to Java **String** objects. By default, Apache CXF does not validate the contents of the generated object. To have Apache CXF validate the contents you will must configure the runtime to use schema validation as described in [the section called "Enforcing facets"](#).

34.6. SIMPLE TYPE SUBSTITUTION

Overview

XML allows for simple type substitution between compatible types using the **xsi:type** attribute. The default mapping of simple types to Java primitive types, however, does not fully support simple type substitution. The runtime can handle basic simple type substitution, but information is lost. The code generators can be customized to generate Java classes that facilitate lossless simple type substitution.

Default mapping and marshaling

Because Java primitive types do not support type substitution, the default mapping of simple types to Java primitive types presents problems for supporting simple type substitution. The Java virtual machine will balk if an attempt is made to pass a **short** into a variable that expects an **int** even though the schema defining the types allows it.

To get around the limitations imposed by the Java type system, Apache CXF allows for simple type substitution when the value of the element's **xsi:type** attribute meets one of the following conditions:

- It specifies a primitive type that is compatible with the element's schema type.
- It specifies a type that derives by restriction from the element's schema type.
- It specifies a complex type that derives by extension from the element's schema type.

When the runtime does the type substitution it does not retain any knowledge of the type specified in the element's **xsi:type** attribute. If the type substitution is from a complex type to a simple type, only the value directly related to the simple type is preserved. Any other elements and attributes added by extension are lost.

Supporting lossless type substitution

You can customize the generation of simple types to facilitate lossless support of simple type substitution in the following ways:

- Set the **globalBindings** customization element's **mapSimpleTypeDef** to **true**.
This instructs the code generator to create Java value classes for all named simple types defined in the global scope.

For more information see [Section 38.3, "Generating Java Classes for Simple Types"](#).
- Add a **javaType** element to the **globalBindings** customization element.
This instructs the code generators to map all instances of an XML Schema primitive type to a specific class of object.

For more information see [Section 38.2, "Specifying the Java Class of an XML Schema Primitive"](#).
- Add a **baseType** customization element to the specific elements you want to customize.
The **baseType** customization element allows you to specify the Java type generated to represent a property. To ensure the best compatibility for simple type substitution, use **java.lang.Object** as the base type.

For more information see [Section 38.6, "Specifying the Base Type of an Element or an Attribute"](#).

CHAPTER 35. USING COMPLEX TYPES

Abstract

Complex types can contain multiple elements and they can have attributes. They are mapped into Java classes that can hold the data represented by the type definition. Typically, the mapping is to a bean with a set of properties representing the elements and the attributes of the content model..

35.1. BASIC COMPLEX TYPE MAPPING

Overview

XML Schema complex types define constructs containing more complex information than a simple type. The most simple complex types define an empty element with an attribute. More intricate complex types are made up of a collection of elements.

By default, an XML Schema complex type is mapped to a Java class, with a member variable to represent each element and attribute listed in the XML Schema definition. The class has setters and getters for each member variable.

Defining in XML Schema

XML Schema complex types are defined using the **complexType** element. The **complexType** element wraps the rest of elements used to define the structure of the data. It can appear either as the parent element of a named type definition, or as the child of an **element** element anonymously defining the structure of the information stored in the element. When the **complexType** element is used to define a named type, it requires the use of the **name** attribute. The **name** attribute specifies a unique identifier for referencing the type.

Complex type definitions that contain one or more elements have one of the child elements described in [Table 35.1, “Elements for Defining How Elements Appear in a Complex Type”](#). These elements determine how the specified elements appear in an instance of the type.

Table 35.1. Elements for Defining How Elements Appear in a Complex Type

Element	Description
all	All of the elements defined as part of the complex type must appear in an instance of the type. However, they can appear in any order.
choice	Only one of the elements defined as part of the complex type can appear in an instance of the type.
sequence	All of the elements defined as part of the complex type must appear in an instance of the type, and they must also appear in the order specified in the type definition.



NOTE

If a complex type definition only uses attributes, you do not need one of the elements described in [Table 35.1, “Elements for Defining How Elements Appear in a Complex Type”](#).

After deciding how the elements will appear, you define the elements by adding one or more **element** element children to the definition.

[Example 35.1, “XML Schema Complex Type”](#) shows a complex type definition in XML Schema.

Example 35.1. XML Schema Complex Type

```
<complexType name="sequence">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="street" type="xsd:short" />
    <element name="city" type="xsd:string" />
    <element name="state" type="xsd:string" />
    <element name="zipCode" type="xsd:string" />
  </sequence>
</complexType>
```

Mapping to Java

XML Schema complex types are mapped to Java classes. Each element in the complex type definition is mapped to a member variable in the Java class. Getter and setter methods are also generated for each element in the complex type.

All generated Java classes are decorated with the **@XmlType** annotation. If the mapping is for a named complex type, the annotations **name** is set to the value of the **complexType** element’s **name** attribute. If the complex type is defined as part of an element definition, the value of the **@XmlType** annotation’s **name** property is the value of the **element** element’s **name** attribute.



NOTE

As described in [the section called “Java mapping of elements with an in-line type”](#), the generated class is decorated with the **@XmlRootElement** annotation if it is generated for a complex type defined as part of an element definition.

To provide the runtime with guidelines indicating how the elements of the XML Schema complex type should be handled, the code generators alter the annotations used to decorate the class and its member variables.

All Complex Type

All complex types are defined using the **all** element. They are annotated as follows:

- The **@XmlType** annotation’s **propOrder** property is empty.
- Each element is decorated with the **@XmlElement** annotation.
- The **@XmlElement** annotation’s **required** property is set to **true**.

[Example 35.2, "Mapping of an All Complex Type"](#) shows the mapping for an all complex type with two elements.

Example 35.2. Mapping of an All Complex Type

```
@XmlType(name = "all", propOrder = {  
})  
public class All {  
    @XmlElement(required = true)  
    protected BigDecimal amount;  
    @XmlElement(required = true)  
    protected String type;  
  
    public BigDecimal getAmount() {  
        return amount;  
    }  
  
    public void setAmount(BigDecimal value) {  
        this.amount = value;  
    }  
  
    public String getType() {  
        return type;  
    }  
  
    public void setType(String value) {  
        this.type = value;  
    }  
}
```

Choice Complex Type

Choice complex types are defined using the **choice** element. They are annotated as follows:

- The **@XmlType** annotation's **propOrder** property lists the names of the elements in the order they appear in the XML Schema definition.
 - None of the member variables are annotated.
- [Example 35.3, "Mapping of a Choice Complex Type"](#) shows the mapping for a choice complex type with two elements.

Example 35.3. Mapping of a Choice Complex Type

```
@XmlType(name = "choice", propOrder = {  
    "address",  
    "floater"  
})  
public class Choice {  
  
    protected Sequence address;  
    protected Float floater;  
  
    public Sequence getAddress() {
```

```

        return address;
    }

    public void setAddress(Sequence value) {
        this.address = value;
    }

    public Float getFloater() {
        return floater;
    }

    public void setFloater(Float value) {
        this.floater = value;
    }

}

```

Sequence Complex Type

A sequence complex type is defined using the **sequence** element. It is annotated as follows:

- The **@XmlType** annotation's **propOrder** property lists the names of the elements in the order they appear in the XML Schema definition.
- Each element is decorated with the **@XmlElement** annotation.
- The **@XmlElement** annotation's **required** property is set to **true**.

[Example 35.4, "Mapping of a Sequence Complex Type"](#) shows the mapping for the complex type defined in [Example 35.1, "XML Schema Complex Type"](#).

Example 35.4. Mapping of a Sequence Complex Type

```

@XmlType(name = "sequence", propOrder = {
    "name",
    "street",
    "city",
    "state",
    "zipCode"
})
public class Sequence {

    @XmlElement(required = true)
    protected String name;
    protected short street;
    @XmlElement(required = true)
    protected String city;
    @XmlElement(required = true)
    protected String state;
    @XmlElement(required = true)
    protected String zipCode;

    public String getName() {
        return name;
    }
}

```

```
public void setName(String value) {
    this.name = value;
}

public short getStreet() {
    return street;
}

public void setStreet(short value) {
    this.street = value;
}

public String getCity() {
    return city;
}

public void setCity(String value) {
    this.city = value;
}

public String getState() {
    return state;
}

public void setState(String value) {
    this.state = value;
}

public String getZipCode() {
    return zipCode;
}

public void setZipCode(String value) {
    this.zipCode = value;
}
}
```

35.2. ATTRIBUTES

Overview

Apache CXF supports the use of **attribute** elements and **attributeGroup** elements within the scope of a **complexType** element. When defining structures for an XML document attribute declarations provide a means of adding information that is specified within the tag, not the value that the tag contains. For example, when describing the XML element <value currency="euro">410</value> in XML Schema the **currency** attribute is described using an **attribute** element as shown in [Example 35.5, “XML Schema Defining and Attribute”](#).

The **attributeGroup** element allows you to define a group of reusable attributes that can be referenced by all complex types defined by the schema. For example, if you are defining a series of elements that all use the attributes **category** and **pubDate**, you could define an attribute group with these attributes and reference them in all the elements that use them. This is shown in [Example 35.7, “Attribute Group Definition”](#).

When describing data types for use in developing application logic, attributes whose **use** attribute is set to either **optional** or **required** are treated as elements of a structure. For each attribute declaration contained within a complex type description, an element is generated in the class for the attribute, along with the appropriate getter and setter methods.

Defining an attribute in XML Schema

An XML Schema **attribute** element has one required attribute, **name**, that is used to identify the attribute. It also has four optional attributes that are described in [Table 35.2, “Optional Attributes Used to Define Attributes in XML Schema”](#).

Table 35.2. Optional Attributes Used to Define Attributes in XML Schema

Attribute	Description
use	Specifies if the attribute is required. Valid values are required , optional , or prohibited . optional is the default value.
type	Specifies the type of value the attribute can take. If it is not used the schema type of the attribute must be defined in-line.
default	Specifies a default value to use for the attribute. It is only used when the attribute element's use attribute is set to optional .
fixed	Specifies a fixed value to use for the attribute. It is only used when the attribute element's use attribute is set to optional .

[Example 35.5, “XML Schema Defining and Attribute”](#) shows an attribute element defining an attribute, currency, whose value is a string.

Example 35.5. XML Schema Defining and Attribute

```
<element name="value">
  <complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:integer">
        <xsd:attribute name="currency" type="xsd:string"
          use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

If the **type** attribute is omitted from the **attribute** element, the format of the data must be described in-line. [Example 35.6, “Attribute with an In-Line Data Description”](#) shows an **attribute** element for an attribute, **category**, that can take the values **autobiography**, **non-fiction**, or **fiction**.

Example 35.6. Attribute with an In-Line Data Description

```
<attribute name="category" use="required">
  <simpleType>
    <restriction base="xsd:string">
      <enumeration value="autobiography"/>
      <enumeration value="non-fiction"/>
      <enumeration value="fiction"/>
    </restriction>
  </simpleType>
</attribute>
```

Using an attribute group in XML Schema

Using an attribute group in a complex type definition is a two step process:

1. Define the attribute group.

An attribute group is defined using an **attributeGroup** element with a number of **attribute** child elements. The **attributeGroup** requires a **name** attribute that defines the string used to refer to the attribute group. The **attribute** elements define the members of the attribute group and are specified as shown in [the section called “Defining an attribute in XML Schema”](#). [Example 35.7, “Attribute Group Definition”](#) shows the description of the attribute group **catalogIndices**. The attribute group has two members: **category**, which is optional, and **pubDate**, which is required.

Example 35.7. Attribute Group Definition

```
<attributeGroup name="catalogIndices">
  <attribute name="category" type="catagoryType" />
  <attribute name="pubDate" type="dateTime"
    use="required" />
</attributeGroup>
```

2. Use the attribute group in the definition of a complex type.

You use attribute groups in complex type definitions by using the **attributeGroup** element with the **ref** attribute. The value of the **ref** attribute is the name given the attribute group that you want to use as part of the type definition. For example if you want to use the attribute group **catalogIndices** in the complex type **dvdType**, you would use `<attributeGroup ref="catalogIndices" />` as shown in [Example 35.8, “Complex Type with an Attribute Group”](#).

Example 35.8. Complex Type with an Attribute Group

```
<complexType name="dvdType">
  <sequence>
    <element name="title" type="xsd:string" />
    <element name="director" type="xsd:string" />
    <element name="numCopies" type="xsd:int" />
  </sequence>
  <attributeGroup ref="catalogIndices" />
</complexType>
```

Mapping attributes to Java

Attributes are mapped to Java in much the same way that member elements are mapped to Java. Required attributes and optional attributes are mapped to member variables in the generated Java class. The member variables are decorated with the **@XmlAttribute** annotation. If the attribute is required, the **@XmlAttribute** annotation's **required** property is set to **true**.

The complex type defined in [Example 35.9, “techDoc Description”](#) is mapped to the Java class shown in [Example 35.10, “techDoc Java Class”](#).

Example 35.9. techDoc Description

```
<complexType name="techDoc">
  <all>
    <element name="product" type="xsd:string" />
    <element name="version" type="xsd:short" />
  </all>
  <attribute name="usefullness" type="xsd:float"
    use="optional" default="0.01" />
</complexType>
```

Example 35.10. techDoc Java Class

```
@XmlType(name = "techDoc", propOrder = {
})
public class TechDoc {

  @XmlElement(required = true)
  protected String product;
  protected short version;
  @XmlAttribute protected Float usefullness;

  public String getProduct() {
    return product;
  }

  public void setProduct(String value) {
    this.product = value;
  }

  public short getVersion() {
    return version;
  }

  public void setVersion(short value) {
    this.version = value;
  }

  public float getUsefullness() { if (usefullness == null) { return 0.01F; } else { return usefullness; } }

  public void setUsefullness(Float value) {
```

```

        this.usefulness = value;
    }
}

```

As shown in [Example 35.10, “techDoc Java Class”](#), the **default** attribute and the **fixed** attribute instruct the code generators to add code to the getter method generated for the attribute. This additional code ensures that the specified value is returned if no value is set.



IMPORTANT

The **fixed** attribute is treated the same as the **default** attribute. If you want the **fixed** attribute to be treated as a Java constant you can use the customization described in [Section 38.5, “Customizing Fixed Value Attribute Mapping”](#).

Mapping attribute groups to Java

Attribute groups are mapped to Java as if the members of the group were explicitly used in the type definition. If the attribute group has three members, and it is used in a complex type, the generated class for that type will include a member variable, along with the getter and setter methods, for each member of the attribute group. For example, the complex type defined in [Example 35.8, “Complex Type with an Attribute Group”](#), Apache CXF generates a class containing the member variables **category** and **pubDate** to support the members of the attribute group as shown in [Example 35.11, “dvdType Java Class”](#).

Example 35.11. dvdType Java Class

```

@XmlType(name = "dvdType", propOrder = {
    "title",
    "director",
    "numCopies"
})
public class DvdType {

    @XmlElement(required = true)
    protected String title;
    @XmlElement(required = true)
    protected String director;
    protected int numCopies;
    @XmlAttribute protected CatagoryType category; @XmlAttribute(required = true)
    @XmlSchemaType(name = "dateTime") protected XMLGregorianCalendar pubDate;

    public String getTitle() {
        return title;
    }

    public void setTitle(String value) {
        this.title = value;
    }

    public String getDirector() {
        return director;
    }
}

```

```

public void setDirector(String value) {
    this.director = value;
}

public int getNumCopies() {
    return numCopies;
}

public void setNumCopies(int value) {
    this.numCopies = value;
}

public CatagoryType getCatagory() {
    return catagory;
}

public void setCatagory(CatagoryType value) {
    this.catagory = value;
}

public XMLGregorianCalendar getPubDate() {
    return pubDate;
}

public void setPubDate(XMLGregorianCalendar value) {
    this.pubDate = value;
}

}

```

35.3. DERIVING COMPLEX TYPES FROM SIMPLE TYPES

Overview

Apache CXF supports derivation of a complex type from a simple type. A simple type has, by definition, neither sub-elements nor attributes. Hence, one of the main reasons for deriving a complex type from a simple type is to add attributes to the simple type.

There are two ways of deriving a complex type from a simple type:

- [By extension](#)
- [By restriction](#)

Derivation by extension

[Example 35.12, “Deriving a Complex Type from a Simple Type by Extension”](#) shows an example of a complex type, **InternationalPrice**, derived by extension from the **xsd:decimal** primitive type to include a currency attribute.

Example 35.12. Deriving a Complex Type from a Simple Type by Extension

```
<complexType name="internationalPrice">
```

```

<simpleContent>
    <extension base="xsd:decimal">
        <attribute name="currency" type="xsd:string"/>
    </extension>
</simpleContent>
</complexType>

```

The **simpleContent** element indicates that the new type does not contain any sub-elements. The **extension** element specifies that the new type extends **xsd:decimal**.

Derivation by restriction

[Example 35.13, “Deriving a Complex Type from a Simple Type by Restriction”](#) shows an example of a complex type, **idType**, that is derived by restriction from **xsd:string**. The defined type restricts the possible values of **xsd:string** to values that are ten characters in length. It also adds an attribute to the type.

Example 35.13. Deriving a Complex Type from a Simple Type by Restriction

```

<complexType name="idType">
    <simpleContent>
        <restriction base="xsd:string">
            <length value="10" />
            <attribute name="expires" type="xsd:dateTime" />
        </restriction>
    </simpleContent>
</complexType>

```

As in [Example 35.12, “Deriving a Complex Type from a Simple Type by Extension”](#) the **simpleContent** element signals that the new type does not contain any children. This example uses a **restriction** element to constrain the possible values used in the new type. The **attribute** element adds the element to the new type.

Mapping to Java

A complex type derived from a simple type is mapped to a Java class that is decorated with the **@XmlType** annotation. The generated class contains a member variable, **value**, of the simple type from which the complex type is derived. The member variable is decorated with the **@XmlValue** annotation. The class also has a **getValue()** method and a **setValue()** method. In addition, the generated class has a member variable, and the associated getter and setter methods, for each attribute that extends the simple type.

[Example 35.14, “idType Java Class”](#) shows the Java class generated for the **idType** type defined in [Example 35.13, “Deriving a Complex Type from a Simple Type by Restriction”](#).

Example 35.14. idType Java Class

```

@XmlType(name = "idType", propOrder = {
    "value"
})
public class IdType {

```

```

@XmlValue
protected String value;
@XmlAttribute
@XmlSchemaType(name = "dateTime")
protected XMLGregorianCalendar expires;

public String getValue() {
    return value;
}

public void setValue(String value) {
    this.value = value;
}

public XMLGregorianCalendar getExpires() {
    return expires;
}

public void setExpires(XMLGregorianCalendar value) {
    this.expires = value;
}

}

```

35.4. DERIVING COMPLEX TYPES FROM COMPLEX TYPES

Overview

Using XML Schema, you can derive new complex types by either extending or restricting other complex types using the **complexContent** element. When generating the Java class to represent the derived complex type, Apache CXF extends the base type's class. In this way, the generated Java code preserves the inheritance hierarchy intended in the XML Schema.

Schema syntax

You derive complex types from other complex types by using the **complexContent** element, and either the **extension** element or the **restriction** element. The **complexContent** element specifies that the included data description includes more than one field. The **extension** element and the **restriction** element, which are children of the **complexContent** element, specify the base type being modified to create the new type. The base type is specified by the **base** attribute.

Extending a complex type

To extend a complex type use the **extension** element to define the additional elements and attributes that make up the new type. All elements that are allowed in a complex type description are allowable as part of the new type's definition. For example, you can add an anonymous enumeration to the new type, or you can use the **choice** element to specify that only one of the new fields can be valid at a time.

[Example 35.15, “Deriving a Complex Type by Extension”](#) shows an XML Schema fragment that defines two complex types, **widgetOrderInfo** and **widgetOrderBillInfo**. **widgetOrderBillInfo** is derived by extending **widgetOrderInfo** to include two new elements: **orderNumber** and **amtDue**.

Example 35.15. Deriving a Complex Type by Extension

```
<complexType name="widgetOrderInfo">
  <sequence>
    <element name="amount" type="xsd:int"/>
    <element name="order_date" type="xsd:dateTime"/>
    <element name="type" type="xsd1:widgetSize"/>
    <element name="shippingAddress" type="xsd1:Address"/>
  </sequence>
  <attribute name="rush" type="xsd:boolean" use="optional" />
</complexType>
<complexType name="widgetOrderBillInfo">
  <complexContent>
    <extension base="xsd1:widgetOrderInfo">
      <sequence>
        <element name="amtDue" type="xsd:decimal"/>
        <element name="orderNumber" type="xsd:string"/>
      </sequence>
      <attribute name="paid" type="xsd:boolean"
                default="false" />
    </extension>
  </complexContent>
</complexType>
```

Restricting a complex type

To restrict a complex type use the **restriction** element to limit the possible values of the base type's elements or attributes. When restricting a complex type you must list all of the elements and attributes of the base type. For each element you can add restrictive attributes to the definition. For example, you can add a **maxOccurs** attribute to an element to limit the number of times it can occur. You can also use the **fixed** attribute to force one or more of the elements to have predetermined values.

[Example 35.16, “Defining a Complex Type by Restriction”](#) shows an example of defining a complex type by restricting another complex type. The restricted type, **wallaWallaAddress**, can only be used for addresses in Walla Walla, Washington because the values for the **city** element, the **state** element, and the **zipCode** element are fixed.

Example 35.16. Defining a Complex Type by Restriction

```
<complexType name="Address">
  <sequence>
    <element name="name" type="xsd:string"/>
    <element name="street" type="xsd:short" maxOccurs="3"/>
    <element name="city" type="xsd:string"/>
    <element name="state" type="xsd:string"/>
    <element name="zipCode" type="xsd:string"/>
  </sequence>
</complexType>
<complexType name="wallawallaAddress">
  <complexContent>
    <restriction base="xsd1:Address">
      <sequence>
        <element name="name" type="xsd:string"/>
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

```

<element name="street" type="xsd:short"
    maxOccurs="3"/>
<element name="city" type="xsd:string"
    fixed="WallaWalla"/>
<element name="state" type="xsd:string"
    fixed="WA" />
<element name="zipCode" type="xsd:string"
    fixed="99362" />
</sequence>
</restriction>
</complexContent>
</complexType>
```

Mapping to Java

As it does with all complex types, Apache CXF generates a class to represent complex types derived from another complex type. The Java class generated for the derived complex type extends the Java class generated to support the base complex type. The base Java class is also modified to include the **@XmlSeeAlso** annotation. The base class' **@XmlSeeAlso** annotation lists all of the classes that extend the base class.

When the new complex type is derived by extension, the generated class will include member variables for all of the added elements and attributes. The new member variables will be generated according to the same mappings as all other elements.

When the new complex type is derived by restriction, the generated class will have no new member variables. The generated class will simply be a shell that does not provide any additional functionality. It is entirely up to you to ensure that the restrictions defined in the XML Schema are enforced.

For example, the schema in [Example 35.15, “Deriving a Complex Type by Extension”](#) results in the generation of two Java classes: **WidgetOrderInfo** and **WidgetBillOrderInfo**. **WidgetOrderBillInfo** extends **WidgetOrderInfo** because **widgetOrderBillInfo** is derived by extension from **widgetOrderInfo**. [Example 35.17, “WidgetOrderBillInfo”](#) shows the generated class for **widgetOrderBillInfo**.

Example 35.17. WidgetOrderBillInfo

```

@XmlType(name = "widgetOrderBillInfo", propOrder = {
    "amtDue",
    "orderNumber"
})
public class WidgetOrderBillInfo
    extends WidgetOrderInfo
{
    @XmlElement(required = true)
    protected BigDecimal amtDue;
    @XmlElement(required = true)
    protected String orderNumber;
    @XmlAttribute
    protected Boolean paid;

    public BigDecimal getAmtDue() {
        return amtDue;
    }
}
```

```

public void setAmtDue(BigDecimal value) {
    this.amtDue = value;
}

public String getOrderNumber() {
    return orderNumber;
}

public void setOrderNumber(String value) {
    this.orderNumber = value;
}

public boolean isPaid() {
    if (paid == null) {
        return false;
    } else {
        return paid;
    }
}

public void setPaid(Boolean value) {
    this.paid = value;
}
}

```

35.5. OCCURRENCE CONSTRAINTS

35.5.1. Schema Elements Supporting Occurrence Constraints

XML Schema allows you to specify the occurrence constraints on four of the XML Schema elements that make up a complex type definition:

- [Section 35.5.2, "Occurrence Constraints on the All Element"](#)
- [Section 35.5.3, "Occurrence Constraints on the Choice Element"](#)
- [Section 35.5.4, "Occurrence Constraints on Elements"](#)
- [Section 35.5.5, "Occurrence Constraints on Sequences"](#)

35.5.2. Occurrence Constraints on the All Element

XML Schema

Complex types defined with the **all** element do not allow for multiple occurrences of the structure defined by the **all** element. You can, however, make the structure defined by the **all** element optional by setting its **minOccurs** attribute to **0**.

Mapping to Java

Setting the **all** element's **minOccurs** attribute to **0** has no effect on the generated Java class.

35.5.3. Occurrence Constraints on the Choice Element

Overview

By default, the results of a **choice** element can only appear once in an instance of a complex type. You can change the number of times the element chosen to represent the structure defined by a **choice** element is allowed to appear using its **minOccurs** attribute and its **maxOccurs** attribute. Using these attributes you can specify that the choice type can occur zero to an unlimited number of times in an instance of a complex type. The element chosen for the choice type does not need to be the same for each occurrence of the type.

Using in XML Schema

The **minOccurs** attribute specifies the minimum number of times the choice type must appear. Its value can be any positive integer. Setting the **minOccurs** attribute to **0** specifies that the choice type does not need to appear inside an instance of the complex type.

The **maxOccurs** attribute specifies the maximum number of times the choice type can appear. Its value can be any non-zero, positive integer or **unbounded**. Setting the **maxOccurs** attribute to **unbounded** specifies that the choice type can appear an infinite number of times.

[Example 35.18, “Choice Occurrence Constraints”](#) shows the definition of a choice type, **ClubEvent**, with choice occurrence constraints. The choice type overall can be repeated 0 to unbounded times.

Example 35.18. Choice Occurrence Constraints

```
<complexType name="ClubEvent">
  <choice minOccurs="0" maxOccurs="unbounded">
    <element name="MemberName" type="xsd:string"/>
    <element name="GuestName" type="xsd:string"/>
  </choice>
</complexType>
```

Mapping to Java

Unlike single instance choice structures, XML Schema choice structures that can occur multiple times are mapped to a Java class with a single member variable. This single member variable is a **List<T>** object that holds all of the data for the multiple occurrences of the sequence. For example, if the sequence defined in [Example 35.18, “Choice Occurrence Constraints”](#) occurred two times, then the list would have two items.

The name of the Java class' member variable is derived by concatenating the names of the member elements. The element names are separated by **Or** and the first letter of the variable name is converted to lower case. For example, the member variable generated from [Example 35.18, “Choice Occurrence Constraints”](#) would be named **memberNameOrGuestName**.

The type of object stored in the list depends on the relationship between the types of the member elements. For example:

- If the member elements are of the same type the generated list will contain **JAXBElement<T>** objects. The base type of the **JAXBElement<T>** objects is determined by the normal mapping of the member elements' type.

- If the member elements are of different types and their Java representations implement a common interface, the list will contain objects of the common interface.
- If the member elements are of different types and their Java representations extend a common base class, the list will contain objects of the common base class.
- If none of the other conditions are met, the list will contain **Object** objects.

The generated Java class will only have a getter method for the member variable. The getter method returns a reference to the live list. Any modifications made to the returned list will effect the actual object.

The Java class is decorated with the **@XmlType** annotation. The annotation's **name** property is set to the value of the **name** attribute from the parent element of the XML Schema definition. The annotation's **propOrder** property contains the single member variable representing the elements in the sequence.

The member variable representing the elements in the choice structure are decorated with the **@XmlElement** annotation. The **@XmlElement** annotation contains a comma separated list of **@XmlElement** annotations. The list has one **@XmlElement** annotation for each member element defined in the XML Schema definition of the type. The **@XmlElement** annotations in the list have their **name** property set to the value of the XML Schema **element** element's **name** attribute and their **type** property set to the Java class resulting from the mapping of the XML Schema **element** element's type.

[Example 35.19, "Java Representation of Choice Structure with an Occurrence Constraint"](#) shows the Java mapping for the XML Schema choice structure defined in [Example 35.18, "Choice Occurrence Constraints"](#).

Example 35.19. Java Representation of Choice Structure with an Occurrence Constraint

```

@XmlType(name = "ClubEvent", propOrder = {
    "memberNameOrGuestName"
})
public class ClubEvent {

    @XmlElementRefs({
        @XmlElementRef(name = "GuestName", type = JAXBElement.class),
        @XmlElementRef(name = "MemberName", type = JAXBElement.class)
    })
    protected List<JAXBElement<String>> memberNameOrGuestName;

    public List<JAXBElement<String>> getMemberNameOrGuestName() {
        if (memberNameOrGuestName == null) {
            memberNameOrGuestName = new ArrayList<JAXBElement<String>>();
        }
        return this.memberNameOrGuestName;
    }
}

```

minOccurs set to 0

If only the **minOccurs** element is specified and its value is **0**, the code generators generate the Java class as if the **minOccurs** attribute were not set.

35.5.4. Occurrence Constraints on Elements

Overview

You can specify how many times a specific element in a complex type appears using the **element** element's **minOccurs** attribute and **maxOccurs** attribute. The default value for both attributes is **1**.

minOccurs set to 0

When you set one of the complex type's member element's **minOccurs** attribute to **0**, the **@XmlElement** annotation decorating the corresponding Java member variable is changed. Instead of having its **required** property set to **true**, the **@XmlElement** annotation's **required** property is set to **false**.

minOccurs set to a value greater than 1

In XML Schema you can specify that an element must occur more than once in an instance of the type by setting the **element** element's **minOccurs** attribute to a value greater than one. However, the generated Java class will not support the XML Schema constraint. Apache CXF generates the supporting Java member variable as if the **minOccurs** attribute were not set.

Elements with maxOccurs set

When you want a member element to appear multiple times in an instance of a complex type, you set the element's **maxOccurs** attribute to a value greater than 1. You can set the **maxOccurs** attribute's value to **unbounded** to specify that the member element can appear an unlimited number of times.

The code generators map a member element with the **maxOccurs** attribute set to a value greater than 1 to a Java member variable that is a **List<T>** object. The base class of the list is determined by mapping the element's type to Java. For XML Schema primitive types, the wrapper classes are used as described in [the section called "Wrapper classes"](#). For example, if the member element is of type **xsd:int** the generated member variable is a **List<Integer>** object.

35.5.5. Occurrence Constraints on Sequences

Overview

By default, the contents of a **sequence** element can only appear once in an instance of a complex type. You can change the number of times the sequence of elements defined by a **sequence** element is allowed to appear using its **minOccurs** attribute and its **maxOccurs** attribute. Using these attributes you can specify that the sequence type can occur zero to an unlimited number of times in an instance of a complex type.

Using XML Schema

The **minOccurs** attribute specifies the minimum number of times the sequence must occur in an instance of the defined complex type. Its value can be any positive integer. Setting the **minOccurs** attribute to **0** specifies that the sequence does not need to appear inside an instance of the complex type.

The **maxOccurs** attribute specifies the upper limit for how many times the sequence can occur in an instance of the defined complex type. Its value can be any non-zero, positive integer or **unbounded**. Setting the **maxOccurs** attribute to **unbounded** specifies that the sequence can appear an infinite number of times.

[Example 35.20, "Sequence with Occurrence Constraints"](#) shows the definition of a sequence type, `CultureInfo`, with sequence occurrence constraints. The sequence can be repeated 0 to 2 times.

Example 35.20. Sequence with Occurrence Constraints

```
<complexType name="CultureInfo">
  <sequence minOccurs="0" maxOccurs="2">
    <element name="Name" type="string"/>
    <element name="Lcid" type="int"/>
  </sequence>
</complexType>
```

Mapping to Java

Unlike single instance sequences, XML Schema sequences that can occur multiple times are mapped to a Java class with a single member variable. This single member variable is a `List<T>` object that holds all of the data for the multiple occurrences of the sequence. For example, if the sequence defined in [Example 35.20, "Sequence with Occurrence Constraints"](#) occurred two times, then the list would have four items.

The name of the Java class' member variable is derived by concatenating the names of the member elements. The element names are separated by **And** and the first letter of the variable name is converted to lower case. For example, the member variable generated from [Example 35.20, "Sequence with Occurrence Constraints"](#) is named `nameAndLcid`.

The type of object stored in the list depends on the relationship between the types of the member elements. For example:

- If the member elements are of the same type the generated list will contain `JAXBElement<T>` objects. The base type of the `JAXBElement<T>` objects is determined by the normal mapping of the member elements' type.
- If the member elements are of different types and their Java representations implement a common interface, the list will contain objects of the common interface.
- If the member elements are of different types and their Java representations extend a common base class, the list will contain objects of the common base class.
- If none of the other conditions are met, the list will contain `Object` objects.

The generated Java class only has a getter method for the member variable. The getter method returns a reference to the live list. Any modifications made to the returned list effects the actual object.

The Java class is decorated with the `@XmlType` annotation. The annotation's `name` property is set to the value of the `name` attribute from the parent element of the XML Schema definition. The annotation's `propOrder` property contains the single member variable representing the elements in the sequence.

The member variable representing the elements in the sequence are decorated with the `@XmlElement` annotation. The `@XmlElement` annotation contains a comma separated list of `@XmlElement` annotations. The list has one `@XmlElement` annotation for each member element defined in the XML Schema definition of the type. The `@XmlElement` annotations in the list have their `name` property set to the value of the XML Schema `element` element's `name` attribute and their `type` property set to the Java class resulting from the mapping of the XML Schema `element` element's type.

[Example 35.21, “Java Representation of Sequence with an Occurrence Constraint”](#) shows the Java mapping for the XML Schema sequence defined in [Example 35.20, “Sequence with Occurrence Constraints”](#).

Example 35.21. Java Representation of Sequence with an Occurrence Constraint

```
@XmlType(name = "CultureInfo", propOrder = {
    "nameAndLcid"
})
public class CultureInfo {

    @XmlElement({
        @XmlElement(name = "Name", type = String.class),
        @XmlElement(name = "Lcid", type = Integer.class)
    })
    protected List<Serializable> nameAndLcid;

    public List<Serializable> getNameAndLcid() {
        if (nameAndLcid == null) {
            nameAndLcid = new ArrayList<Serializable>();
        }
        return this.nameAndLcid;
    }

}
```

minOccurs set to 0

If only the **minOccurs** element is specified and its value is **0**, the code generators generate the Java class as if the **minOccurs** attribute is not set.

35.6. USING MODEL GROUPS

Overview

XML Schema model groups are convenient shortcuts that allows you to reference a group of elements from a user-defined complex type. For example, you can define a group of elements that are common to several types in your application and then reference the group repeatedly. Model groups are defined using the **group** element, and are similar to complex type definitions. The mapping of model groups to Java is also similar to the mapping for complex types.

Defining a model group in XML Schema

You define a model group in XML Schema using the **group** element with the **name** attribute. The value of the **name** attribute is a string that is used to refer to the group throughout the schema. The **group** element, like the **complexType** element, can have the **sequence** element, the **all** element, or the **choice** element as its immediate child.

Inside the **child** element, you define the members of the group using **element** elements. For each member of the group, specify one **element** element. Group members can use any of the standard attributes for the **element** element including **minOccurs** and **maxOccurs**. So, if your group has three

elements and one of them can occur up to three times, you define a group with three **element** elements, one of which uses maxOccurs="3". [Example 35.22, "XML Schema Model Group"](#) shows a model group with three elements.

Example 35.22. XML Schema Model Group

```
<group name="passenger">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="clubNum" type="xsd:long" />
    <element name="seatPref" type="xsd:string"
      maxOccurs="3" />
  </sequence>
</group>
```

Using a model group in a type definition

Once a model group has been defined, it can be used as part of a complex type definition. To use a model group in a complex type definition, use the **group** element with the **ref** attribute. The value of the **ref** attribute is the name given to the group when it was defined. For example, to use the group defined in [Example 35.22, "XML Schema Model Group"](#) you use `<group ref="tns:passenger" />` as shown in [Example 35.23, "Complex Type with a Model Group"](#).

Example 35.23. Complex Type with a Model Group

```
<complexType name="reservation">
  <sequence>
    <group ref="tns:passenger" />
    <element name="origin" type="xsd:string" />
    <element name="destination" type="xsd:string" />
    <element name="fltNum" type="xsd:long" />
  </sequence>
</complexType>
```

When a model group is used in a type definition, the group becomes a member of the type. So an instance of **reservation** has four member elements. The first element is the **passenger** element and it contains the member elements defined by the group shown in [Example 35.22, "XML Schema Model Group"](#). An example of an instance of **reservation** is shown in [Example 35.24, "Instance of a Type with a Model Group"](#).

Example 35.24. Instance of a Type with a Model Group

```
<reservation>
  <passenger> <name>A. Smart</name> <clubNum>99</clubNum> <seatPref>isle1</seatPref>
  </passenger>
  <origin>LAX</origin>
  <destination>FRA</destination>
  <fltNum>34567</fltNum>
</reservation>
```

Mapping to Java

By default, a model group is only mapped to Java artifacts when it is included in a complex type definition. When generating code for a complex type that includes a model group, Apache CXF simply includes the member variables for the model group into the Java class generated for the type. The member variables representing the model group are annotated based on the definitions of the model group.

[Example 35.25, “Type with a Group”](#) shows the Java class generated for the complex type defined in [Example 35.23, “Complex Type with a Model Group”](#).

Example 35.25. Type with a Group

```
@XmlType(name = "reservation", propOrder = {
    "name",
    "clubNum",
    "seatPref",
    "origin",
    "destination",
    "fltNum"
})
public class Reservation {

    @XmlElement(required = true)
    protected String name;
    protected long clubNum;
    @XmlElement(required = true)
    protected List<String> seatPref;
    @XmlElement(required = true)
    protected String origin;
    @XmlElement(required = true)
    protected String destination;
    protected long fltNum;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public long getClubNum() {
        return clubNum;
    }

    public void setClubNum(long value) {
        this.clubNum = value;
    }

    public List<String> getSeatPref() {
        if (seatPref == null) {
            seatPref = new ArrayList<String>();
        }
        return this.seatPref;
    }
}
```

```
public String getOrigin() {
    return origin;
}

public void setOrigin(String value) {
    this.origin = value;
}

public String getDestination() {
    return destination;
}

public void setDestination(String value) {
    this.destination = value;
}

public long getFltNum() {
    return fltNum;
}

public void setFltNum(long value) {
    this.fltNum = value;
}
```

Multiple occurrences

You can specify that the model group appears more than once by setting the **group** element's **maxOccurs** attribute to a value greater than one. To allow for multiple occurrences of the model group Apache CXF maps the model group to a **List<T>** object. The **List<T>** object is generated following the rules for the group's first child:

- If the group is defined using a **sequence** element see [Section 35.5.5, "Occurrence Constraints on Sequences"](#).
- If the group is defined using a **choice** element see [Section 35.5.3, "Occurrence Constraints on the Choice Element"](#).

CHAPTER 36. USING WILD CARD TYPES

Abstract

There are instances when a schema author wants to defer binding elements or attributes to a defined type. For these cases, XML Schema provides three mechanisms for specifying wild card place holders. These are all mapped to Java in ways that preserve their XML Schema functionality.

36.1. USING ANY ELEMENTS

Overview

The XML Schema **any** element is used to create a wild card place holder in complex type definitions. When an XML element is instantiated for an XML Schema **any** element, it can be any valid XML element. The **any** element does not place any restrictions on either the content or the name of the instantiated XML element.

For example, given the complex type defined in [Example 36.1, "XML Schema Type Defined with an Any Element"](#) you can instantiate either of the XML elements shown in [Example 36.2, "XML Document with an Any Element"](#).

Example 36.1. XML Schema Type Defined with an Any Element

```
<element name="FlyBoy">
  <complexType>
    <sequence>
      <any />
      <element name="rank" type="xsd:int" />
    </sequence>
  </complexType>
</element>
```

Example 36.2. XML Document with an Any Element

```
<FlyBoy>
  <learJet>CL-215</learJet>
  <rank>2</rank>
</element>
<FlyBoy>
  <viper>Mark II</viper>
  <rank>1</rank>
</element>
```

XML Schema **any** elements are mapped to either a Java **Object** object or a Java **org.w3c.dom.Element** object.

Specifying in XML Schema

The **any** element can be used when defining sequence complex types and choice complex types. In most cases, the **any** element is an empty element. It can, however, take an **annotation** element as a child.

[Table 36.1, "Attributes of the XML Schema Any Element"](#) describes the **any** element's attributes.

Table 36.1. Attributes of the XML Schema Any Element

Attribute	Description
namespace	<p>Specifies the namespace of the elements that can be used to instantiate the element in an XML document. The valid values are:</p> <p>##any Specifies that elements from any namespace can be used. This is the default.</p> <p>##other Specifies that elements from any namespace other than the parent element's namespace can be used.</p> <p>##local Specifies elements without a namespace must be used.</p> <p>##targetNamespace Specifies that elements from the parent element's namespace must be used.</p> <p>A space delimited list of URIs##local and ##targetNamespace Specifies that elements from any of the listed namespaces can be used.</p>
maxOccurs	<p>Specifies the maximum number of times an instance of the element can appear in the parent element. The default value is 1. To specify that an instance of the element can appear an unlimited number of times, you can set the attribute's value to unbounded.</p>
minOccurs	<p>Specifies the minimum number of times an instance of the element can appear in the parent element. The default value is 1.</p>

Attribute	Description
processContents	<p>Specifies how the element used to instantiate the any element should be validated. Valid values are:</p> <ul style="list-style-type: none"> strict Specifies that the element must be validated against the proper schema. This is the default value. lax Specifies that the element should be validated against the proper schema. If it cannot be validated, no errors are thrown. skip Specifies that the element should not be validated.

[Example 36.3, “Complex Type Defined with an Any Element”](#) shows a complex type defined with an **any** element

Example 36.3. Complex Type Defined with an Any Element

```
<complexType name="surprisePackage">
<sequence>
<any processContents="lax" />
<element name="to" type="xsd:string" />
<element name="from" type="xsd:string" />
</sequence>
</complexType>
```

Mapping to Java

XML Schema **any** elements result in the creation of a Java property named **any**. The property has associated getter and setter methods. The type of the resulting property depends on the value of the element’s **processContents** attribute. If the **any** element’s **processContents** attribute is set to **skip**, the element is mapped to a **org.w3c.dom.Element** object. For all other values of the **processContents** attribute an **any** element is mapped to a Java **Object** object.

The generated property is decorated with the **@XmlAnyElement** annotation. This annotation has an optional **lax** property that instructs the runtime what to do when marshaling the data. Its default value is **false** which instructs the runtime to automatically marshal the data into a **org.w3c.dom.Element** object. Setting **lax** to **true** instructs the runtime to attempt to marshal the data into JAXB types. When the **any** element’s **processContents** attribute is set to **skip**, the **lax** property is set to its default value. For all other values of the **processContents** attribute, **lax** is set to **true**.

[Example 36.4, “Java Class with an Any Element”](#) shows how the complex type defined in [Example 36.3, “Complex Type Defined with an Any Element”](#) is mapped to a Java class.

Example 36.4. Java Class with an Any Element

```
public class SurprisePackage {
```

```

@XmlAnyElement(lax = true) protected Object any;
@XmlElement(required = true)
protected String to;
@XmlElement(required = true)
protected String from;

public Object getAny() { return any; }

public void setAny(Object value) { this.any = value; }

public String getTo() {
    return to;
}

public void setTo(String value) {
    this.to = value;
}

public String getFrom() {
    return from;
}

public void setFrom(String value) {
    this.from = value;
}

}

```

Marshalling

If the Java property for an **any** element has its **lax** set to **false**, or the property is not specified, the runtime makes no attempt to parse the XML data into JAXB objects. The data is always stored in a DOM **Element** object.

If the Java property for an **any** element has its **lax** set to **true**, the runtime attempts to marshal the XML data into the appropriate JAXB objects. The runtime attempts to identify the proper JAXB classes using the following procedure:

1. It checks the element tag of the XML element against the list of elements known to the runtime. If it finds a match, the runtime marshals the XML data into the proper JAXB class for the element.
2. It checks the XML element's **xsi:type** attribute. If it finds a match, the runtime marshals the XML element into the proper JAXB class for that type.
3. If it cannot find a match it marshals the XML data into a DOM **Element** object.

Usually an application's runtime knows about all of the types generated from the schema's included in its contract. This includes the types defined in the contract's **wsdl:types** element, any data types added to the contract through inclusion, and any types added to the contract through importing other schemas. You can also make the runtime aware of additional types using the **@XmlSeeAlso** annotation which is described in [Section 32.4, "Adding Classes to the Runtime Marshaller"](#).

Unmarshalling

If the Java property for an **any** element has its **lax** set to **false**, or the property is not specified, the runtime will only accept DOM **Element** objects. Attempting to use any other type of object will result in a marshalling error.

If the Java property for an **any** element has its **lax** set to **true**, the runtime uses its internal map between Java data types and the XML Schema constructs they represent to determine the XML structure to write to the wire. If the runtime knows the class and can map it to an XML Schema construct, it writes out the data and inserts an **xsi:type** attribute to identify the type of data the element contains.

If the runtime cannot map the Java object to a known XML Schema construct, it will throw a marshaling exception. You can add types to the runtime's map using the **@XmlSeeAlso** annotation which is described in [Section 32.4, "Adding Classes to the Runtime Marshaller"](#).

36.2. USING THE XML SCHEMA ANYTYPE TYPE

Overview

The XML Schema type **xsd:anyType** is the root type for all XML Schema types. All of the primitives are derivatives of this type, as are all user defined complex types. As a result, elements defined as being of **xsd:anyType** can contain data in the form of any of the XML Schema primitives as well as any complex type defined in a schema document.

In Java the closest matching type is the **Object** class. It is the class from which all other Java classes are sub-typed.

Using in XML Schema

You use the **xsd:anyType** type as you would any other XML Schema complex type. It can be used as the value of an **element** element's **type** element. It can also be used as the base type from which other types are defined.

[Example 36.5, "Complex Type with a Wild Card Element"](#) shows an example of a complex type that contains an element of type **xsd:anyType**.

Example 36.5. Complex Type with a Wild Card Element

```
<complexType name="wildStar">
<sequence>
  <element name="name" type="xsd:string" />
  <element name="ship" type="xsd:anyType" />
</sequence>
</complexType>
```

Mapping to Java

Elements that are of type **xsd:anyType** are mapped to **Object** objects. [Example 36.6, "Java Representation of a Wild Card Element"](#) shows the mapping of [Example 36.5, "Complex Type with a Wild Card Element"](#) to a Java class.

Example 36.6. Java Representation of a Wild Card Element

```

public class WildStar {

    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true) protected Object ship;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public Object getShip() { return ship; }

    public void setShip(Object value) { this.ship = value; }
}

```

This mapping allows you to place any data into the property representing the wild card element. The Apache CXF runtime handles the marshaling and unmarshaling of the data into usable Java representation.

Marshalling

When Apache CXF marshals XML data into Java types, it attempts to marshal **anyType** elements into known JAXB objects. To determine if it is possible to marshal an **anyType** element into a JAXB generated object, the runtime inspects the element's **xsi:type** attribute to determine the actual type used to construct the data in the element. If the **xsi:type** attribute is not present, the runtime attempts to identify the element's actual data type by introspection. If the element's actual data type is determined to be one of the types known by the application's JAXB context, the element is marshaled into a JAXB object of the proper type.

If the runtime cannot determine the actual data type of the element, or the actual data type of the element is not a known type, the runtime marshals the content into a **org.w3c.dom.Element** object. You will then need to work with the element's content using the DOM APIs.

An application's runtime usually knows about all of the types generated from the schema's included in its contract. This includes the types defined in the contract's **wsdl:types** element, any data types added to the contract through inclusion, and any types added to the contract through importing other schema documents. You can also make the runtime aware of additional types using the **@XmlSeeAlso** annotation which is described in [Section 32.4, “Adding Classes to the Runtime Marshaller”](#).

Unmarshalling

When Apache CXF unmarshals Java types into XML data, it uses an internal map between Java data types and the XML Schema constructs they represent to determine the XML structure to write to the wire. If the runtime knows the class and can map the class to an XML Schema construct, it writes out the data and inserts an **xsi:type** attribute to identify the type of data the element contains. If the data is stored in a **org.w3c.dom.Element** object, the runtime writes the XML structure represented by the object but it does not include an **xsi:type** attribute.

If the runtime cannot map the Java object to a known XML Schema construct, it throws a marshaling exception. You can add types to the runtime's map using the `@XmlSeeAlso` annotation which is described in [Section 32.4, "Adding Classes to the Runtime Marshaller"](#).

36.3. USING UNBOUND ATTRIBUTES

Overview

XML Schema has a mechanism that allows you to leave a place holder for an arbitrary attribute in a complex type definition. Using this mechanism, you can define a complex type that can have any attribute. For example, you can create a type that defines the elements `<robot name="epsilon" />`, `<robot age="10000" />`, or `<robot type="weevil" />` without specifying the three attributes. This can be particularly useful when flexibility in your data is required.

Defining in XML Schema

Undeclared attributes are defined in XML Schema using the `anyAttribute` element. It can be used wherever an attribute element can be used. The `anyAttribute` element has no attributes, as shown in [Example 36.7, "Complex Type with an Undeclared Attribute"](#).

Example 36.7. Complex Type with an Undeclared Attribute

```
<complexType name="arbitter">
<sequence>
  <element name="name" type="xsd:string" />
  <element name="rate" type="xsd:float" />
</sequence>
<anyAttribute />
</complexType>
```

The defined type, **arbitter**, has two elements and can have one attribute of any type. The elements three elements shown in [Example 36.8, "Examples of Elements Defined with a Wild Card Attribute"](#) can all be generated from the complex type **arbitter**.

Example 36.8. Examples of Elements Defined with a Wild Card Attribute

```
<officer rank="12"><name>...</name><rate>...</rate></officer>
<lawyer type="divorce"><name>...</name><rate>...</rate></lawyer>
<judge><name>...</name><rate>...</rate></judge>
```

Mapping to Java

When a complex type containing an `anyAttribute` element is mapped to Java, the code generator adds a member called **otherAttributes** to the generated class. **otherAttributes** is of type `java.util.Map<QName, String>` and it has a getter method that returns a live instance of the map. Because the map returned from the getter is live, any modifications to the map are automatically applied. [Example 36.9, "Class for a Complex Type with an Undeclared Attribute"](#) shows the class generated for the complex type defined in [Example 36.7, "Complex Type with an Undeclared Attribute"](#).

Example 36.9. Class for a Complex Type with an Undeclared Attribute

```

public class Arbitter {

    @XmlElement(required = true)
    protected String name;
    protected float rate;

    @XmlAnyAttribute private Map<QName, String> otherAttributes = new HashMap<QName,
String>();

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public float getRate() {
        return rate;
    }

    public void setRate(float value) {
        this.rate = value;
    }

    public Map<QName, String> getOtherAttributes() { return otherAttributes; }

}

```

Working with undeclared attributes

The **otherAttributes** member of the generated class expects to be populated with a **Map** object. The map is keyed using **QNames**. Once you get the map , you can access any attributes set on the object and set new attributes on the object.

[Example 36.10, “Working with Undeclared Attributes”](#) shows sample code for working with undeclared attributes.

Example 36.10. Working with Undeclared Attributes

```

Arbitter judge = new Arbitter();
Map<QName, String> otherAtts = judge.getOtherAttributes();

QName at1 = new QName("test.apache.org", "house");
QName at2 = new QName("test.apache.org", "veteran");

otherAtts.put(at1, "Cape");
otherAtts.put(at2, "false");

String vetStatus = otherAtts.get(at2);

```

The code in [Example 36.10, “Working with Undeclared Attributes”](#) does the following:

Gets the map containing the undeclared attributes.

Creates QNames to work with the attributes.

Sets the values for the attributes into the map.

Retrieves the value for one of the attributes.

CHAPTER 37. ELEMENT SUBSTITUTION

Abstract

XML Schema substitution groups allow you to define a group of elements that can replace a top level, or head, element. This is useful in cases where you have multiple elements that share a common base type or with elements that need to be interchangeable.

37.1. SUBSTITUTION GROUPS IN XML SCHEMA

Overview

A substitution group is a feature of XML schema that allows you to specify elements that can replace another element in documents generated from that schema. The replaceable element is called the head element and must be defined in the schema's global scope. The elements of the substitution group must be of the same type as the head element or a type that is derived from the head element's type.

In essence, a substitution group allows you to build a collection of elements that can be specified using a generic element. For example, if you are building an ordering system for a company that sells three types of widgets you might define a generic widget element that contains a set of common data for all three widget types. Then you can define a substitution group that contains a more specific set of data for each type of widget. In your contract you can then specify the generic widget element as a message part instead of defining a specific ordering operation for each type of widget. When the actual message is built, the message can contain any of the elements of the substitution group.

Syntax

Substitution groups are defined using the **substitutionGroup** attribute of the XML Schema **element** element. The value of the **substitutionGroup** attribute is the name of the element that the element being defined replaces. For example, if your head element is **widget**, adding the attribute `substitutionGroup="widget"` to an element named **woodWidget** specifies that anywhere a **widget** element is used, you can substitute a **woodWidget** element. This is shown in [Example 37.1, "Using a Substitution Group"](#).

Example 37.1. Using a Substitution Group

```
<element name="widget" type="xsd:string" />
<element name="woodWidget" type="xsd:string"
       substitutionGroup="widget" />
```

Type restrictions

The elements of a substitution group must be of the same type as the head element or of a type derived from the head element's type. For example, if the head element is of type **xsd:int** all members of the substitution group must be of type **xsd:int** or of a type derived from **xsd:int**. You can also define a substitution group similar to the one shown in [Example 37.2, "Substitution Group with Complex Types"](#) where the elements of the substitution group are of types derived from the head element's type.

Example 37.2. Substitution Group with Complex Types

```
<complexType name="widgetType">
```

```

<sequence>
  <element name="shape" type="xsd:string" />
  <element name="color" type="xsd:string" />
</sequence>
</complexType>
<complexType name="woodWidgetType">
  <complexContent>
    <extension base="widgetType">
      <sequence>
        <element name="woodType" type="xsd:string" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
<complexType name="plasticWidgetType">
  <complexContent>
    <extension base="widgetType">
      <sequence>
        <element name="moldProcess" type="xsd:string" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
<element name="widget" type="widgetType" />
<element name="woodWidget" type="woodWidgetType"
  substitutionGroup="widget" />
<element name="plasticWidget" type="plasticWidgetType"
  substitutionGroup="widget" />
<complexType name="partType">
  <sequence>
    <element ref="widget" />
  </sequence>
</complexType>
<element name="part" type="partType" />

```

The head element of the substitution group, **widget**, is defined as being of type **widgetType**. Each element of the substitution group extends **widgetType** to include data that is specific to ordering that type of widget.

Based on the schema in Example 37.2, “Substitution Group with Complex Types”, the **part** elements in Example 37.3, “XML Document using a Substitution Group” are valid.

Example 37.3. XML Document using a Substitution Group

```

<part>
  <widget>
    <shape>round</shape>
    <color>blue</color>
  </widget>
</part>
<part>
  <plasticWidget>
    <shape>round</shape>
    <color>blue</color>
  </plasticWidget>

```

```

<moldProcess>sandCast</moldProcess>
</plasticWidget>
</part>
<part>
<woodWidget>
<shape>round</shape>
<color>blue</color>
<woodType>elm</woodType>
</woodWidget>
</part>

```

Abstract head elements

You can define an abstract head element that can never appear in a document produced using your schema. Abstract head elements are similar to abstract classes in Java because they are used as the basis for defining more specific implementations of a generic class. Abstract heads also prevent the use of the generic element in the final product.

You declare an abstract head element by setting the **abstract** attribute of an **element** element to **true**, as shown in [Example 37.4, "Abstract Head Definition"](#). Using this schema, a valid **review** element can contain either a **positiveComment** element or a **negativeComment** element, but cannot contain a **comment** element.

Example 37.4. Abstract Head Definition

```

<element name="comment" type="xsd:string" abstract="true" />
<element name="positiveComment" type="xsd:string"
    substitutionGroup="comment" />
<element name="negativeComment" type="xsd:string"
    substitutionGroup="comment" />
<element name="review">
    <complexContent>
        <all>
            <element name="custName" type="xsd:string" />
            <element name="impression" ref="comment" />
        </all>
    </complexContent>
</element>

```

37.2. SUBSTITUTION GROUPS IN JAVA

Overview

Apache CXF, as specified in the JAXB specification, supports substitution groups using Java's native class hierarchy in combination with the ability of the **JAXBELEMENT** class' support for wildcard definitions. Because the members of a substitution group must all share a common base type, the classes generated to support the elements' types also share a common base type. In addition, Apache CXF maps instances of the head element to **JAXBELEMENT<? extends T>** properties.

Generated object factory methods

The object factory generated to support a package containing a substitution group has methods for each of the elements in the substitution group. For each of the members of the substitution group, except for the head element, the **@XmlElementDecl** annotation decorating the object factory method includes two additional properties, as described in [Table 37.1, “Properties for Declaring a JAXB Element is a Member of a Substitution Group”](#).

Table 37.1. Properties for Declaring a JAXB Element is a Member of a Substitution Group

Property	Description
substitutionHeadNamespace	Specifies the namespace where the head element is defined.
substitutionHeadName	Specifies the value of the head element's name attribute.

The object factory method for the head element of the substitution group's **@XmlElementDecl** contains only the default **namespace** property and the default **name** property.

In addition to the element instantiation methods, the object factory contains a method for instantiating an object representing the head element. If the members of the substitution group are all of complex types, the object factory also contains methods for instantiating instances of each complex type used.

[Example 37.5, “Object Factory Method for a Substitution Group”](#) shows the object factory method for the substitution group defined in [Example 37.2, “Substitution Group with Complex Types”](#).

Example 37.5. Object Factory Method for a Substitution Group

```
public class ObjectFactory {

    private final static QName _Widget_QNAME = new QName(...);
    private final static QName _PlasticWidget_QNAME = new QName(...);
    private final static QName _WoodWidget_QNAME = new QName(...);

    public ObjectFactory() {
    }

    public WidgetType createWidgetType() {
        return new WidgetType();
    }

    public PlasticWidgetType createPlasticWidgetType() {
        return new PlasticWidgetType();
    }

    public WoodWidgetType createWoodWidgetType() {
        return new WoodWidgetType();
    }

    @XmlElementDecl(namespace="...", name = "widget")
    public JAXBElement<WidgetType> createWidget(WidgetType value) {
        return new JAXBElement<WidgetType>(_Widget_QNAME, WidgetType.class, null, value);
    }
}
```

```

    @XmlElementDecl(namespace = "...", name = "plasticWidget", substitutionHeadNamespace =
    "...", substitutionHeadName = "widget")
    public JAXBElement<PlasticWidgetType> createPlasticWidget(PlasticWidgetType value) {
        return new JAXBElement<PlasticWidgetType>(_PlasticWidget_QNAME,
PlasticWidgetType.class, null, value);
    }

    @XmlElementDecl(namespace = "...", name = "woodWidget", substitutionHeadNamespace =
    "...", substitutionHeadName = "widget")
    public JAXBElement<WoodWidgetType> createWoodWidget(WoodWidgetType value) {
        return new JAXBElement<WoodWidgetType>(_WoodWidget_QNAME,
WoodWidgetType.class, null, value);
    }

}

```

Substitution groups in interfaces

If the head element of a substitution group is used as a message part in one of an operation's messages, the resulting method parameter will be an object of the class generated to support that element. It will not necessarily be an instance of the **JAXBElement<? extends T>** class. The runtime relies on Java's native type hierarchy to support the type substitution, and Java will catch any attempts to use unsupported types.

To ensure that the runtime knows all of the classes needed to support the element substitution, the SEI is decorated with the **@XmlSeeAlso** annotation. This annotation specifies a list of classes required by the runtime for marshalling. For more information on using the **@XmlSeeAlso** annotation see [Section 32.4, "Adding Classes to the Runtime Marshaller"](#).

[Example 37.7, "Generated Interface Using a Substitution Group"](#) shows the SEI generated for the interface shown in [Example 37.6, "WSDL Interface Using a Substitution Group"](#). The interface uses the substitution group defined in [Example 37.2, "Substitution Group with Complex Types"](#).

Example 37.6. WSDL Interface Using a Substitution Group

```

<message name="widgetMessage">
    <part name="widgetPart" element="xsd1:widget" />
</message>
<message name="numWidgets">
    <part name="numInventory" type="xsd:int" />
</message>
<message name="badSize">
    <part name="numInventory" type="xsd:int" />
</message>
<portType name="orderWidgets">
    <operation name="placeWidgetOrder">
        <input message="tns:widgetOrder" name="order" />
        <output message="tns:widgetOrderBill" name="bill" />
        <fault message="tns:badSize" name="sizeFault" />
    </operation>
    <operation name="checkWidgets">
        <input message="tns:widgetMessage" name="request" />

```

```

<output message="tns:numWidgets" name="response" />
</operation>
</portType>
```

Example 37.7. Generated Interface Using a Substitution Group

```

@WebService(targetNamespace = "...", name = "orderWidgets")
@XmlSeeAlso({com.widgetvendor.types.widgettypes.ObjectFactory.class})
public interface OrderWidgets {

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "numInventory", targetNamespace = "", partName = "numInventory")
    @WebMethod
    public int checkWidgets(
        @WebParam(partName = "widgetPart", name = "widget", targetNamespace = "...")
        com.widgetvendor.types.widgettypes.WidgetType widgetPart
    );
}
```

The SEI shown in [Example 37.7, “Generated Interface Using a Substitution Group”](#) lists the object factory in the **@XmlSeeAlso** annotation. Listing the object factory for a namespace provides access to all of the generated classes for that namespace.

Substitution groups in complex types

When the head element of a substitution group is used as an element in a complex type, the code generator maps the element to a **JAXBElement<? extends T>** property. It does not map it to a property containing an instance of the generated class generated to support the substitution group.

For example, the complex type defined in [Example 37.8, “Complex Type Using a Substitution Group”](#) results in the Java class shown in [Example 37.9, “Java Class for a Complex Type Using a Substitution Group”](#). The complex type uses the substitution group defined in [Example 37.2, “Substitution Group with Complex Types”](#).

Example 37.8. Complex Type Using a Substitution Group

```

<complexType name="widgetOrderInfo">
    <sequence>
        <element name="amount" type="xsd:int"/>
        <element ref="xsd1:widget"/>
    </sequence>
</complexType>
```

Example 37.9. Java Class for a Complex Type Using a Substitution Group

```

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "widgetOrderInfo", propOrder = {"amount", "widget"})
public class WidgetOrderInfo {

    protected int amount;
```

```

@XmlElementRef(name = "widget", namespace = "...", type = JAXBELEMENT.class) protected
JAXBELEMENT<? extends WidgetType> widget;
public int getAmount() {
    return amount;
}

public void setAmount(int value) {
    this.amount = value;
}

public JAXBELEMENT<? extends WidgetType> getWidget() { return widget; }

public void setWidget(JAXBELEMENT<? extends WidgetType> value) { this.widget =
((JAXBELEMENT<? extends WidgetType>) value); }

}

```

Setting a substitution group property

How you work with a substitution group depends on whether the code generator mapped the group to a straight Java class or to a **JAXBELEMENT<? extends T>** class. When the element is simply mapped to an object of the generated value class, you work with the object the same way you work with other Java objects that are part of a type hierarchy. You can substitute any of the subclasses for the parent class. You can inspect the object to determine its exact class, and cast it appropriately.

The JAXB specification recommends that you use the object factory methods for instantiating objects of the generated classes.

When the code generators create a **JAXBELEMENT<? extends T>** object to hold instances of a substitution group, you must wrap the element's value in a **JAXBELEMENT<? extends T>** object. The best method to do this is to use the element creation methods provided by the object factory. They provide an easy means for creating an element based on its value.

[Example 37.10, “Setting a Member of a Substitution Group”](#) shows code for setting an instance of a substitution group.

Example 37.10. Setting a Member of a Substitution Group

```

ObjectFactory of = new ObjectFactory();
PlasticWidgetType pWidget = of.createPlasticWidgetType();
pWidget.setShape = "round";
pWidget.setColor = "green";
pWidget.setMoldProcess = "injection";

JAXBELEMENT<PlasticWidgetType> widget = of.createPlasticWidget(pWidget);

WidgetOrderInfo order = of.createWidgetOrderInfo();
order.setWidget(widget);

```

The code in [Example 37.10, “Setting a Member of a Substitution Group”](#) does the following:

Instantiates an object factory.

Instantiates a **PlasticWidgetType** object.

Instantiates a **JAXBELEMENT<PlasticWidgetType>** object to hold a plastic widget element.

Instantiates a **WidgetOrderInfo** object.

Sets the **WidgetOrderInfo** object's **widget** to the **JAXBELEMENT** object holding the plastic widget element.

Getting the value of a substitution group property

The object factory methods do not help when extracting the element's value from a **JAXBELEMENT<? extends T>** object. You must use the **JAXBELEMENT<? extends T>** object's **getValue()** method. The following options determine the type of object returned by the **getValue()** method:

- Use the **isInstance()** method of all the possible classes to determine the class of the element's value object.
- Use the **JAXBELEMENT<? extends T>** object's **getName()** method to determine the element's name.
The **getName()** method returns a QName. Using the local name of the element, you can determine the proper class for the value object.
- Use the **JAXBELEMENT<? extends T>** object's **getDeclaredType()** method to determine the class of the value object.
The **getDeclaredType()** method returns the **Class** object of the element's value object.



WARNING

There is a possibility that the **getDeclaredType()** method will return the base class for the head element regardless of the actual class of the value object.

[Example 37.11, "Getting the Value of a Member of the Substitution Group"](#) shows code retrieving the value from a substitution group. To determine the proper class of the element's value object the example uses the element's **getName()** method.

Example 37.11. Getting the Value of a Member of the Substitution Group

```
String elementName = order.getWidget().getName().getLocalPart();
if (elementName.equals("woodWidget"))
{
    WoodWidgetType widget=order.getWidget().getValue();
}
else if (elementName.equals("plasticWidget"))
{
    PlasticWidgetType widget=order.getWidget().getValue();
}
else
```

```
{
    WidgetType widget=order.getWidget().getValue();
}
```

37.3. WIDGET VENDOR EXAMPLE

37.3.1. Widget Ordering Interface

This section shows an example of substitution groups being used in Apache CXF to solve a real world application. A service and consumer are developed using the widget substitution group defined in [Example 37.2, "Substitution Group with Complex Types"](#). The service offers two operations: **checkWidgets** and **placeWidgetOrder**. [Example 37.12, "Widget Ordering Interface"](#) shows the interface for the ordering service.

Example 37.12. Widget Ordering Interface

```
<message name="widgetOrder">
    <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
</message>
<message name="widgetOrderBill">
    <part name="widgetOrderConformation"
        type="xsd1:widgetOrderBillInfo"/>
</message>
<message name="widgetMessage">
    <part name="widgetPart" element="xsd1:widget" />
</message>
<message name="numWidgets">
    <part name="numInventory" type="xsd:int" />
</message>
<portType name="orderWidgets">
    <operation name="placeWidgetOrder">
        <input message="tns:widgetOrder" name="order"/>
        <output message="tns:widgetOrderBill" name="bill"/>
    </operation>
    <operation name="checkWidgets">
        <input message="tns:widgetMessage" name="request" />
        <output message="tns:numWidgets" name="response" />
    </operation>
</portType>
```

[Example 37.13, "Widget Ordering SEI"](#) shows the generated Java SEI for the interface.

Example 37.13. Widget Ordering SEI

```
@WebService(targetNamespace = "http://widgetVendor.com/widgetOrderForm", name =
"orderWidgets")
@XmlSeeAlso({com.widgetvendor.types.widgettypes.ObjectFactory.class})
public interface OrderWidgets {

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "numInventory", targetNamespace = "", partName = "numInventory")
```

```

    @WebMethod
    public int checkWidgets(
        @WebParam(partName = "widgetPart", name = "widget", targetNamespace =
        "http://widgetVendor.com/types/widgetTypes")
        com.widgetVendor.types.widgettypes.WidgetType widgetPart
    );

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "widgetOrderConformation", targetNamespace = "", partName =
    "widgetOrderConformation")
    @WebMethod
    public com.widgetVendor.types.widgettypes.WidgetOrderBillInfo placeWidgetOrder(
        @WebParam(partName = "widgetOrderForm", name = "widgetOrderForm",
        targetNamespace = "")
        com.widgetVendor.types.widgettypes.WidgetOrderInfo widgetOrderForm
    ) throws BadSize;
}

```



NOTE

Because the example only demonstrates the use of substitution groups, some of the business logic is not shown.

37.3.2. The `checkWidgets` Operation

Overview

checkWidgets is a simple operation that has a parameter that is the head member of a substitution group. This operation demonstrates how to deal with individual parameters that are members of a substitution group. The consumer must ensure that the parameter is a valid member of the substitution group. The service must properly determine which member of the substitution group was sent in the request.

Consumer implementation

The generated method signature uses the Java class supporting the type of the substitution group's head element. Because the member elements of a substitution group are either of the same type as the head element or of a type derived from the head element's type, the Java classes generated to support the members of the substitution group inherit from the Java class generated to support the head element. Java's type hierarchy natively supports using subclasses in place of the parent class.

Because of how Apache CXF generates the types for a substitution group and Java's type hierarchy, the client can invoke **checkWidgets()** without using any special code. When developing the logic to invoke **checkWidgets()** you can pass in an object of one of the classes generated to support the widget substitution group.

Example 37.14, “Consumer Invoking **checkWidgets()**” shows a consumer invoking **checkWidgets()**.

Example 37.14. Consumer Invoking **checkWidgets()**

```

System.out.println("What type of widgets do you want to order?");
System.out.println("1 - Normal");
System.out.println("2 - Wood");

```

```

System.out.println("3 - Plastic");
System.out.println("Selection [1-3]");
String selection = reader.readLine();
String trimmed = selection.trim();
char widgetType = trimmed.charAt(0);
switch (widgetType)
{
    case '1':
    {
        WidgetType widget = new WidgetType();
        ...
        break;
    }
    case '2':
    {
        WoodWidgetType widget = new WoodWidgetType();
        ...
        break;
    }
    case '3':
    {
        PlasticWidgetType widget = new PlasticWidgetType();
        ...
        break;
    }
    default :
        System.out.println("Invaid Widget Selection!!!");
}
proxy.checkWidgets(widgets);

```

Service implementation

The service's implementation of **checkWidgets()** gets a widget description as a **WidgetType** object, checks the inventory of widgets, and returns the number of widgets in stock. Because all of the classes used to implement the substitution group inherit from the same base class, you can implement **checkWidgets()** without using any JAXB specific APIs.

All of the classes generated to support the members of the substitution group for **widget** extend the **WidgetType** class. Because of this fact, you can use **instanceof** to determine what type of widget was passed in and simply cast the **widgetPart** object into the more restrictive type if appropriate. Once you have the proper type of object, you can check the inventory of the right kind of widget.

[Example 37.15, "Service Implementation of **checkWidgets\(\)**"](#) shows a possible implementation.

Example 37.15. Service Implementation of **checkWidgets()**

```

public int checkWidgets(WidgetType widgetPart)
{
    if (widgetPart instanceof WidgetType)
    {
        return checkWidgetInventory(widgetType);
    }
    else if (widgetPart instanceof WoodWidgetType)

```

```

{
    WoodWidgetType widget = (WoodWidgetType)widgetPart;
    return checkWoodWidgetInventory(widget);
}
else if (widgetPart instanceof PlasticWidgetType)
{
    PlasticWidgetType widget = (PlasticWidgetType)widgetPart;
    return checkPlasticWidgetInventory(widget);
}
}
}

```

37.3.3. The placeWidgetOrder Operation

Overview

placeWidgetOrder uses two complex types containing the substitution group. This operation demonstrates to use such a structure in a Java implementation. Both the consumer and the service must get and set members of a substitution group.

Consumer implementation

To invoke **placeWidgetOrder()** the consumer must construct a widget order containing one element of the widget substitution group. When adding the widget to the order, the consumer should use the object factory methods generated for each element of the substitution group. This ensures that the runtime and the service can correctly process the order. For example, if an order is being placed for a plastic widget, the **ObjectFactory.createPlasticWidget()** method is used to create the element before adding it to the order.

[Example 37.16, “Setting a Substitution Group Member”](#) shows consumer code for setting the **widget** property of the **WidgetOrderInfo** object.

Example 37.16. Setting a Substitution Group Member

```

ObjectFactory of = new ObjectFactory();

WidgetOrderInfo order = new of.createWidgetOrderInfo();
...
System.out.println();
System.out.println("What color widgets do you want to order?");
String color = reader.readLine();
System.out.println();
System.out.println("What shape widgets do you want to order?");
String shape = reader.readLine();
System.out.println();
System.out.println("What type of widgets do you want to order?");
System.out.println("1 - Normal");
System.out.println("2 - Wood");
System.out.println("3 - Plastic");
System.out.println("Selection [1-3]");
String selection = reader.readLine();
String trimmed = selection.trim();
char widgetType = trimmed.charAt(0);
switch (widgetType)

```

```

{
    case '1':
    {
        WidgetType widget = of.createWidgetType();
        widget.setColor(color);
        widget.setShape(shape);
        JAXB<WidgetType> widgetElement = of.createWidget(widget);
        order.setWidget(widgetElement);
        break;
    }
    case '2':
    {
        WoodWidgetType woodWidget = of.createWoodWidgetType();
        woodWidget.setColor(color);
        woodWidget.setShape(shape);
        System.out.println();
        System.out.println("What type of wood are your widgets?");
        String wood = reader.readLine();
        woodWidget.setWoodType(wood);
        JAXB<WoodWidgetType> widgetElement = of.createWoodWidget(woodWidget);
        order.setWoodWidget(widgetElement);
        break;
    }
    case '3':
    {
        PlasticWidgetType plasticWidget = of.createPlasticWidgetType();
        plasticWidget.setColor(color);
        plasticWidget.setShape(shape);
        System.out.println();
        System.out.println("What type of mold to use for your
                           widgets?");
        String mold = reader.readLine();
        plasticWidget.setMoldProcess(mold);
        JAXB<WidgetType> widgetElement = of.createPlasticWidget(plasticWidget);
        order.setPlasticWidget(widgetElement);
        break;
    }
    default :
        System.out.println("Invaid Widget Selection!!");
    }
}

```

Service implementation

The **placeWidgetOrder()** method receives an order in the form of a **WidgetOrderInfo** object, processes the order, and returns a bill to the consumer in the form of a **WidgetOrderBillInfo** object. The orders can be for a plain widget, a plastic widget, or a wooden widget. The type of widget ordered is determined by what type of object is stored in **widgetOrderForm** object's **widget** property. The **widget** property is a substitution group and can contain a **widget** element, a **woodWidget** element, or a **plasticWidget** element.

The implementation must determine which of the possible elements is stored in the order. This can be accomplished using the **JAXBElement<? extends T>** object's **getName()** method to determine the element's QName. The QName can then be used to determine which element in the substitution group

is in the order. Once the element included in the bill is known, you can extract its value into the proper type of object.

[Example 37.17, "Implementation of `placeWidgetOrder\(\)`"](#) shows a possible implementation.

Example 37.17. Implementation of `placeWidgetOrder()`

```
public com.widgetvendor.types.widgettypes.WidgetOrderBillInfo
placeWidgetOrder(WidgetOrderInfo widgetOrderForm)
{
    ObjectFactory of = new ObjectFactory();

    WidgetOrderBillInfo bill = new WidgetOrderBillInfo()

        // Copy the shipping address and the number of widgets
        // ordered from widgetOrderForm to bill
        ...

    int numOrdered = widgetOrderForm.getAmount();

    String elementName = widgetOrderForm.getWidget().getName().getLocalPart();
    if (elementName.equals("woodWidget"))
    {
        WoodWidgetType widget=order.getWidget().getValue();
        buildWoodWidget(widget, numOrdered);

        // Add the widget info to bill
        JAXBElement<WoodWidgetType> widgetElement = of.createWoodWidget(widget);
        bill.setWidget(widgetElement);

        float amtDue = numOrdered * 0.75;
        bill.setAmountDue(amtDue);
    }
    else if (elementName.equals("plasticWidget"))
    {
        PlasticWidgetType widget=order.getWidget().getValue();
        buildPlasticWidget(widget, numOrdered);

        // Add the widget info to bill
        JAXBElement<PlasticWidgetType> widgetElement = of.createPlasticWidget(widget);
        bill.setWidget(widgetElement);

        float amtDue = numOrdered * 0.90;
        bill.setAmountDue(amtDue);
    }
    else
    {
        WidgetType widget=order.getWidget().getValue();
        buildWidget(widget, numOrdered);

        // Add the widget info to bill
        JAXBElement<WidgetType> widgetElement = of.createWidget(widget);
        bill.setWidget(widgetElement);

        float amtDue = numOrdered * 0.30;
        bill.setAmountDue(amtDue);
    }
}
```

```
    }  
  
    return(bill);  
}
```

The code in [Example 37.17, “Implementation of `placeWidgetOrder\(\)`”](#) does the following:

Instantiates an object factory to create elements.

Instantiates a **WidgetOrderBillInfo** object to hold the bill.

Gets the number of widgets ordered.

Gets the local name of the element stored in the order.

Checks to see if the element is a **woodWidget** element.

Extracts the value of the element from the order to the proper type of object.

Creates a **JAXBElement<T>** object placed into the bill.

Sets the bill object's **widget** property.

Sets the bill object's **amountDue** property.

CHAPTER 38. CUSTOMIZING HOW TYPES ARE GENERATED

Abstract

The default JAXB mappings address most of the cases encountered when using XML Schema to define the objects for a Java application. For instances where the default mappings are insufficient, JAXB provides an extensive customization mechanism.

38.1. BASICS OF CUSTOMIZING TYPE MAPPINGS

Overview

The JAXB specification defines a number of XML elements that customize how Java types are mapped to XML Schema constructs. These elements can be specified in-line with XML Schema constructs. If you cannot, or do not want to, modify the XML Schema definitions, you can specify the customizations in external binding document.

Namespace

The elements used to customize the JAXB data bindings are defined in the namespace <http://java.sun.com/xml/ns/jaxb>. You must add a namespace declaration similar to the one shown in Example 38.1, “JAXB Customization Namespace”. This is added to the root element of all XML documents defining JAXB customizations.

Example 38.1. JAXB Customization Namespace

```
 xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
```

Version declaration

When using the JAXB customizations, you must indicate the JAXB version being used. This is done by adding a **jaxb:version** attribute to the root element of the external binding declaration. If you are using in-line customization, you must include the **jaxb:version** attribute in the **schema** element containing the customizations. The value of the attribute is always **2.0**.

Example 38.2, “Specifying the JAXB Customization Version” shows an example of the **jaxb:version** attribute used in a **schema** element.

Example 38.2. Specifying the JAXB Customization Version

```
< schema ...
    jaxb:version="2.0">
```

Using in-line customization

The most direct way to customize how the code generators map XML Schema constructs to Java constructs is to add the customization elements directly to the XML Schema definitions. The JAXB customization elements are placed inside the **xsd:appinfo** element of the XML schema construct that is being modified.

[Example 38.3, "Customized XML Schema"](#) shows an example of a schema containing an in-line JAXB customization.

Example 38.3. Customized XML Schema

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    jaxb:version="2.0">
  <complexType name="size">
    <annotation> <appinfo> <jaxb:class name="widgetSize" /> </appinfo> </annotation>
    <sequence>
      <element name="longSize" type="xsd:string" />
      <element name="numberSize" type="xsd:int" />
    </sequence>
  </complexType>
</schema>
```

Using an external binding declaration

When you cannot, or do not want to, make changes to the XML Schema document that defines your type, you can specify the customizations using an external binding declaration. An external binding declaration consists of a number of nested **jaxb:bindings** elements. [Example 38.4, "JAXB External Binding Declaration Syntax"](#) shows the syntax of an external binding declaration.

Example 38.4. JAXB External Binding Declaration Syntax

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    jaxb:version="2.0">
  <jaxb:bindings [schemaLocation="schemaUri" | wsdlLocation="wsdlUri">
    <jaxb:bindings node="nodeXPath">
      binding declaration
    </jaxb:bindings>
    ...
  </jaxb:bindings>
<jaxb:bindings>
```

The **schemaLocation** attribute and the **wsdlLocation** attribute are used to identify the schema document to which the modifications are applied. Use the **schemaLocation** attribute if you are generating code from a schema document. Use the **wsdlLocation** attribute if you are generating code from a WSDL document.

The **node** attribute is used to identify the specific XML schema construct that is to be modified. It is an XPath statement that resolves to an XML Schema element.

Given the schema document **widgetSchema.xsd**, shown in [Example 38.5, "XML Schema File"](#), the external binding declaration shown in [Example 38.6, "External Binding Declaration"](#) modifies the generation of the complex type **size**.

Example 38.5. XML Schema File

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    version="1.0">
  <complexType name="size">
    <sequence>
      <element name="longSize" type="xsd:string" />
      <element name="numberSize" type="xsd:int" />
    </sequence>
  </complexType>
</schema>
```

Example 38.6. External Binding Declaration

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    jaxb:version="2.0">
  <jaxb:bindings schemaLocation="wsdlSchema.xsd">
    <jaxb:bindings node="xsd:complexType[@name='size']">
      <jaxb:class name="widgetSize" />
    </jaxb:bindings>
  </jaxb:bindings>
</jaxb:bindings>
```

To instruct the code generators to use the external binding declaration use the **wsdl2java** tool's **-b binding-file** option, as shown below:

```
wsdl2java -b widgetBinding.xml widget.wsdl
```

38.2. SPECIFYING THE JAVA CLASS OF AN XML SCHEMA PRIMITIVE

Overview

By default, XML Schema types are mapped to Java primitive types. While this is the most logical mapping between XML Schema and Java, it does not always meet the requirements of the application developer. You might want to map an XML Schema primitive type to a Java class that can hold extra information, or you might want to map an XML primitive type to a class that allows for simple type substitution.

The JAXB **javaType** customization element allows you to customize the mapping between an XML Schema primitive type and a Java primitive type. It can be used to customize the mappings at both the global level and the individual instance level. You can use the **javaType** element as part of a simple type definition or as part of a complex type definition.

When using the **javaType** customization element you must specify methods for converting the XML representation of the primitive type to and from the target Java class. Some mappings have default conversion methods. For instances where there are no default mappings, Apache CXF provides JAXB methods to ease the development of the required methods.

Syntax

The **javaType** customization element takes four attributes, as described in [Table 38.1, "Attributes for Customizing the Generation of a Java Class for an XML Schema Type"](#).

Table 38.1. Attributes for Customizing the Generation of a Java Class for an XML Schema Type

Attribute	Required	Description
name	Yes	Specifies the name of the Java class to which the XML Schema primitive type is mapped. It must be either a valid Java class name or the name of a Java primitive type. You must ensure that this class exists and is accessible to your application. The code generator does not check for this class.
xmlType	No	Specifies the XML Schema primitive type that is being customized. This attribute is only used when the javaType element is used as a child of the globalBindings element.
parseMethod	No	Specifies the method responsible for parsing the string-based XML representation of the data into an instance of the Java class. For more information see the section called "Specifying the converters" .
printMethod	No	Specifies the method responsible for converting a Java object to the string-based XML representation of the data. For more information see the section called "Specifying the converters" .

The **javaType** customization element can be used in three ways:

- To modify all instances of an XML Schema primitive type – The **javaType** element modifies all instances of an XML Schema type in the schema document when it is used as a child of the **globalBindings** customization element. When it is used in this manner, you must specify a value for the **xmlType** attribute that identifies the XML Schema primitive type being modified. [Example 38.7, "Global Primitive Type Customization"](#) shows an in-line global customization that instructs the code generators to use **java.lang.Integer** for all instances of **xsd:short** in the schema.

Example 38.7. Global Primitive Type Customization

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
       xmlns="http://www.w3.org/2001/XMLSchema"
```

```

    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    jaxb:version="2.0">
<annotation>
    <appinfo>
        <jaxb:globalBindings ...>
            <jaxb:javaType name="java.lang.Integer"
                xmlType="xsd:short" />
        </globalBindings>
    </appinfo>
</annotation>
...
</schema>
```

- To modify a simple type definition – The **javaType** element modifies the class generated for all instances of an XML simple type when it is applied to a named simple type definition. When using the **javaType** element to modify a simple type definition, do not use the **xmlType** attribute.

[Example 38.8, “Binding File for Customizing a Simple Type”](#) shows an external binding file that modifies the generation of a simple type named **zipCode**.

Example 38.8. Binding File for Customizing a Simple Type

```

<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    jaxb:version="2.0">
<jaxb:bindings wsdlLocation="widgets.wsdl">
    <jaxb:bindings node="xsd:simpleType[@name='zipCode']">
        <jaxb:javaType name="com.widgetVendor.widgetTypes.zipCodeType"
            parseMethod="com.widgetVendor.widgetTypes.support.parseZipCode"
            printMethod="com.widgetVendor.widgetTypes.support.printZipCode" />
    </jaxb:bindings>
</jaxb:bindings>
<jaxb:bindings>
```

- To modify an element or attribute of a complex type definition – The **javaType** can be applied to individual parts of a complex type definition by including it as part of a JAXB property customization. The **javaType** element is placed as a child to the property’s **baseType** element. When using the **javaType** element to modify a specific part of a complex type definition, do not use the **xmlType** attribute.

[Example 38.9, “Binding File for Customizing an Element in a Complex Type”](#) shows a binding file that modifies an element of a complex type.

Example 38.9. Binding File for Customizing an Element in a Complex Type

```

<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    jaxb:version="2.0">
<jaxb:bindings schemaLocation="enumMap.xsd">
    <jaxb:bindings node="xsd:ComplexType[@name='widgetOrderInfo']">
        <jaxb:bindings node="xsd:element[@name='cost']">
            <jaxb:property>
```

```

<jaxb:baseType>
  <jaxb:javaType name="com.widgetVendor.widgetTypes.costType"
      parseMethod="parseCost"
      printMethod="printCost" >
  </jaxb:baseType>
</jaxb:property>
</jaxb:bindings>
</jaxb:bindings>
</jaxb:bindings>
<jaxb:bindings>

```

For more information on using the **baseType** element see [Section 38.6, "Specifying the Base Type of an Element or an Attribute"](#).

Specifying the converters

The Apache CXF cannot convert XML Schema primitive types into random Java classes. When you use the **javaType** element to customize the mapping of an XML Schema primitive type, the code generator creates an adapter class that is used to marshal and unmarshal the customized XML Schema primitive type. A sample adapter class is shown in [Example 38.10, "JAXB Adapter Class"](#).

Example 38.10. JAXB Adapter Class

```

public class Adapter1 extends XmlAdapter<String, javaType>
{
    public javaType unmarshal(String value)
    {
        return(parseMethod(value));
    }

    public String marshal(javaType value)
    {
        return(printMethod(value));
    }
}

```

parseMethod and *printMethod* are replaced by the value of the corresponding **parseMethod** attribute and **printMethod** attribute. The values must identify valid Java methods. You can specify the method's name in one of two ways:

- A fully qualified Java method name in the form of *packagename.ClassName.methodName*
- A simple method name in the form of *methodName*

When you only provide a simple method name, the code generator assumes that the method exists in the class specified by the **javaType** element's **name** attribute.



IMPORTANT

The code generators **do not** generate parse or print methods. You are responsible for supplying them. For information on developing parse and print methods see [the section called "Implementing converters"](#).

If a value for the **parseMethod** attribute is not provided, the code generator assumes that the Java class specified by the **name** attribute has a constructor whose first parameter is a Java **String** object. The generated adapter's **unmarshal()** method uses the assumed constructor to populate the Java object with the XML data.

If a value for the **printMethod** attribute is not provided, the code generator assumes that the Java class specified by the **name** attribute has a **toString()** method. The generated adapter's **marshal()** method uses the assumed **toString()** method to convert the Java object to XML data.

If the **javaType** element's **name** attribute specifies a Java primitive type, or one of the Java primitive's wrapper types, the code generators use the default converters. For more information on default converters see [the section called "Default primitive type converters"](#).

What is generated

As mentioned in [the section called "Specifying the converters"](#), using the **javaType** customization element triggers the generation of one adapter class for each customization of an XML Schema primitive type. The adapters are named in sequence using the pattern **AdapterN**. If you specify two primitive type customizations, the code generators create two adapter classes: **Adapter1** and **Adapter2**.

The code generated for an XML schema construct depends on whether the effected XML Schema construct is a globally defined element or is defined as part of a complex type.

When the XML Schema construct is a globally defined element, the object factory method generated for the type is modified from the default method as follows:

- The method is decorated with an **@XmlJavaTypeAdapter** annotation.
The annotation instructs the runtime which adapter class to use when processing instances of this element. The adapter class is specified as a class object.
- The default type is replaced by the class specified by the **javaType** element's **name** attribute.

[Example 38.11, "Customized Object Factory Method for a Global Element"](#) shows the object factory method for an element affected by the customization shown in [Example 38.7, "Global Primitive Type Customization"](#).

Example 38.11. Customized Object Factory Method for a Global Element

```
@XmlElementDecl(namespace = "http://widgetVendor.com/types/widgetTypes", name = "shorty")
@XmlJavaTypeAdapter(org.w3._2001.xmlschema.Adapter1.class)
public JAXBELEMENT<Integer> createShorty(Integer value) {
    return new JAXBELEMENT<Integer>(_Shorty_QNAME, Integer.class, null, value);
}
```

When the XML Schema construct is defined as part of a complex type, the generated Java property is modified as follows:

- The property is decorated with an **@XmlJavaTypeAdapter** annotation.
The annotation instructs the runtime which adapter class to use when processing instances of this element. The adapter class is specified as a class object.
- The property's **@XmlElement** includes a **type** property.

The value of the **type** property is the class object representing the generated object's default base type. In the case of XML Schema primitive types, the class is **String**.

- The property is decorated with an **@XmlSchemaType** annotation. The annotation identifies the XML Schema primitive type of the construct.
- The default type is replaced by the class specified by the **javaType** element's **name** attribute.

[Example 38.12, "Customized Complex Type"](#) shows the object factory method for an element affected by the customization shown in [Example 38.7, "Global Primitive Type Customization"](#).

Example 38.12. Customized Complex Type

```
public class NumInventory {

    @XmlElement(required = true, type = String.class) @XmlJavaTypeAdapter(Adapter1.class)
    @XmlSchemaType(name = "short") protected Integer numLeft;
    @XmlElement(required = true)
    protected String size;

    public Integer getNumLeft() {
        return numLeft;
    }

    public void setNumLeft(Integer value) {
        this.numLeft = value;
    }

    public String getSize() {
        return size;
    }

    public void setSize(String value) {
        this.size = value;
    }

}
```

Implementing converters

The Apache CXF runtime does not know how to convert XML primitive types to and from the Java class specified by the **javaType** element, except that it should call the methods specified by the **parseMethod** attribute and the **printMethod** attribute. You are responsible for providing implementations of the methods the runtime calls. The implemented methods must be capable of working with the lexical structures of the XML primitive type.

To simplify the implementation of the data conversion methods, Apache CXF provides the **javax.xml.bind.DatatypeConverter** class. This class provides methods for parsing and printing all of the XML Schema primitive types. The parse methods take string representations of the XML data and they return an instance of the default type defined in [Table 34.1, "XML Schema Primitive Type to Java Native Type Mapping"](#). The print methods take an instance of the default type and they return a string representation of the XML data.

The Java documentation for the **DatatypeConverter** class can be found at <https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/DatatypeConverter.html>.

Default primitive type converters

When specifying a Java primitive type, or one of the Java primitive type Wrapper classes, in the **javaType** element's **name** attribute, it is not necessary to specify values for the **parseMethod** attribute or the **printMethod** attribute. The Apache CXF runtime substitutes default converters if no values are provided.

The default data converters use the JAXB **DatatypeConverter** class to parse the XML data. The default converters will also provide any type casting necessary to make the conversion work.

38.3. GENERATING JAVA CLASSES FOR SIMPLE TYPES

Overview

By default, named simple types do not result in generated types unless they are enumerations. Elements defined using a simple type are mapped to properties of a Java primitive type.

There are instances when you need to have simple types generated into Java classes, such as is when you want to use type substitution.

To instruct the code generators to generate classes for all globally defined simple types, set the **globalBindings** customization element's **mapSimpleTypeDef** to **true**.

Adding the customization

To instruct the code generators to create Java classes for named simple types add the **globalBinding** element's **mapSimpleTypeDef** attribute and set its value to **true**.

[Example 38.13, “in-Line Customization to Force Generation of Java Classes for SimpleTypes”](#) shows an in-line customization that forces the code generator to generate Java classes for named simple types.

Example 38.13. in-Line Customization to Force Generation of Java Classes for SimpleTypes

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
       xmlns="http://www.w3.org/2001/XMLSchema"
       xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
       xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
       jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings mapSimpleTypeDef="true" />
    </appinfo>
  </annotation>
  ...
</schema>
```

[Example 38.14, “Binding File to Force Generation of Constants”](#) shows an external binding file that customizes the generation of simple types.

Example 38.14. Binding File to Force Generation of Constants

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    jaxb:version="2.0">
<jaxb:bindings schemaLocation="types.xsd">
    <jaxb:globalBindings mapSimpleTypeDef="true" />
    <jaxb:bindings>
        <jaxb:bindings>
```

**IMPORTANT**

This customization only affects **named** simple types that are defined in the **global** scope.

Generated classes

The class generated for a simple type has one property called **value**. The **value** property is of the Java type defined by the mappings in [Section 34.1, “Primitive Types”](#). The generated class has a getter and a setter for the **value** property.

[Example 38.16, “Customized Mapping of a Simple Type”](#) shows the Java class generated for the simple type defined in [Example 38.15, “Simple Type for Customized Mapping”](#).

Example 38.15. Simple Type for Customized Mapping

```
<simpleType name="simpleton">
    <restriction base="xsd:string">
        <maxLength value="10"/>
    </restriction>
</simpleType>
```

Example 38.16. Customized Mapping of a Simple Type

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "simpleton", propOrder = {"value"})
public class Simpleton {

    @XmlValue
    protected String value;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}
```

38.4. CUSTOMIZING ENUMERATION MAPPING

Overview

If you want enumerated types that are based on a schema type other than **xsd:string**, you must instruct the code generator to map it. You can also control the name of the generated enumeration constants.

The customization is done using the **jaxb:typesafeEnumClass** element along with one or more **jaxb:typesafeEnumMember** elements.

There might also be instances where the default settings for the code generator cannot create valid Java identifiers for all of the members of an enumeration. You can customize how the code generators handle this by using an attribute of the **globalBindings** customization.

Member name customizer

If the code generator encounters a naming collision when generating the members of an enumeration or if it cannot create a valid Java identifier for a member of the enumeration, the code generator, by default, generates a warning and does not generate a Java **enum** type for the enumeration.

You can alter this behavior by adding the **globalBinding** element's **typesafeEnumMemberName** attribute. The **typesafeEnumMemberName** attribute's values are described in [Table 38.2, "Values for Customizing Enumeration Member Name Generation"](#).

Table 38.2. Values for Customizing Enumeration Member Name Generation

Value	Description
skipGeneration (default)	Specifies that the Java enum type is not generated and generates a warning.
generateName	Specifies that member names will be generated following the pattern VALUE_N . <i>N</i> starts off at one, and is incremented for each member of the enumeration.
generateError	Specifies that the code generator generates an error when it cannot map an enumeration to a Java enum type.

[Example 38.17, "Customization to Force Type Safe Member Names"](#) shows an in-line customization that forces the code generator to generate type safe member names.

Example 38.17. Customization to Force Type Safe Member Names

```

<schema targetNamespace="http://widget.com/types/widgetTypes"
       xmlns="http://www.w3.org/2001/XMLSchema"
       xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
       xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
       jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings typesafeEnumMemberName="generateName" />
    </appinfo>
  </annotation>

```

```

</appinfo>
</annotation>
...
</schema>
```

Class customizer

The **jaxb:typesafeEnumClass** element specifies that an XML Schema enumeration should be mapped to a Java **enum** type. It has two attributes that are described in [Table 38.3, “Attributes for Customizing a Generated Enumeration Class”](#). When the **jaxb:typesafeEnumClass** element is specified in-line, it must be placed inside the **xsd:annotation** element of the simple type it is modifying.

Table 38.3. Attributes for Customizing a Generated Enumeration Class

Attribute	Description
name	Specifies the name of the generated Java enum type. This value must be a valid Java identifier.
map	Specifies if the enumeration should be mapped to a Java enum type. The default value is true .

Member customizer

The **jaxb:typesafeEnumMember** element specifies the mapping between an XML Schema **enumeration** facet and a Java **enum** type constant. You must use one **jaxb:typesafeEnumMember** element for each **enumeration** facet in the enumeration being customized.

When using in-line customization, this element can be used in one of two ways:

- It can be placed inside the **xsd:annotation** element of the **enumeration** facet it is modifying.
- They can all be placed as children of the **jaxb:typesafeEnumClass** element used to customize the enumeration.

The **jaxb:typesafeEnumMember** element has a **name** attribute that is required. The **name** attribute specifies the name of the generated Java **enum** type constant. Its value must be a valid Java identifier.

The **jaxb:typesafeEnumMember** element also has a **value** attribute. The **value** is used to associate the **enumeration** facet with the proper **jaxb:typesafeEnumMember** element. The value of the **value** attribute must match one of the values of an **enumeration** facets' **value** attribute. This attribute is required when you use an external binding specification for customizing the type generation, or when you group the **jaxb:typesafeEnumMember** elements as children of the **jaxb:typesafeEnumClass** element.

Examples

[Example 38.18, “In-line Customization of an Enumerated Type”](#) shows an enumerated type that uses in-line customization and has the enumeration's members customized separately.

Example 38.18. In-line Customization of an Enumerated Type

```

<schema targetNamespace="http://widget.com/types/widgetTypes"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    jaxb:version="2.0">
  <simpleType name="widgetInteger">
    <annotation>
      <appinfo>
        <jaxb:typesafeEnumClass />
      </appinfo>
    </annotation>
    <restriction base="xsd:int">
      <enumeration value="1">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="one" />
          </appinfo>
        </annotation>
      </enumeration>
      <enumeration value="2">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="two" />
          </appinfo>
        </annotation>
      </enumeration>
      <enumeration value="3">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="three" />
          </appinfo>
        </annotation>
      </enumeration>
      <enumeration value="4">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="four" />
          </appinfo>
        </annotation>
      </enumeration>
    </restriction>
  </simpleType>
</schema>

```

[Example 38.19, “In-line Customization of an Enumerated Type Using a Combined Mapping”](#) shows an enumerated type that uses in-line customization and combines the member’s customization in the class customization.

Example 38.19. In-line Customization of an Enumerated Type Using a Combined Mapping

```

<schema targetNamespace="http://widget.com/types/widgetTypes"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"

```

```

jaxb:version="2.0">
<simpleType name="widgetInteger">
<annotation>
<appinfo>
<jaxb:typesafeEnumClass>
<jaxb:typesafeEnumMember value="1" name="one" />
<jaxb:typesafeEnumMember value="2" name="two" />
<jaxb:typesafeEnumMember value="3" name="three" />
<jaxb:typesafeEnumMember value="4" name="four" />
</jaxb:typesafeEnumClass>
</appinfo>
</annotation>
<restriction base="xsd:int">
<enumeration value="1" />
<enumeration value="2" />
<enumeration value="3" />
<enumeration value="4" />
</restriction>
</simpleType>
<schema>

```

[Example 38.20, “Binding File for Customizing an Enumeration”](#) shows an external binding file that customizes an enumerated type.

Example 38.20. Binding File for Customizing an Enumeration

```

<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    jaxb:version="2.0">
<jaxb:bindings schemaLocation="enumMap.xsd">
<jaxb:bindings node="xsd:simpleType[@name='widgetInteger']">
<jaxb:typesafeEnumClass>
<jaxb:typesafeEnumMember value="1" name="one" />
<jaxb:typesafeEnumMember value="2" name="two" />
<jaxb:typesafeEnumMember value="3" name="three" />
<jaxb:typesafeEnumMember value="4" name="four" />
</jaxb:typesafeEnumClass>
</jaxb:bindings>
</jaxb:bindings>
<jaxb:bindings>

```

38.5. CUSTOMIZING FIXED VALUE ATTRIBUTE MAPPING

Overview

By default, the code generators map attributes defined as having a fixed value to normal properties. When using schema validation, Apache CXF can enforce the schema definition (see [Section 24.3.4.7, “Schema Validation Type Values”](#)). However, using schema validation increases message processing time.

Another way to map attributes that have fixed values to Java is to map them to Java constants. You can

instruct the code generator to map fixed value attributes to Java constants using the **globalBindings** customization element. You can also customize the mapping of fixed value attributes to Java constants at a more localized level using the **property** element.

Global customization

You can alter this behavior by adding the **globalBinding** element's **fixedAttributeAsConstantProperty** attribute. Setting this attribute to **true** instructs the code generator to map any attribute defined using **fixed** attribute to a Java constant.

[Example 38.21, "in-Line Customization to Force Generation of Constants"](#) shows an in-line customization that forces the code generator to generate constants for attributes with fixed values.

Example 38.21. in-Line Customization to Force Generation of Constants

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings fixedAttributeAsConstantProperty="true" />
    </appinfo>
  </annotation>
  ...
</schema>
```

[Example 38.22, "Binding File to Force Generation of Constants"](#) shows an external binding file that customizes the generation of fixed attributes.

Example 38.22. Binding File to Force Generation of Constants

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    jaxb:version="2.0">
  <jaxb:bindings schemaLocation="types.xsd">
    <jaxb:globalBindings fixedAttributeAsConstantProperty="true" />
    <jaxb:bindings>
    <jaxb:bindings>
```

Local mapping

You can customize attribute mapping on a per-attribute basis using the **property** element's **fixedAttributeAsConstantProperty** attribute. Setting this attribute to **true** instructs the code generator to map any attribute defined using **fixed** attribute to a Java constant.

[Example 38.23, "In-Line Customization to Force Generation of Constants"](#) shows an in-line customization that forces the code generator to generate constants for a single attribute with a fixed value.

Example 38.23. In-Line Customization to Force Generation of Constants

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    jaxb:version="2.0">
  <complexType name="widgetAttr">
    <sequence>
      ...
    </sequence>
    <attribute name="fixer" type="xsd:int" fixed="7">
      <annotation> <appinfo> <jaxb:property fixedAttributeAsConstantProperty="true" /> </appinfo>
    </annotation>
    </attribute>
  </complexType>
  ...
</schema>
```

[Example 38.24, “Binding File to Force Generation of Constants”](#) shows an external binding file that customizes the generation of a fixed attribute.

Example 38.24. Binding File to Force Generation of Constants

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    jaxb:version="2.0">
  <jaxb:bindings schemaLocation="types.xsd">
    <jaxb:bindings node="xsd:complexType[@name='widgetAttr']">
      <jaxb:bindings node="xsd:attribute[@name='fixer']">
        <jaxb:property fixedAttributeAsConstantProperty="true" />
      </jaxb:bindings>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

Java mapping

In the default mapping, all attributes are mapped to standard Java properties with getter and setter methods. When this customization is applied to an attribute defined using the **fixed** attribute, the attribute is mapped to a Java constant, as shown in [Example 38.25, “Mapping of a Fixed Value Attribute to a Java Constant”](#).

Example 38.25. Mapping of a Fixed Value Attribute to a Java Constant

```
@XmlAttribute
public final static type NAME = value;
```

type is determined by mapping the base type of the attribute to a Java type using the mappings described in [Section 34.1, “Primitive Types”](#).

NAME is determined by converting the value of the **attribute** element's **name** attribute to all capital letters.

value is determined by the value of the **attribute** element's **fixed** attribute.

For example, the attribute defined in [Example 38.23, "In-Line Customization to Force Generation of Constants"](#) is mapped as shown in [Example 38.26, "Fixed Value Attribute Mapped to a Java Constant"](#).

Example 38.26. Fixed Value Attribute Mapped to a Java Constant

```
@XmlRootElement(name = "widgetAttr")
public class WidgetAttr {

    ...
    @XmlAttribute
    public final static int FIXER = 7;
    ...
}
```

38.6. SPECIFYING THE BASE TYPE OF AN ELEMENT OR AN ATTRIBUTE

Overview

Occasionally you need to customize the class of the object generated for an element, or for an attribute defined as part of an XML Schema complex type. For example, you might want to use a more generalized class of object to allow for simple type substitution.

One way to do this is to use the JAXB base type customization. It allows a developer, on a case by case basis, to specify the class of object generated to represent an element or an attribute. The base type customization allows you to specify an alternate mapping between the XML Schema construct and the generated Java object. This alternate mapping can be a simple specialization or a generalization of the default base class. It can also be a mapping of an XML Schema primitive type to a Java class.

Customization usage

To apply the JAXB base type property to an XML Schema construct use the JAXB **baseType** customization element. The **baseType** customization element is a child of the JAXB **property** element, so it must be properly nested.

Depending on how you want to customize the mapping of the XML Schema construct to Java object, you add either the **baseType** customization element's **name** attribute, or a **javaType** child element. The **name** attribute is used to map the default class of the generated object to another class within the same class hierarchy. The **javaType** element is used when you want to map XML Schema primitive types to a Java class.



IMPORTANT

You cannot use both the **name** attribute and a **javaType** child element in the same **baseType** customization element.

Specializing or generalizing the default mapping

The **baseType** customization element's **name** attribute is used to redefine the class of the generated object to a class within the same Java class hierarchy. The attribute specifies the fully qualified name of the Java class to which the XML Schema construct is mapped. The specified Java class **must** be either a super-class or a sub-class of the Java class that the code generator normally generates for the XML Schema construct. For XML Schema primitive types that map to Java primitive types, the wrapper class is used as the default base class for the purpose of customization.

For example, an element defined as being of **xsd:int** uses **java.lang.Integer** as its default base class. The value of the **name** attribute can specify any super-class of **Integer** such as **Number** or **Object**.

For simple type substitution, the most common customization is to map the primitive types to an **Object** object.

[Example 38.27, "In-Line Customization of a Base Type"](#) shows an in-line customization that maps one element in a complex type to a Java **Object** object.

Example 38.27. In-Line Customization of a Base Type

```
<complexType name="widgetOrderInfo">
  <all>
    <element name="amount" type="xsd:int" />
    <element name="shippingAddress" type="Address">
      <annotation> <appinfo> <jaxb:property> <jaxb:baseType name="java.lang.Object" />
      </jaxb:property> </appinfo> </annotation>
    </element>
    <element name="type" type="xsd:string"/>
  </all>
</complexType>
```

[Example 38.28, "External Binding File to Customize a Base Type"](#) shows an external binding file for the customization shown in [Example 38.27, "In-Line Customization of a Base Type"](#).

Example 38.28. External Binding File to Customize a Base Type

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  jaxb:version="2.0">
  <jaxb:bindings schemaLocation="enumMap.xsd">
    <jaxb:bindings node="xsd:ComplexType[@name='widgetOrderInfo']">
      <jaxb:bindings node="xsd:element[@name='shippingAddress']">
        <jaxb:property>
          <jaxb:baseType name="java.lang.Object" />
        </jaxb:property>
      </jaxb:bindings>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

The resulting Java object's **@XmlElement** annotation includes a **type** property. The value of the **type** property is the class object representing the generated object's default base type. In the case of XML Schema primitive types, the class is the wrapper class of the corresponding Java primitive type.

[Example 38.29, "Java Class with a Modified Base Class"](#) shows the class generated based on the schema definition in [Example 38.28, "External Binding File to Customize a Base Type"](#).

Example 38.29. Java Class with a Modified Base Class

```
public class WidgetOrderInfo {  
  
    protected int amount;  
    @XmlElement(required = true)  
    protected String type;  
    @XmlElement(required = true, type = Address.class) protected Object shippingAddress;  
  
    ...  
    public Object getShippingAddress() {  
        return shippingAddress;  
    }  
  
    public void setShippingAddress(Object value) {  
        this.shippingAddress = value;  
    }  
}
```

Usage with **javaType**

The **javaType** element can be used to customize how elements and attributes defined using XML Schema primitive types are mapped to Java objects. Using the **javaType** element provides a lot more flexibility than simply using the **baseType** element's **name** attribute. The **javaType** element allows you to map a primitive type to any class of object.

For a detailed description of using the **javaType** element, see [Section 38.2, "Specifying the Java Class of an XML Schema Primitive"](#).

CHAPTER 39. USING A JAXBCONTEXT OBJECT

Abstract

The **JAXBContext** object allows the Apache CXF's runtime to transform data between XML elements and Java object. Application developers need to instantiate a **JAXBContext** object they want to use JAXB objects in message handlers and when implementing consumers that work with raw XML messages.

OVERVIEW

The **JAXBContext** object is a low-level object used by the runtime. It allows the runtime to convert between XML elements and their corresponding Java representations. An application developer generally does not need to work with **JAXBContext** objects. The marshaling and unmarshaling of XML data is typically handled by the transport and binding layers of a JAX-WS application.

However, there are instances when an application will need to manipulate the XML message content directly. In two of these instances:

- [Section 41.1, "Using XML in a Consumer"](#)
- [Chapter 43, *Writing Handlers*](#)

You will need instantiate a **JAXBContext** object using one of the two available **JAXBContext.newInstance()** methods.

BEST PRACTICES

JAXBContext objects are resource intensive to instantiate. It is recommended that an application create as few instances as possible. One way to do this is to create a single **JAXBContext** object that can manage all of the JAXB objects used by your application and share it among as many parts of your application as possible.

JAXBContext objects are thread safe.

GETTING A JAXBCONTEXT OBJECT USING AN OBJECT FACTORY

The **JAXBContext** class provides a **newInstance()** method, shown in [Example 39.1, "Getting a JAXB Context Using Classes"](#), that takes a list of classes that implement JAXB objects.

Example 39.1. Getting a JAXB Context Using Classes

```
static JAXBContext newInstance(Class... classesToBeBound) throws JAXBException
```

The returned **JAXBObject** object will be able to marshal and unmarshal data for the JAXB object implemented by the classes passed into the method. It will also be able to work with any classes that are statically referenced from any of the classes passed into the method.

While it is possible to pass the name of every JAXB class used by your application to the **newInstance()** method it is not efficient. A more efficient way to accomplish the same goal is to pass in the object factory, or object factories, generated for your application. The resulting **JAXBContext** object will be able to manage any JAXB classes the specified object factories can instantiate.

GETTING A JAXBCONTEXT OBJECT USING PACKAGE NAMES

The **JAXBContext** class provides a **newInstance()** method, shown in [Example 39.2, "Getting a JAXB Context Using Classes"](#), that takes a colon (:) separated list of package names. The specified packages should contain JAXB objects derived from XML Schema.

Example 39.2. Getting a JAXB Context Using Classes

```
static JAXBContext newInstance(String contextPath) throws JAXBException
```

The returned **JAXBContext** object will be able to marshal and unmarshal data for all of the JAXB objects implemented by the classes in the specified packages.

CHAPTER 40. DEVELOPING ASYNCHRONOUS APPLICATIONS

Abstract

JAX-WS provides an easy mechanism for accessing services asynchronously. The SEI can specify additional methods that can be used to access a service asynchronously. The Apache CXF code generators generate the extra methods for you. You simply add the business logic.

40.1. TYPES OF ASYNCHRONOUS INVOCATION

In addition to the usual synchronous mode of invocation, Apache CXF supports two forms of asynchronous invocation:

- Polling approach – To invoke the remote operation using the polling approach, you call a method that has no output parameters, but returns a **javax.xml.ws.Response** object. The **Response** object (which inherits from the javax.util.concurrency.Future interface) can be polled to check whether or not a response message has arrived.
- Callback approach – To invoke the remote operation using the callback approach, you call a method that takes a reference to a callback object (of **javax.xml.ws.AsyncHandler** type) as one of its parameters. When the response message arrives at the client, the runtime calls back on the **AsyncHandler** object, and gives it the contents of the response message.

40.2. WSDL FOR ASYNCHRONOUS EXAMPLES

[Example 40.1, "WSDL Contract for Asynchronous Example"](#) shows the WSDL contract that is used for the asynchronous examples. The contract defines a single interface, GreeterAsync, which contains a single operation, greetMeSometime.

Example 40.1. WSDL Contract for Asynchronous Example

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://apache.org/hello_world_async_soap_http"
  xmlns:x1="http://apache.org/hello_world_async_soap_http/types"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://apache.org/hello_world_async_soap_http"
  name="HelloWorld">

  <wsdl:types>
    <schema targetNamespace="http://apache.org/hello_world_async_soap_http/types"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:x1="http://apache.org/hello_world_async_soap_http/types"
      elementFormDefault="qualified">
      <element name="greetMeSometime">
        <complexType>
          <sequence>
            <element name="requestType" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </wsdl:types>
```

```

<element name="greetMeSometimeResponse">
  <complexType>
    <sequence>
      <element name="responseType"
              type="xsd:string"/>
    </sequence>
  </complexType>
</element>
</schema>
</wsdl:types>

<wsdl:message name="greetMeSometimeRequest">
  <wsdl:part name="in" element="x1:greetMeSometime"/>
</wsdl:message>
<wsdl:message name="greetMeSometimeResponse">
  <wsdl:part name="out"
              element="x1:greetMeSometimeResponse"/>
</wsdl:message>

<wsdl:portType name="GreeterAsync">
  <wsdl:operation name="greetMeSometime">
    <wsdl:input name="greetMeSometimeRequest"
                message="tns:greetMeSometimeRequest"/>
    <wsdl:output name="greetMeSometimeResponse"
                 message="tns:greetMeSometimeResponse"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="GreeterAsync_SOAPBinding"
               type="tns:GreeterAsync">
  ...
</wsdl:binding>

<wsdl:service name="SOAPService">
  <wsdl:port name="SoapPort"
             binding="tns:GreeterAsync_SOAPBinding">
    <soap:address location="http://localhost:9000/SapContext/SapPort"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

40.3. GENERATING THE STUB CODE

Overview

The asynchronous style of invocation requires extra stub code for the dedicated asynchronous methods defined on the SEI. This special stub code is not generated by default. To switch on the asynchronous feature and generate the requisite stub code, you must use the mapping customization feature from the WSDL 2.0 specification.

Customization enables you to modify the way the Maven code generation plug-in generates stub code. In particular, it enables you to modify the WSDL-to-Java mapping and to switch on certain features. Here, customization is used to switch on the asynchronous invocation feature. Customizations are

specified using a binding declaration, which you define using a **jaxws:bindings** tag (where the **jaxws** prefix is tied to the <http://java.sun.com/xml/ns/jaxws> namespace). There are two ways of specifying a binding declaration:

External Binding Declaration

When using an external binding declaration the **jaxws:bindings** element is defined in a file separate from the WSDL contract. You specify the location of the binding declaration file to code generator when you generate the stub code.

Embedded Binding Declaration

When using an embedded binding declaration you embed the **jaxws:bindings** element directly in a WSDL contract, treating it as a WSDL extension. In this case, the settings in **jaxws:bindings** apply only to the immediate parent element.

Using an external binding declaration

The template for a binding declaration file that switches on asynchronous invocations is shown in [Example 40.2, "Template for an Asynchronous Binding Declaration"](#).

Example 40.2. Template for an Asynchronous Binding Declaration

```
<bindings xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
          wsdlLocation="AffectedWSDL"
          xmlns="http://java.sun.com/xml/ns/jaxws">
  <bindings node="AffectedNode">
    <enableAsyncMapping>true</enableAsyncMapping>
  </bindings>
</bindings>
```

Where *AffectedWSDL* specifies the URL of the WSDL contract that is affected by this binding declaration. The *AffectedNode* is an XPath value that specifies which node (or nodes) from the WSDL contract are affected by this binding declaration. You can set *AffectedNode* to **wsdl:definitions**, if you want the entire WSDL contract to be affected. The **jaxws:enableAsyncMapping** element is set to **true** to enable the asynchronous invocation feature.

For example, if you want to generate asynchronous methods only for the GreeterAsync interface, you can specify `<bindings node="wsdl:definitions/wsdl:portType[@name='GreeterAsync']">` in the preceding binding declaration.

Assuming that the binding declaration is stored in a file, **async_binding.xml**, you would set up your POM as shown in [Example 40.3, "Consumer Code Generation"](#).

Example 40.3. Consumer Code Generation

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>${cxf.version}</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
```

```

<configuration>
    <sourceRoot>outputDir</sourceRoot>
    <wsdlOptions>
        <wsdlOption>
            <wsdl>hello_world.wsdl</wsdl>
            <extraargs>
                <extraarg>client</extraarg>
                <extraarg>-b async_binding.xml</extraarg>
            </extraargs>
        </wsdlOption>
    </wsdlOptions>
</configuration>
<goals>
    <goal>wsdl2java</goal>
</goals>
</execution>
</executions>
</plugin>

```

The **-b** option tells the code generator where to locate the external binding file.

For more information on the code generator see [Section 44.2, “cxfrs-codegen-plugin”](#).

Using an embedded binding declaration

You can also embed the binding customization directly into the WSDL document defining the service by placing the **jaxws:bindings** element and its associated **jaxws:enableAsyncMapping** child directly into the WSDL. You also must add a namespace declaration for the **jaxws** prefix.

[Example 40.4, “WSDL with Embedded Binding Declaration for Asynchronous Mapping”](#) shows a WSDL file with an embedded binding declaration that activates the asynchronous mapping for an operation.

Example 40.4. WSDL with Embedded Binding Declaration for Asynchronous Mapping

```

<wsdl:definitions xmlns="http://schemas.xmlsoap.org/wsdl/">
    ...
    xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
    ...
    ...
    <wsdl:portType name="GreeterAsync">
        <wsdl:operation name="greetMeSometime">
            <jaxws:bindings> <jaxws:enableAsyncMapping>true</jaxws:enableAsyncMapping>
        </jaxws:bindings>
        <wsdl:input name="greetMeSometimeRequest"
            message="tns:greetMeSometimeRequest"/>
        <wsdl:output name="greetMeSometimeResponse"
            message="tns:greetMeSometimeResponse"/>
    </wsdl:operation>
</wsdl:portType>
...
</wsdl:definitions>

```

When embedding the binding declaration into the WSDL document you can control the scope affected by the declaration by changing where you place the declaration. When the declaration is placed as a child of the **wsdl:definitions** element the code generator creates asynchronous methods for all of the operations defined in the WSDL document. If it is placed as a child of a **wsdl:portType** element the code generator creates asynchronous methods for all of the operations defined in the interface. If it is placed as a child of a **wsdl:operation** element the code generator creates asynchronous methods for only that operation.

It is not necessary to pass any special options to the code generator when using embedded declarations. The code generator will recognize them and act accordingly.

Generated interface

After generating the stub code in this way, the GreeterAsync SEI (in the file **GreeterAsync.java**) is defined as shown in [Example 40.5, "Service Endpoint Interface with Methods for Asynchronous Invocations"](#).

Example 40.5. Service Endpoint Interface with Methods for Asynchronous Invocations

```
package org.apache.hello_world_async_soap_http;

import org.apache.hello_world_async_soap_http.types.GreetMeSometimeResponse;
...

public interface GreeterAsync
{
    public Future<?> greetMeSometimeAsync(
        java.lang.String requestType,
        AsyncHandler<GreetMeSometimeResponse> asyncHandler
    );

    public Response<GreetMeSometimeResponse> greetMeSometimeAsync(
        java.lang.String requestType
    );

    public java.lang.String greetMeSometime(
        java.lang.String requestType
    );
}
```

In addition to the usual synchronous method, **greetMeSometime()**, two asynchronous methods are also generated for the **greetMeSometime** operation:

- Callback approach public **Future<?**
>greetMeSometimeAsyncjava.lang.StringrequestTypeAsyncHandler<GreetMeSometimeRes
ponse>asyncHandler
- Polling approach
public**Response<GreetMeSomeTimeResponse>greetMeSometimeAsyncjava.lang.Stringre**
questType

40.4. IMPLEMENTING AN ASYNCHRONOUS CLIENT WITH THE POLLING APPROACH

Overview

The polling approach is the more straightforward of the two approaches to developing an asynchronous application. The client invokes the asynchronous method called ***OperationNameAsync()*** and is returned a ***Response<T>*** object that it polls for a response. What the client does while it is waiting for a response is depends on the requirements of the application. There are two basic patterns for handling the polling:

- **Non-blocking polling** – You periodically check to see if the result is ready by calling the non-blocking ***Response<T>.isDone()*** method. If the result is ready, the client processes it. If it not, the client continues doing other things.
- **Blocking polling** – You call ***Response<T>.get()*** right away, and block until the response arrives (optionally specifying a timeout).

Using the non-blocking pattern

[Example 40.6, “Non-Blocking Polling Approach for an Asynchronous Operation Call”](#) illustrates using non-blocking polling to make an asynchronous invocation on the `greetMeSometime` operation defined in [Example 40.1, “WSDL Contract for Asynchronous Example”](#). The client invokes the asynchronous operation and periodically checks to see if the result is returned.

Example 40.6. Non-Blocking Polling Approach for an Asynchronous Operation Call

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client {
    private static final QName SERVICE_NAME
        = new QName("http://apache.org/hello_world_async_soap_http",
                    "SOAPService");

    private Client() {}

    public static void main(String args[]) throws Exception {
        // set up the proxy for the client

        Response<GreetMeSometimeResponse> greetMeSomeTimeResp =
            port.greetMeSometimeAsync(System.getProperty("user.name"));

        while (!greetMeSomeTimeResp.isDone()) {
            // client does some work
        }
        GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
        // process the response

        System.exit(0);
    }
}
```



The code in [Example 40.6, “Non-Blocking Polling Approach for an Asynchronous Operation Call”](#) does the following:

Invokes the **greetMeSometimeAsync()** on the proxy.

The method call returns the **Response<GreetMeSometimeResponse>** object to the client immediately. The Apache CXF runtime handles the details of receiving the reply from the remote endpoint and populating the **Response<GreetMeSometimeResponse>** object.



NOTE

The runtime transmits the request to the remote endpoint’s **greetMeSometime()** method and handles the details of the asynchronous nature of the call transparently. The endpoint, and therefore the service implementation, never worries about the details of how the client intends to wait for a response.

Checks to see if a response has arrived by checking the **isDone()** of the returned **Response** object.

If the response has not arrived, the client continues working before checking again.

When the response arrives, the client retrieves it from the **Response** object using the **get()** method.

Using the blocking pattern

When using the block polling pattern, the **Response** object’s **isDone()** is never called. Instead, the **Response** object’s **get()** method is called immediately after invoking the remote operation. The **get()** blocks until the response is available.

You can also pass a timeout limit to the **get()** method.

[Example 40.7, “Blocking Polling Approach for an Asynchronous Operation Call”](#) shows a client that uses blocking polling.

Example 40.7. Blocking Polling Approach for an Asynchronous Operation Call

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client {
    private static final QName SERVICE_NAME
        = new QName("http://apache.org/hello_world_async_soap_http",
                    "SOAPService");

    private Client() {}

    public static void main(String args[]) throws Exception {
```

```

// set up the proxy for the client

Response<GreetMeSometimeResponse> greetMeSomeTimeResp =
    port.greetMeSometimeAsync(System.getProperty("user.name"));
GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
// process the response
System.exit(0);
}
}

```

40.5. IMPLEMENTING AN ASYNCHRONOUS CLIENT WITH THE CALLBACK APPROACH

Overview

An alternative approach to making an asynchronous operation invocation is to implement a callback class. You then call the asynchronous remote method that takes the callback object as a parameter. The runtime returns the response to the callback object.

To implement an application that uses callbacks, do the following:

1. Create a callback class that implements the `AsyncHandler` interface.



NOTE

Your callback object can perform any amount of response processing required by your application.

2. Make remote invocations using the `operationNameAsync()` that takes the callback object as a parameter and returns a `Future<?>` object.
3. If your client requires access to the response data, you can poll the returned `Future<?>` object's `isDone()` method to see if the remote endpoint has sent the response.
If the callback object does all of the response processing, it is not necessary to check if the response has arrived.

Implementing the callback

The callback class must implement the `javax.xml.ws.AsyncHandler` interface. The interface defines a single method: `handleResponseResponse<T>res`. The Apache CXF runtime calls the `handleResponse()` method to notify the client that the response has arrived. [Example 40.8, "The `javax.xml.ws.AsyncHandler` Interface"](#) shows an outline of the `AsyncHandler` interface that you must implement.

Example 40.8. The `javax.xml.ws.AsyncHandler` Interface

```

public interface javax.xml.ws.AsyncHandler
{
    void handleResponse(Response<T> res)
}

```

[Example 40.9, “Callback Implementation Class”](#) shows a callback class for the greetMeSometime operation defined in [Example 40.1, “WSDL Contract for Asynchronous Example”](#).

Example 40.9. Callback Implementation Class

```
package demo.hw.client;

import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.types.*;

public class GreeterAsyncHandler implements AsyncHandler<GreetMeSometimeResponse>
{
    private GreetMeSometimeResponse reply;

    public void handleResponse(Response<GreetMeSometimeResponse>
                               response)
    {
        try
        {
            reply = response.get();
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }

    public String getResponse()
    {
        return reply.getResponse();
    }
}
```

The callback implementation shown in [Example 40.9, “Callback Implementation Class”](#) does the following:

Defines a member variable, **response**, that holds the response returned from the remote endpoint.

Implements **handleResponse()**.

This implementation simply extracts the response and assigns it to the member variable **reply**.

Implements an added method called **getResponseType()**.

This method is a convenience method that extracts the data from **reply** and returns it.

Implementing the consumer

[Example 40.10, “Callback Approach for an Asynchronous Operation Call”](#) illustrates a client that uses the callback approach to make an asynchronous call to the GreetMeSometime operation defined in [Example 40.1, “WSDL Contract for Asynchronous Example”](#).

Example 40.10. Callback Approach for an Asynchronous Operation Call

```

package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client {

    ...

    public static void main(String args[]) throws Exception
    {
        ...

        // Callback approach
        GreeterAsyncHandler callback = new GreeterAsyncHandler();

        Future<?> response =
            port.greetMeSometimeAsync(System.getProperty("user.name"),
                                      callback);
        while (!response.isDone())
        {
            // Do some work
        }
        resp = callback.getResponse();
        ...
        System.exit(0);
    }
}

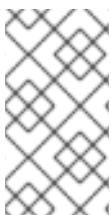
```

The code in [Example 40.10, “Callback Approach for an Asynchronous Operation Call”](#) does the following:

Instantiates a callback object.

Invokes the **greetMeSometimeAsync()** that takes the callback object on the proxy.

The method call returns the **Future<?>** object to the client immediately. The Apache CXF runtime handles the details of receiving the reply from the remote endpoint, invoking the callback object’s **handleResponse()** method, and populating the **Response<GreetMeSometimeResponse>** object.



NOTE

The runtime transmits the request to the remote endpoint’s **greetMeSometime()** method and handles the details of the asynchronous nature of the call without the remote endpoint’s knowledge. The endpoint, and therefore the service implementation, does not need to worry about the details of how the client intends to wait for a response.

Uses the returned **Future<?>** object’s **isDone()** method to check if the response has arrived from the remote endpoint.

Invokes the callback object's **getResponse()** method to get the response data.

40.6. CATCHING EXCEPTIONS RETURNED FROM A REMOTE SERVICE

Overview

Consumers making asynchronous requests will not receive the same exceptions returned when they make synchronous requests. Any exceptions returned to the consumer asynchronously are wrapped in an `ExecutionException` exception. The actual exception thrown by the service is stored in the `ExecutionException` exception's **cause** field.

Catching the exception

Exceptions generated by a remote service are thrown locally by the method that passes the response to the consumer's business logic. When the consumer makes a synchronous request, the method making the remote invocation throws the exception. When the consumer makes an asynchronous request, the `Response<T>` object's **get()** method throws the exception. The consumer will not discover that an error was encountered in processing the request until it attempts to retrieve the response message.

Unlike the methods generated by the JAX-WS framework, the `Response<T>` object's **get()** method throws neither user modeled exceptions nor generic JAX-WS exceptions. Instead, it throws a `java.util.concurrent.ExecutionException` exception.

Getting the exception details

The framework stores the exception returned from the remote service in the `ExecutionException` exception's **cause** field. The details about the remote exception are extracted by getting the value of the **cause** field and examining the stored exception. The stored exception can be any user defined exception or one of the generic JAX-WS exceptions.

Example

[Example 40.11, "Catching an Exception using the Polling Approach"](#) shows an example of catching an exception using the polling approach.

Example 40.11. Catching an Exception using the Polling Approach

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client
{
    private static final QName SERVICE_NAME
        = new QName("http://apache.org/hello_world_async_soap_http",
                    "SOAPService");

    private Client() {}

    public static void main(String[] args) throws Exception
    {
        Client client = new Client();
        Future<Response<String>> future = client.getPort().get();
        Response<String> response = future.get();
        System.out.println(response.getValue());
    }
}
```

```
public static void main(String args[]) throws Exception
{
    ...
    // port is a previously established proxy object.
    Response<GreetMeSometimeResponse> resp =
        port.greetMeSometimeAsync(System.getProperty("user.name"));

    while (!resp.isDone())
    {
        // client does some work
    }

    try
    {
        GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
        // process the response
    }
    catch (ExecutionException ee)
    {
        Throwable cause = ee.getCause();
        System.out.println("Exception "+cause.getClass().getName()+" thrown by the remote
service.");
    }
}
}
```

The code in [Example 40.11, “Catching an Exception using the Polling Approach”](#) does the following:

Wraps the call to the Response<T> object’s **get()** method in a try/catch block.

Catches a ExecutionException exception.

Extracts the **cause** field from the exception.

If the consumer was using the callback approach the code used to catch the exception would be placed in the callback object where the service’s response is extracted.

CHAPTER 41. USING RAW XML MESSAGES

Abstract

The high-level JAX-WS APIs shield the developer from using native XML messages by marshaling the data into JAXB objects. However, there are cases when it is better to have direct access to the raw XML message data that is passing on the wire. The JAX-WS APIs provide two interfaces that provide access to the raw XML: the Dispatch interface is the client-side interface, and the Provider interface is the server-side interface.

41.1. USING XML IN A CONSUMER

Abstract

The Dispatch interface is a low-level JAX-WS API that allows you work directly with raw messages. It accepts and returns messages, or payloads, of a number of types including DOM objects, SOAP messages, and JAXB objects. Because it is a low-level API, the Dispatch interface does not perform any of the message preparation that the higher-level JAX-WS APIs perform. You must ensure that the messages, or payloads, that you pass to the Dispatch object are properly constructed, and make sense for the remote operation being invoked.

41.1.1. Usage Modes

Overview

Dispatch objects have two *usage modes*:

- [Message mode](#)
- [Message Payload mode \(Payload mode\)](#)

The usage mode you specify for a Dispatch object determines the amount of detail that is passed to the user level code.

Message mode

In *message mode*, a Dispatch object works with complete messages. A complete message includes any binding specific headers and wrappers. For example, a consumer interacting with a service that requires SOAP messages must provide the Dispatch object's **invoke()** method a fully specified SOAP message. The **invoke()** method also returns a fully specified SOAP message. The consumer code is responsible for completing and reading the SOAP message's headers and the SOAP message's envelope information.

Message mode is not ideal when working with JAXB objects.

To specify that a Dispatch object uses message mode provide the value `java.xml.ws.Service.Mode.MESSAGE` when creating the Dispatch object. For more information about creating a Dispatch object see [the section called "Creating a Dispatch object"](#).

Payload mode

In *payload mode*, also called message payload mode, a Dispatch object works with only the payload of a message. For example, a Dispatch object working in payload mode works only with the body of a SOAP

message. The binding layer processes any binding level wrappers and headers. When a result is returned from the **invoke()** method the binding level wrappers and headers are already striped away, and only the body of the message is left.

When working with a binding that does not use special wrappers, such as the Apache CXF XML binding, payload mode and message mode provide the same results.

To specify that a Dispatch object uses payload mode provide the value `java.xml.ws.Service.Mode.PAYLOAD` when creating the Dispatch object. For more information about creating a Dispatch object see [the section called “Creating a Dispatch object”](#).

41.1.2. Data Types

Overview

Because Dispatch objects are low-level objects, they are not optimized for using the same JAXB generated types as the higher level consumer APIs. Dispatch objects work with the following types of objects:

- [javax.xml.transform.Source](#)
- [javax.xml.soap.SOAPMessage](#)
- [javax.activation.DataSource](#)
- [the section called “Using JAXB objects”](#)

Using Source objects

A Dispatch object accepts and returns objects that are derived from the `javax.xml.transform.Source` interface. Source objects are supported by any binding, and in either message mode or payload mode.

Source objects are low level objects that hold XML documents. Each Source implementation provides methods that access the stored XML documents and then manipulate its contents. The following objects implement the Source interface:

DOMSource

Holds XML messages as a Document Object Model(DOM) tree. The XML message is stored as a set of **Node** objects that are accessed using the **getNode()** method. Nodes can be either updated or added to the DOM tree using the **setNode()** method.

SAXSource

Holds XML messages as a Simple API for XML (SAX) object. SAX objects contain an **InputSource** object that holds the raw data and an **XMLReader** object that parses the raw data.

StreamSource

Holds XML messages as a data stream. The data stream can be manipulated the same as any other data stream.

If you create your Dispatch object so that it uses generic Source objects, Apache CXF returns the messages as **SAXSource** objects.

This behavior can be changed using the endpoint’s **source-preferred-format** property. See [Part IV, “Configuring Web Service Endpoints”](#) for information about configuring the Apache CXF runtime.

Using SOAPMessage objects

Dispatch objects can use **javax.xml.soap.SOAPMessage** objects when the following conditions are true:

- The Dispatch object is using the SOAP binding
- The Dispatch object is using message mode

A **SOAPMessage** object holds a SOAP message. They contain one **SOAPPart** object and zero or more **AttachmentPart** objects. The **SOAPPart** object contains the SOAP specific portions of the SOAP message including the SOAP envelope, any SOAP headers, and the SOAP message body. The **AttachmentPart** objects contain binary data that is passed as an attachment.

Using DataSource objects

Dispatch objects can use objects that implement the `javax.activation.DataSource` interface when the following conditions are true:

- The Dispatch object is using the HTTP binding
- The Dispatch object is using message mode

DataSource objects provide a mechanism for working with MIME typed data from a variety of sources, including URLs, files, and byte arrays.

Using JAXB objects

While Dispatch objects are intended to be low level APIs that allow you to work with raw messages, they also allow you to work with JAXB objects. To work with JAXB objects a Dispatch object must be passed a **JAXBContext** that can marshal and unmarshal the JAXB objects in use. The **JAXBContext** is passed when the **Dispatch** object is created.

You can pass any JAXB object understood by the **JAXBContext** object as the parameter to the **invoke()** method. You can also cast the returned message into any JAXB object understood by the **JAXBContext** object.

For information on creating a **JAXBContext** object see [Chapter 39, Using A **JAXBContext** Object](#).

41.1.3. Working with Dispatch Objects

Procedure

To use a Dispatch object to invoke a remote service the following sequence should be followed:

1. [Create](#) a Dispatch object.
2. [Construct](#) a request message.
3. Call the proper **invoke()** method.
4. Parse the response message.

Creating a Dispatch object

To create a Dispatch object do the following:

1. Create a **Service** object to represent the **wsdl:service** element that defines the service on which the Dispatch object will make invocations. See [Section 25.2, “Creating a Service Object”](#).
2. Create the Dispatch object using the **Service** object’s **createDispatch()** method, shown in [Example 41.1, “The **createDispatch\(\)** Method”](#).

Example 41.1. The **createDispatch()** Method

```
public Dispatch<T> createDispatch(QName portName, java.lang.Class<T> type, Service.Mode mode) throws WebServiceException
```



NOTE

If you are using JAXB objects the method signature for **createDispatch()** is:

```
public Dispatch<T> createDispatch(QName portName, javax.xml.bind.JAXBContext context, Service.Mode mode) throws WebServiceException
```

[Table 41.1, “Parameters for **createDispatch\(\)**”](#) describes the parameters for the **createDispatch()** method.

[Table 41.1. Parameters for **createDispatch\(\)**](#)

Parameter	Description
portName	Specifies the QName of the wsdl:port element that represents the service provider where the Dispatch object will make invocations.
type	Specifies the data type of the objects used by the Dispatch object. See Section 41.1.2, “Data Types” . When working with JAXB objects, this parameter specifies the JAXBContext object used to marshal and unmarshal the JAXB objects.
mode	Specifies the usage mode for the Dispatch object. See Section 41.1.1, “Usage Modes” .

[Example 41.2, “Creating a Dispatch Object”](#) shows the code for creating a Dispatch object that works with **DOMSource** objects in payload mode.

Example 41.2. Creating a Dispatch Object

```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
```

```

public static void main(String args[])
{
    QName serviceName = new QName("http://org.apache.cxf", "stockQuoteReporter");
    Service s = Service.create(serviceName);

    QName portName = new QName("http://org.apache.cxf", "stockQuoteReporterPort");
    Dispatch<DOMSource> dispatch = s.createDispatch(portName,
                                                    DOMSource.class,
                                                    Service.Mode.PAYLOAD);
    ...
}

```

Constructing request messages

When working with Dispatch objects, requests must be built from scratch. The developer is responsible for ensuring that the messages passed to a Dispatch object match a request that the targeted service provider can process. This requires precise knowledge about the messages used by the service provider and what, if any, header information it requires.

This information can be provided by a WSDL document or an XML Schema document that defines the messages. While service providers vary greatly there are a few guidelines to be followed:

- The root element of the request is based in the value of the **name** attribute of the **wsdl:operation** element corresponding to the operation being invoked.



WARNING

If the service being invoked uses doc/literal bare messages, the root element of the request is based on the value of the **name** attribute of the **wsdl:part** element referred to by the **wsdl:operation** element.

- The root element of the request is namespace qualified.
- If the service being invoked uses rpc/literal messages, the top-level elements in the request will not be namespace qualified.



IMPORTANT

The children of top-level elements may be namespace qualified. To be certain you must check their schema definitions.

- If the service being invoked uses rpc/literal messages, none of the top-level elements can be null.
- If the service being invoked uses doc/literal messages, the schema definition of the message determines if any of the elements are namespace qualified.

For more information about how services use XML messages see, the [WS-I Basic Profile](#).

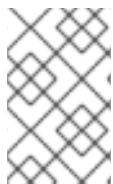
Synchronous invocation

For consumers that make synchronous invocations that generate a response, use the Dispatch object's **invoke()** method shown in [Example 41.3, "The Dispatch.invoke\(\) Method"](#).

Example 41.3. The Dispatch.invoke() Method

TinvokeTmsgWebServiceException

The type of both the response and the request passed to the **invoke()** method are determined when the Dispatch object is created. For example if you create a Dispatch object using **createDispatch(portName, SOAPMessage.class, Service.Mode.MESSAGE)**, both the response and the request are **SOAPMessage** objects.



NOTE

When using JAXB objects, both the response and the request can be of any type the provided **JAXBContext** object can marshal and unmarshal. Also, the response and the request can be different JAXB objects.

[Example 41.4, "Making a Synchronous Invocation Using a Dispatch Object"](#) shows code for making a synchronous invocation on a remote service using a **DOMSource** object.

Example 41.4. Making a Synchronous Invocation Using a Dispatch Object

```
// Creating a DOMSource Object for the request
DocumentBuilder db = DocumentBuilderFactory.newInstance().newDocumentBuilder();
Document requestDoc = db.newDocument();
Element root = requestDoc.createElementNS("http://org.apache.cxf/stockExample",
                                           "getStockPrice");
root.setText("DOW");
DOMSource request = new DOMSource(requestDoc);

// Dispatch disp created previously
DOMSource response = disp.invoke(request);
```

Asynchronous invocation

Dispatch objects also support asynchronous invocations. As with the higher level asynchronous APIs discussed in [Chapter 40, Developing Asynchronous Applications](#), Dispatch objects can use both the polling approach and the callback approach.

When using the polling approach, the **invokeAsync()** method returns a **Response<t>** object that can be polled to see if the response has arrived. [Example 41.5, "The Dispatch.invokeAsync\(\) Method for Polling"](#) shows the signature of the method used to make an asynchronous invocation using the polling approach.

Example 41.5. The Dispatch.invokeAsync() Method for Polling

Response <T>invokeAsyncTmsgWebServiceException

For detailed information on using the polling approach for asynchronous invocations see [Section 40.4, "Implementing an Asynchronous Client with the Polling Approach"](#).

When using the callback approach, the **invokeAsync()** method takes an AsyncHandler implementation that processes the response when it is returned. [Example 41.6, "The Dispatch.invokeAsync\(\) Method Using a Callback"](#) shows the signature of the method used to make an asynchronous invocation using the callback approach.

Example 41.6. The Dispatch.invokeAsync() Method Using a Callback

```
Future<?>invokeAsyncTmsgAsyncHandler<T>handlerWebServiceException
```

For detailed information on using the callback approach for asynchronous invocations see [Section 40.5, "Implementing an Asynchronous Client with the Callback Approach"](#).



NOTE

As with the synchronous **invoke()** method, the type of the response and the type of the request are determined when you create the Dispatch object.

Oneway invocation

When a request does not generate a response, make remote invocations using the Dispatch object's **invokeOneWay()**. [Example 41.7, "The Dispatch.invokeOneWay\(\) Method"](#) shows the signature for this method.

Example 41.7. The Dispatch.invokeOneWay() Method

```
invokeOneWayTmsgWebServiceException
```



NOTE

When using JAXB objects, the response and the request can be of any type the provided **JAXBContext** object can marshal and unmarshal.

[Example 41.8, "Making a One Way Invocation Using a Dispatch Object"](#) shows code for making a oneway invocation on a remote service using a JAXB object.

Example 41.8. Making a One Way Invocation Using a Dispatch Object

```
// Creating a JAXBContext and an Unmarshaller for the request
JAXBContext jbc = JAXBContext.newInstance("org.apache.cxf.StockExample");
Unmarshaller u = jbc.createUnmarshaller();

// Read the request from disk
File rf = new File("request.xml");
GetStockPrice request = (GetStockPrice)u.unmarshal(rf);
```

```
// Dispatch disp created previously
disp.invokeOneWay(request);
```

41.2. USING XML IN A SERVICE PROVIDER

Abstract

The Provider interface is a low-level JAX-WS API that allows you to implement a service provider that works directly with messages as raw XML. The messages are not packaged into JAXB objects before being passed to an object that implements the Provider interface.

41.2.1. Messaging Modes

Overview

Objects that implement the Provider interface have two *messaging modes*:

- [Message](#) mode
- [Payload](#) mode

The messaging mode you specify determines the level of messaging detail that is passed to your implementation.

Message mode

When using *message mode*, a Provider implementation works with complete messages. A complete message includes any binding specific headers and wrappers. For example, a Provider implementation that uses a SOAP binding receives requests as fully specified SOAP message. Any response returned from the implementation must be a fully specified SOAP message.

To specify that a Provider implementation uses message mode by provide the value `java.xml.ws.Service.Mode.MESSAGE` as the value to the **`javax.xml.ws.ServiceMode`** annotation, as shown in [Example 41.9, "Specifying that a Provider Implementation Uses Message Mode"](#) .

Example 41.9. Specifying that a Provider Implementation Uses Message Mode

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.MESSAGE)
public class stockQuoteProvider implements Provider<SOAPMessage>
{
    ...
}
```

Payload mode

In *payload mode* a Provider implementation works with only the payload of a message. For example, a Provider implementation working in payload mode works only with the body of a SOAP message. The binding layer processes any binding level wrappers and headers.

When working with a binding that does not use special wrappers, such as the Apache CXF XML binding, payload mode and message mode provide the same results.

To specify that a Provider implementation uses payload mode by provide the value `java.xml.ws.Service.Mode.PAYLOAD` as the value to the **`javax.xml.ws.ServiceMode`** annotation, as shown in [Example 41.10, “Specifying that a Provider Implementation Uses Payload Mode”](#).

Example 41.10. Specifying that a Provider Implementation Uses Payload Mode

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.PAYLOAD)
public class stockQuoteProvider implements Provider<DOMSource>
{
    ...
}
```

If you do not provide a value for the **`@ServiceMode`** annotation, the Provider implementation uses payload mode.

41.2.2. Data Types

Overview

Because they are low-level objects, Provider implementations cannot use the same JAXB generated types as the higher level consumer APIs. Provider implementations work with the following types of objects:

- [javax.xml.transform.Source](#)
- [javax.xml.soap.SOAPMessage](#)
- [javax.activation.DataSource](#)

Using Source objects

A Provider implementation can accept and return objects that are derived from the `javax.xml.transform.Source` interface. Source objects are low level objects that hold XML documents. Each Source implementation provides methods that access the stored XML documents and manipulate its contents. The following objects implement the Source interface:

DOMSource

Holds XML messages as a Document Object Model(DOM) tree. The XML message is stored as a set of **Node** objects that are accessed using the **`getNode()`** method. Nodes can be either updated or added to the DOM tree using the **`setNode()`** method.

SAXSource

Holds XML messages as a Simple API for XML (SAX) object. SAX objects contain an **`InputSource`** object that holds the raw data and an **`XMLReader`** object that parses the raw data.

StreamSource

Holds XML messages as a data stream. The data stream can be manipulated the same as any other data stream.

If you create your Provider object so that it uses generic Source objects, Apache CXF returns the messages as **SAXSource** objects.

This behavior can be changed using the endpoint's **source-preferred-format** property. See [Part IV, "Configuring Web Service Endpoints"](#) for information about configuring the Apache CXF runtime.



IMPORTANT

When using Source objects the developer is responsible for ensuring that all required binding specific wrappers are added to the message. For example, when interacting with a service expecting SOAP messages, the developer must ensure that the required SOAP envelope is added to the outgoing request and that the SOAP envelope's contents are correct.

Using SOAPMessage objects

Provider implementations can use **javax.xml.soap.SOAPMessage** objects when the following conditions are true:

- The Provider implementation is using the SOAP binding
- The Provider implementation is using message mode

A **SOAPMessage** object holds a SOAP message. They contain one **SOAPPart** object and zero or more **AttachmentPart** objects. The **SOAPPart** object contains the SOAP specific portions of the SOAP message including the SOAP envelope, any SOAP headers, and the SOAP message body. The **AttachmentPart** objects contain binary data that is passed as an attachment.

Using DataSource objects

Provider implementations can use objects that implement the `javax.activation.DataSource` interface when the following conditions are true:

- The implementation is using the HTTP binding
- The implementation is using message mode

DataSource objects provide a mechanism for working with MIME typed data from a variety of sources, including URLs, files, and byte arrays.

41.2.3. Implementing a Provider Object

Overview

The Provider interface is relatively easy to implement. It only has one method, **invoke()**, that must be implemented. In addition it has three simple requirements:

- An implementation must have the **@WebServiceProvider** annotation.
- An implementation must have a default public constructor.
- An implementation must implement a typed version of the Provider interface.
In other words, you cannot implement a `Provider<T>` interface. You must implement a version of the interface that uses a concrete data type as listed in [Section 41.2.2, "Data Types"](#). For example, you can implement an instance of a `Provider<SAXSource>`.

The complexity of implementing the Provider interface is in the logic handling the request messages and building the proper responses.

Working with messages

Unlike the higher-level SEI based service implementations, Provider implementations receive requests as raw XML data, and must send responses as raw XML data. This requires that the developer has intimate knowledge of the messages used by the service being implemented. These details can typically be found in the WSDL document describing the service.

[WS-I Basic Profile](#) provides guidelines about the messages used by services, including:

- The root element of a request is based in the value of the **name** attribute of the **wsdl:operation** element that corresponds to the operation that is invoked.



WARNING

If the service uses doc/literal bare messages, the root element of the request is based on the value of **name** attribute of the **wsdl:part** element referred to by the **wsdl:operation** element.

- The root element of all messages is namespace qualified.
- If the service uses rpc/literal messages, the top-level elements in the messages are not namespace qualified.



IMPORTANT

The children of top-level elements might be namespace qualified, but to be certain you will must check their schema definitions.

- If the service uses rpc/literal messages, none of the top-level elements can be null.
- If the service uses doc/literal messages, then the schema definition of the message determines if any of the elements are namespace qualified.

The **@WebServiceProvider** annotation

To be recognized by JAX-WS as a service implementation, a Provider implementation must be decorated with the **@WebServiceProvider** annotation.

[Table 41.2, “@WebServiceProvider Properties”](#) describes the properties that can be set for the **@WebServiceProvider** annotation.

Table 41.2. @WebServiceProvider Properties

Property	Description
----------	-------------

Property	Description
portName	Specifies the value of the name attribute of the wsdl:port element that defines the service's endpoint.
serviceName	Specifies the value of the name attribute of the wsdl:service element that contains the service's endpoint.
targetNamespace	Specifies the targetname space of the service's WSDL definition.
wsdlLocation	Specifies the URI for the WSDL document defining the service.

All of these properties are optional, and are empty by default. If you leave them empty, Apache CXF creates values using information from the implementation class.

Implementing the `invoke()` method

The Provider interface has only one method, **invoke()**, that must be implemented. The **invoke()** method receives the incoming request packaged into the type of object declared by the type of Provider interface being implemented, and returns the response message packaged into the same type of object. For example, an implementation of a `Provider<SOAPMessage>` interface receives the request as a **SOAPMessage** object and returns the response as a **SOAPMessage** object.

The messaging mode used by the Provider implementation determines the amount of binding specific information the request and the response messages contain. Implementations using message mode receive all of the binding specific wrappers and headers along with the request. They must also add all of the binding specific wrappers and headers to the response message. Implementations using payload mode only receive the body of the request. The XML document returned by an implementation using payload mode is placed into the body of the request message.

Examples

[Example 41.11, “Provider<SOAPMessage> Implementation”](#) shows a Provider implementation that works with **SOAPMessage** objects in message mode.

Example 41.11. `Provider<SOAPMessage>` Implementation

```
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

@WebServiceProvider(portName="stockQuoteReporterPort"
                  serviceName="stockQuoteReporter")
@ServiceMode(value="Service.Mode.MESSAGE")
public class stockQuoteReporterProvider implements Provider<SOAPMessage>
```

```

{
public stockQuoteReporterProvider()
{
}

public SOAPMessage invoke(SOAPMessage request)
{
    SOAPBody requestBody = request.getSOAPBody();
    if(requestBody.getElementName.getLocalName.equals("getStockPrice"))
    {
        MessageFactory mf = MessageFactory.newInstance();
        SOAPFactory sf = SOAPFactory.newInstance();

        SOAPMessage response = mf.createMessage();
        SOAPBody respBody = response.getSOAPBody();
        Name bodyName = sf.createName("getStockPriceResponse");
        respBody.addBodyElement(bodyName);
        SOAPElement respContent = respBody.addChildElement("price");
        respContent.setValue("123.00");
        response.saveChanges();
        return response;
    }
    ...
}
}

```

The code in [Example 41.11, “Provider<SOAPMessage> Implementation”](#) does the following:

Specifies that the following class implements a Provider object that implements the service whose **wsdl:service** element is named **stockQuoteReporter**, and whose **wsdl:port** element is named **stockQuoteReporterPort**.

Specifies that this Provider implementation uses message mode.

Provides the required default public constructor.

Provides an implementation of the **invoke()** method that takes a **SOAPMessage** object and returns a **SOAPMessage** object.

Extracts the request message from the body of the incoming SOAP message.

Checks the root element of the request message to determine how to process the request.

Creates the factories required for building the response.

Builds the SOAP message for the response.

Returns the response as a **SOAPMessage** object.

[Example 41.12, “Provider<DOMSource> Implementation”](#) shows an example of a Provider implementation using **DOMSource** objects in payload mode.

Example 41.12. Provider<DOMSource> Implementation

```
import javax.xml.ws.Provider;
```

```
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

@WebServiceProvider(portName="stockQuoteReporterPort"
serviceName="stockQuoteReporter")
@ServiceMode(value="Service.Mode.PAYLOAD")
public class stockQuoteReporterProvider implements Provider<DOMSource>
public stockQuoteReporterProvider()
{
}

public DOMSource invoke(DOMSource request)
{
    DOMSource response = new DOMSource();
    ...
    return response;
}
```

The code in [Example 41.12, “Provider<DOMSource> Implementation”](#) does the following:

Specifies that the class implements a Provider object that implements the service whose **wsdl:service** element is named **stockQuoteReporter**, and whose **wsdl:port** element is named **stockQuoteReporterPort**.

Specifies that this Provider implementation uses payload mode.

Provides the required default public constructor.

Provides an implementation of the **invoke()** method that takes a **DOMSource** object and returns a **DOMSource** object.

CHAPTER 42. WORKING WITH CONTEXTS

Abstract

JAX-WS uses contexts to pass metadata along the messaging chain. This metadata, depending on its scope, is accessible to implementation level code. It is also accessible to JAX-WS handlers that operate on the message below the implementation level.

42.1. UNDERSTANDING CONTEXTS

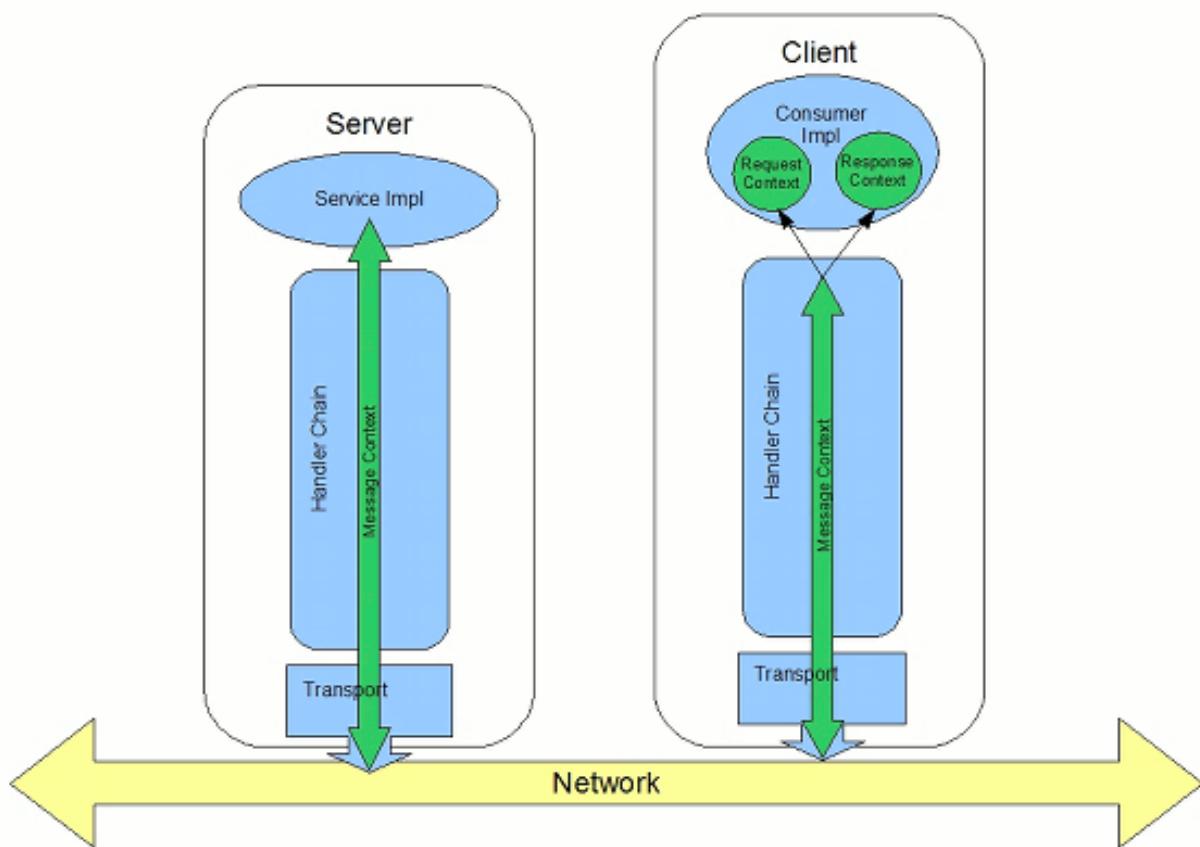
Overview

In many instances it is necessary to pass information about a message to other parts of an application. Apache CXF does this using a context mechanism. Contexts are maps that hold properties relating to an outgoing or an incoming message. The properties stored in the context are typically metadata about the message, and the underlying transport used to communicate the message. For example, the transport specific headers used in transmitting the message, such as the HTTP response code or the JMS correlation ID, are stored in the JAX-WS contexts.

The contexts are available at all levels of a JAX-WS application. However, they differ in subtle ways depending upon where in the message processing stack you are accessing the context. JAX-WS Handler implementations have direct access to the contexts and can access all properties that are set in them. Service implementations access contexts by having them injected, and can only access properties that are set in the **APPLICATION** scope. Consumer implementations can only access properties that are set in the **APPLICATION** scope.

[Figure 42.1, “Message Contexts and Message Processing Path”](#) shows how the context properties pass through Apache CXF. As a message passes through the messaging chain, its associated message context passes along with it.

Figure 42.1. Message Contexts and Message Processing Path



How properties are stored in a context

The message contexts are all implementations of the `javax.xml.ws.handler.MessageContext` interface. The `MessageContext` interface extends the `java.util.Map<String key, Object value>` interface. Map objects store information as key value pairs.

In a message context, properties are stored as name/value pairs. A property's key is a **String** that identifies the property. The value of a property can be any value stored in any Java object. When the value is returned from a message context, the application must know the type to expect and cast accordingly. For example, if a property's value is stored in a **UserInfo** object it is still returned from a message context as an **Object** object that must be cast back into a **UserInfo** object.

Properties in a message context also have a scope. The scope determines where a property can be accessed in the message processing chain.

Property scopes

Properties in a message context are scoped. A property can be in one of the following scopes:

APPLICATION

Properties scoped as **APPLICATION** are available to JAX-WS Handler implementations, consumer implementation code, and service provider implementation code. If a handler needs to pass a property to the service provider implementation, it sets the property's scope to **APPLICATION**. All properties set from either the consumer implementation or the service provider implementation contexts are automatically scoped as **APPLICATION**.

HANDLER

Properties scoped as **HANDLER** are only available to JAX-WS Handler implementations. Properties stored in a message context from a Handler implementation are scoped as **HANDLER** by default.

You can change a property's scope using the message context's **setScope()** method. [Example 42.1, "The MessageContext.setScope\(\) Method"](#) shows the method's signature.

Example 42.1. The MessageContext.setScope() Method

```
setScopeStringkeyMessageContext.Scopescopejava.lang.IllegalArgumentException
```

The first parameter specifies the property's key. The second parameter specifies the new scope for the property. The scope can be either:

- **MessageContext.Scope.APPLICATION**
- **MessageContext.Scope.HANDLER**

Overview of contexts in handlers

Classes that implement the JAX-WS Handler interface have direct access to a message's context information. The message's context information is passed into the Handler implementation's **handleMessage()**, **handleFault()**, and **close()** methods.

Handler implementations have access to all of the properties stored in the message context, regardless of their scope. In addition, logical handlers use a specialized message context called a **LogicalMessageContext**. **LogicalMessageContext** objects have methods that access the contents of the message body.

Overview of contexts in service implementations

Service implementations can access properties scoped as **APPLICATION** from the message context. The service provider's implementation object accesses the message context through the **WebServiceContext** object.

For more information see [Section 42.2, "Working with Contexts in a Service Implementation"](#).

Overview of contexts in consumer implementations

Consumer implementations have indirect access to the contents of the message context. The consumer implementation has two separate message contexts:

- Request context – holds a copy of the properties used for outgoing requests
- Response context – holds a copy of the properties from an incoming response

The dispatch layer transfers the properties between the consumer implementation's message contexts and the message context used by the Handler implementations.

When a request is passed to the dispatch layer from the consumer implementation, the contents of the request context are copied into the message context that is used by the dispatch layer. When the response is returned from the service, the dispatch layer processes the message and sets the

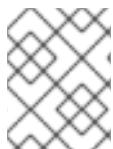
appropriate properties into its message context. After the dispatch layer processes a response, it copies all of the properties scoped as **APPLICATION** in its message context to the consumer implementation's response context.

For more information see [Section 42.3, "Working with Contexts in a Consumer Implementation"](#).

42.2. WORKING WITH CONTEXTS IN A SERVICE IMPLEMENTATION

Overview

Context information is made available to service implementations using the WebServiceContext interface. From the WebServiceContext object you can obtain a **MessageContext** object that is populated with the current request's context properties in the application scope. You can manipulate the values of the properties, and they are propagated back through the response chain.



NOTE

The MessageContext interface inherits from the java.util.Map interface. Its contents can be manipulated using the Map interface's methods.

Obtaining a context

To obtain the message context in a service implementation do the following:

1. Declare a variable of type WebServiceContext.
2. Decorate the variable with the **javax.annotation.Resource** annotation to indicate that the context information is being injected into the variable.
3. Obtain the **MessageContext** object from the WebServiceContext object using the **getMessageContext()** method.



IMPORTANT

getMessageContext() can only be used in methods that are decorated with the **@WebMethod** annotation.

[Example 42.2, "Obtaining a Context Object in a Service Implementation"](#) shows code for obtaining a context object.

Example 42.2. Obtaining a Context Object in a Service Implementation

```
import javax.xml.ws.*;
import javax.xml.ws.handler.*;
import javax.annotation.*;

@WebServiceProvider
public class WidgetServiceImpl
{
    @Resource
    WebServiceContext wsc;

    @WebMethod
}
```

```

public String getColor(String itemNum)
{
    MessageContext context = wsc.getMessageContext();
}

...
}

```

Reading a property from a context

Once you have obtained the `MessageContext` object for your implementation, you can access the properties stored there using the `get()` method shown in [Example 42.3, "The `MessageContext.get\(\)` Method"](#).

Example 42.3. The `MessageContext.get()` Method

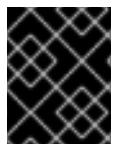
`getObject(key)`



NOTE

This `get()` is inherited from the `Map` interface.

The `key` parameter is the string representing the property you want to retrieve from the context. The `get()` returns an object that must be cast to the proper type for the property. [Table 42.1, "Properties Available in the Service Implementation Context"](#) lists a number of the properties that are available in a service implementation's context.



IMPORTANT

Changing the values of the object returned from the context also changes the value of the property in the context.

[Example 42.4, "Getting a Property from a Service's Message Context"](#) shows code for getting the name of the WSDL `operation` element that represents the invoked operation.

Example 42.4. Getting a Property from a Service's Message Context

```

import javax.xml.ws.handler.MessageContext;
import org.apache.cxf.message.Message;

...
// MessageContext context retrieved in a previous example
QName wsdl_operation = (QName)context.get(Message.WSDL_OPERATION);

```

Setting properties in a context

Once you have obtained the `MessageContext` object for your implementation, you can set properties, and change existing properties, using the `put()` method shown in [Example 42.5, "The `MessageContext.put\(\)` Method"](#).

Example 42.5. The MessageContext.put() Method

```
VputKkeyVvalueClassCastExceptionIllegalArgumentExceptionNullPointerException
```

If the property being set already exists in the message context, the **put()** method replaces the existing value with the new value and returns the old value. If the property does not already exist in the message context, the **put()** method sets the property and returns **null**.

[Example 42.6, “Setting a Property in a Service’s Message Context”](#) shows code for setting the response code for an HTTP request.

Example 42.6. Setting a Property in a Service’s Message Context

```
import javax.xml.ws.handler.MessageContext;
import org.apache.cxf.message.Message;

...
// MessageContext context retrieved in a previous example
context.put(Message.RESPONSE_CODE, new Integer(404));
```

Supported contexts

[Table 42.1, “Properties Available in the Service Implementation Context”](#) lists the properties accessible through the context in a service implementation object.

Table 42.1. Properties Available in the Service Implementation Context

Property Name	Description
org.apache.cxf.message.Message	
PROTOCOL_HEADERS ^[a]	Specifies the transport specific header information. The value is stored as a java.util.Map<String, List<String>> .
RESPONSE_CODE	Specifies the response code returned to the consumer. The value is stored as an Integer object.
ENDPOINT_ADDRESS	Specifies the address of the service provider. The value is stored as a String .
HTTP_REQUEST_METHOD	Specifies the HTTP verb sent with a request. The value is stored as a String .

Property Name	Description
PATH_INFO	<p>Specifies the path of the resource being requested. The value is stored as a String.</p> <p>The path is the portion of the URI after the hostname and before any query string. For example, if an endpoint's URI is http://cxf.apache.org/demo/widgets the path is /demo/widgets.</p>
QUERY_STRING	<p>Specifies the query, if any, attached to the URI used to invoke the request. The value is stored as a String.</p> <p>Queries appear at the end of the URI after a ?. For example, if a request is made to http://cxf.apache.org/demo/widgets?color the query is color.</p>
MTOM_ENABLED	<p>Specifies whether or not the service provider can use MTOM for SOAP attachments. The value is stored as a Boolean.</p>
SCHEMA_VALIDATION_ENABLED	<p>Specifies whether or not the service provider validates messages against a schema. The value is stored as a Boolean.</p>
FAULT_STACKTRACE_ENABLED	<p>Specifies if the runtime provides a stack trace along with a fault message. The value is stored as a Boolean.</p>
CONTENT_TYPE	<p>Specifies the MIME type of the message. The value is stored as a String.</p>
BASE_PATH	<p>Specifies the path of the resource being requested. The value is stored as a java.net.URL.</p> <p>The path is the portion of the URI after the hostname and before any query string. For example, if an endpoint's URL is http://cxf.apache.org/demo/widgets the base path is /demo/widgets.</p>
ENCODING	<p>Specifies the encoding of the message. The value is stored as a String.</p>
FIXED_PARAMETER_ORDER	<p>Specifies whether the parameters must appear in the message in a particular order. The value is stored as a Boolean.</p>

Property Name	Description
MAINTAIN_SESSION	Specifies if the consumer wants to maintain the current session for future requests. The value is stored as a Boolean .
WSDL_DESCRIPTION	Specifies the WSDL document that defines the service being implemented. The value is stored as a org.xml.sax.InputSource object.
WSDL_SERVICE	Specifies the qualified name of the wsdl:service element that defines the service being implemented. The value is stored as a QName .
WSDL_PORT	Specifies the qualified name of the wsdl:port element that defines the endpoint used to access the service. The value is stored as a QName .
WSDL_INTERFACE	Specifies the qualified name of the wsdl:portType element that defines the service being implemented. The value is stored as a QName .
WSDL_OPERATION	Specifies the qualified name of the wsdl:operation element that corresponds to the operation invoked by the consumer. The value is stored as a QName .

javax.xml.ws.handler.MessageContext

MESSAGE_OUTBOUND_PROPERTY	Specifies if a message is outbound. The value is stored as a Boolean . true specifies that a message is outbound.
INBOUND_MESSAGE_ATTACHMENTS	Contains any attachments included in the request message. The value is stored as a java.util.Map<String, DataHandler> . The key value for the map is the MIME Content-ID for the header.
OUTBOUND_MESSAGE_ATTACHMENTS	Contains any attachments for the response message. The value is stored as a java.util.Map<String, DataHandler> . The key value for the map is the MIME Content-ID for the header.
WSDL_DESCRIPTION	Specifies the WSDL document that defines the service being implemented. The value is stored as a org.xml.sax.InputSource object.

Property Name	Description
WSDL_SERVICE	Specifies the qualified name of the wsdl:service element that defines the service being implemented. The value is stored as a QName .
WSDL_PORT	Specifies the qualified name of the wsdl:port element that defines the endpoint used to access the service. The value is stored as a QName .
WSDL_INTERFACE	Specifies the qualified name of the wsdl:portType element that defines the service being implemented. The value is stored as a QName .
WSDL_OPERATION	Specifies the qualified name of the wsdl:operation element that corresponds to the operation invoked by the consumer. The value is stored as a QName .
HTTP_RESPONSE_CODE	Specifies the response code returned to the consumer. The value is stored as an Integer object.
HTTP_REQUEST_HEADERS	Specifies the HTTP headers on a request. The value is stored as a java.util.Map<String, List<String>> .
HTTP_RESPONSE_HEADERS	Specifies the HTTP headers for the response. The value is stored as a java.util.Map<String, List<String>> .
HTTP_REQUEST_METHOD	Specifies the HTTP verb sent with a request. The value is stored as a String .
SERVLET_REQUEST	Contains the servlet's request object. The value is stored as a javax.servlet.http.HttpServletRequest .
SERVLET_RESPONSE	Contains the servlet's response object. The value is stored as a javax.servlet.http.HttpServletResponse .
SERVLET_CONTEXT	Contains the servlet's context object. The value is stored as a javax.servlet.ServletContext .
PATH_INFO	<p>Specifies the path of the resource being requested. The value is stored as a String. The path is the portion of the URI after the hostname and before any query string. For example, if an endpoint's URL is http://cxf.apache.org/demo/widgets the path is /demo/widgets.</p>

Property Name	Description
QUERY_STRING	Specifies the query, if any, attached to the URI used to invoke the request. The value is stored as a String . Queries appear at the end of the URI after a ? . For example, if a request is made to http://cxf.apache.org/demo/widgets?color the query string is color .
REFERENCE_PARAMETERS	Specifies the WS-Addressing reference parameters. This includes all of the SOAP headers whose wsa:isReferenceParameter attribute is set to true . The value is stored as a java.util.List .
org.apache.cxf.transport.jms.JMSConstants	
JMS_SERVER_HEADERS	Contains the JMS message headers. For more information see Section 42.4, "Working with JMS Message Properties" .

[a] When using HTTP this property is the same as the standard JAX-WS defined property.

42.3. WORKING WITH CONTEXTS IN A CONSUMER IMPLEMENTATION

Overview

Consumer implementations have access to context information through the `BindingProvider` interface. The `BindingProvider` instance holds context information in two separate contexts:

- Request Context The *request context* enables you to set properties that affect outbound messages. Request context properties are applied to a specific port instance and, once set, the properties affect every subsequent operation invocation made on the port, until such time as a property is explicitly cleared. For example, you might use a request context property to set a connection timeout or to initialize data for sending in a header.
- Response Context The *response context* enables you to read the property values set by the response to the last operation invocation made from the current thread. Response context properties are reset after every operation invocation. For example, you might access a response context property to read header information received from the last inbound message.



IMPORTANT

Only information that is placed in the application scope of a message context can be accessed by the consumer implementation.

Obtaining a context

Contexts are obtained using the `javax.xml.ws.BindingProvider` interface. The `BindingProvider` interface has two methods for obtaining a context:

- **get RequestContext()** The **get RequestContext()** method, shown in [Example 42.7, "The get RequestContext\(\) Method"](#), returns the request context as a **Map** object. The returned **Map** object can be used to directly manipulate the contents of the context.

Example 42.7. The get RequestContext() Method

```
Map<String, Object> getRequestContext
```

- **getResponse Context()** The **get Response Context()**, shown in [Example 42.8, "The get Response Context\(\) Method"](#), returns the response context as a **Map** object. The returned **Map** object's contents reflect the state of the response context's contents from the most recent successful request on a remote service made in the current thread.

Example 42.8. The get Response Context() Method

```
Map<String, Object> getResponseContext
```

Since proxy objects implement the `BindingProvider` interface, a `BindingProvider` object can be obtained by casting a proxy object. The contexts obtained from the `BindingProvider` object are only valid for operations invoked on the proxy object used to create it.

[Example 42.9, "Getting a Consumer's Request Context"](#) shows code for obtaining the request context for a proxy.

Example 42.9. Getting a Consumer's Request Context

```
// Proxy widgetProxy obtained previously
BindingProvider bp = (BindingProvider)widgetProxy;
Map<String, Object> requestContext = bp.getRequestContext();
```

Reading a property from a context

Consumer contexts are stored in `java.util.Map<String, Object>` objects. The map has keys that are **String** objects and values that contain arbitrary objects. Use `java.util.Map.get()` to access an entry in the map of response context properties.

To retrieve a particular context property, `ContextPropertyName`, use the code shown in [Example 42.10, "Reading a Response Context Property"](#).

Example 42.10. Reading a Response Context Property

```
// Invoke an operation.
port.SomeOperation();

// Read response context property.
java.util.Map<String, Object> responseContext =
((javax.xml.ws.BindingProvider)port).getResponseContext();
PropertyType propValue = (PropertyType) responseContext.get(ContextPropertyName);
```

Setting properties in a context

Consumer contexts are hash maps stored in **java.util.Map<String, Object>** objects. The map has keys that are **String** objects and values that are arbitrary objects. To set a property in a context use the **java.util.Map.put()** method.

While you can set properties in both the request context and the response context, only the changes made to the request context have any impact on message processing. The properties in the response context are reset when each remote invocation is completed on the current thread.

The code shown in [Example 42.11, "Setting a Request Context Property"](#) changes the address of the target service provider by setting the value of the `BindingProvider.ENDPOINT_ADDRESS_PROPERTY`.

Example 42.11. Setting a Request Context Property

```
// Set request context property.
java.util.Map<String, Object> requestContext =
    ((javax.xml.ws.BindingProvider)port).getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
"http://localhost:8080/widgets");

// Invoke an operation.
port.SomeOperation();
```



IMPORTANT

Once a property is set in the request context its value is used for all subsequent remote invocations. You can change the value and the changed value will then be used.

Supported contexts

Apache CXF supports the following context properties in consumer implementations:

Table 42.2. Consumer Context Properties

Property Name	Description
javax.xml.ws.BindingProvider	
ENDPOINT_ADDRESS_PROPERTY	Specifies the address of the target service. The value is stored as a String .
USERNAME_PROPERTY ^[a]	Specifies the username used for HTTP basic authentication. The value is stored as a String .
PASSWORD_PROPERTY ^[b]	Specifies the password used for HTTP basic authentication. The value is stored as a String .
SESSION_MAINTAIN_PROPERTY ^[c]	Specifies if the client wants to maintain session information. The value is stored as a Boolean object.
org.apache.cxf.ws.addressing.JAXWSAConstants	

Property Name	Description
CLIENT_ADDRESSING_PROPERTIES	Specifies the WS-Addressing information used by the consumer to contact the desired service provider. The value is stored as a org.apache.cxf.ws.addressing.AddressingProperties .
org.apache.cxf.transports.jms.context.JMSConstants	
JMS_CLIENT_REQUEST_HEADERS	Contains the JMS headers for the message. For more information see Section 42.4, “Working with JMS Message Properties” .

[a] This property is overridden by the username defined in the HTTP security settings.
 [b] This property is overridden by the password defined in the HTTP security settings.
 [c] The Apache CXF ignores this property.

42.4. WORKING WITH JMS MESSAGE PROPERTIES

Abstract

The Apache CXF JMS transport has a context mechanism that can be used to inspect a JMS message's properties. The context mechanism can also be used to set a JMS message's properties.

42.4.1. Inspecting JMS Message Headers

Abstract

Consumers and services use different context mechanisms to access the JMS message header properties. However, both mechanisms return the header properties as a **org.apache.cxf.transports.jms.context.JMSMessageHeadersType** object.

Getting the JMS Message Headers in a Service

To get the JMS message header properties from the **WebServiceContext** object, do the following:

1. Obtain the context as described in [the section called “Obtaining a context”](#).
2. Get the message headers from the message context using the message context's **get()** method with the parameter `org.apache.cxf.transports.jms.JMSConstants.JMS_SERVER_HEADERS`.

[Example 42.12, “Getting JMS Message Headers in a Service Implementation”](#) shows code for getting the JMS message headers from a service's message context:

Example 42.12. Getting JMS Message Headers in a Service Implementation

```
import org.apache.cxf.transport.jms.JMSConstants;
import org.apache.cxf.transports.jms.context.JMSMessageHeadersType;
```

```

@WebService(serviceName = "HelloWorldService",
            portName = "HelloWorldPort",
            endpointInterface = "org.apache.cxf.hello_world_jms.HelloWorldPortType",
            targetNamespace = "http://cxf.apache.org/hello_world_jms")
public class GreeterImplTwoWayJMS implements HelloWorldPortType
{
    @Resource
    protected WebServiceContext wsContext;
    ...

    @WebMethod
    public String greetMe(String me)
    {
        MessageContext mc = wsContext.getMessageContext();
        JMSMessageHeadersType headers = (JMSMessageHeadersType)
mc.get(JMSConstants.JMS_SERVER_HEADERS);
        ...
    }
    ...
}

```

Getting JMS Message Header Properties in a Consumer

Once a message is successfully retrieved from the JMS transport you can inspect the JMS header properties using the consumer's response context. In addition, you can set or check the length of time the client will wait for a response before timing out, as described in [the section called “Client Receive Timeout”](#). To get the JMS message headers from a consumer's response context do the following:

1. Get the response context as described in [the section called “Obtaining a context”](#).
2. Get the JMS message header properties from the response context using the context's `get()` method with `org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_RESPONSE_HEADERS` as the parameter.

[Example 42.13, “Getting the JMS Headers from a Consumer Response Header”](#) shows code for getting the JMS message header properties from a consumer's response context.

Example 42.13. Getting the JMS Headers from a Consumer Response Header

```

import org.apache.cxf.transports.jms.context.*;
// Proxy greeter initialized previously
BindingProvider bp = (BindingProvider)greeter;
Map<String, Object> responseContext = bp.getResponseContext();
JMSMessageHeadersType responseHdr = (JMSMessageHeadersType)
    responseContext.get(JMSConstants.JMS_CLIENT_RESPONSE_HEADERS);
...
}

```

The code in [Example 42.13, “Getting the JMS Headers from a Consumer Response Header”](#) does the following:

Casts the proxy to a BindingProvider.

Gets the response context.

Retrieves the JMS message headers from the response context.

42.4.2. Inspecting the Message Header Properties

Standard JMS Header Properties

[Table 42.3, “JMS Header Properties”](#) lists the standard properties in the JMS header that you can inspect.

Table 42.3. JMS Header Properties

Property Name	Property Type	Getter Method
Correlation ID	string	<code>getJMSCorrelationID()</code>
Delivery Mode	int	<code>getJMSDeliveryMode()</code>
Message Expiration	long	<code>getJMSExpiration()</code>
Message ID	string	<code>getJMSMessageID()</code>
Priority	int	<code>getJMSPriority()</code>
Redelivered	boolean	<code>getJMSRedelivered()</code>
Time Stamp	long	<code>getJMSTimeStamp()</code>
Type	string	<code>getJMSType()</code>
Time To Live	long	<code>getTimeToLive()</code>

Optional Header Properties

In addition, you can inspect any optional properties stored in the JMS header using `JMSMessageHeadersType.getProperty()`. The optional properties are returned as a [List](#) of `org.apache.cxf.transports.jms.context.JMSPropertyType`. Optional properties are stored as name/value pairs.

Example

[Example 42.14, “Reading the JMS Header Properties”](#) shows code for inspecting some of the JMS properties using the response context.

Example 42.14. Reading the JMS Header Properties

```
// JMSMessageHeadersType messageHdr retrieved previously
```

```

System.out.println("Correlation ID: "+messageHdr.getJMSCorrelationID());
System.out.println("Message Priority: "+messageHdr.getJMSPriority());
System.out.println("Redelivered: "+messageHdr.getRedelivered());

JMSPropertyType prop = null;
List<JMSPropertyType> optProps = messageHdr.getProperty();
Iterator<JMSPropertyType> iter = optProps.iterator();
while (iter.hasNext())
{
    prop = iter.next();
    System.out.println("Property name: "+prop.getName());
    System.out.println("Property value: "+prop.getValue());
}

```

The code in [Example 42.14, “Reading the JMS Header Properties”](#) does the following:

Prints the value of the message’s correlation ID.

Prints the value of the message’s priority property.

Prints the value of the message’s redelivered property.

Gets the list of the message’s optional header properties.

Gets an **Iterator** to traverse the list of properties.

Iterates through the list of optional properties and prints their name and value.

42.4.3. Setting JMS Properties

Abstract

Using the request context in a consumer endpoint, you can set a number of the JMS message header properties and the consumer endpoint’s timeout value. These properties are valid for a single invocation. You must reset them each time you invoke an operation on the service proxy.

Note that you cannot set header properties in a service.

JMS Header Properties

[Table 42.4, “Settable JMS Header Properties”](#) lists the properties in the JMS header that can be set using the consumer endpoint’s request context.

Table 42.4. Settable JMS Header Properties

Property Name	Property Type	Setter Method
Correlation ID	string	setJMSCorralationID()
Delivery Mode	int	setJMSDeliveryMode()
Priority	int	setJMSPriority()

Property Name	Property Type	Setter Method
Time To Live	long	setTimeToLive()

To set these properties do the following:

1. Create an **org.apache.cxf.transports.jms.context.JMSMessageHeadersType** object.
2. Populate the values you want to set using the appropriate setter methods described in [Table 42.4, "Settable JMS Header Properties"](#).
3. Set the values to the request context by calling the request context's **put()** method using `org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_REQUEST_HEADERS` as the first argument, and the new **JMSMessageHeadersType** object as the second argument.

Optional JMS Header Properties

You can also set optional properties to the JMS header. Optional JMS header properties are stored in the **JMSMessageHeadersType** object that is used to set the other JMS header properties. They are stored as a **List** object containing **org.apache.cxf.transports.jms.context.JMSPropertyType** objects. To add optional properties to the JMS header do the following:

1. Create a **JMSPropertyType** object.
2. Set the property's name field using **setName()**.
3. Set the property's value field using **setValue()**.
4. Add the property to the JMS message header using **JMSMessageHeadersType.getProperty().add(JMSPropertyType)**.
5. Repeat the procedure until all of the properties have been added to the message header.

Client Receive Timeout

In addition to the JMS header properties, you can set the amount of time a consumer endpoint waits for a response before timing out. You set the value by calling the request context's **put()** method with `org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_RECEIVE_TIMEOUT` as the first argument and a **long** representing the amount of time in milliseconds that you want the consumer to wait as the second argument.

Example

[Example 42.15, "Setting JMS Properties using the Request Context"](#) shows code for setting some of the JMS properties using the request context.

Example 42.15. Setting JMS Properties using the Request Context

```
import org.apache.cxf.transports.jms.context.*;
// Proxy greeter initialized previously
InvocationHandler handler = Proxy.getInvocationHandler(greeter);
```

```
BindingProvider bp= null;
if (handler instanceof BindingProvider)
{
    bp = (BindingProvider)handler;
    Map<String, Object> requestContext = bp.getRequestContext();

    JMSMessageHeadersType requestHdr = new JMSMessageHeadersType();
    requestHdr.setJMSCorrelationID("WithBob");
    requestHdr.setJMSExpiration(3600000L);

    JMSPropertyType prop = new JMSPropertyType;
    prop.setName("MyProperty");
    prop.setValue("Bluebird");
    requestHdr.getProperty().add(prop);

    requestContext.put(JMSConstants.CLIENT_REQUEST_HEADERS, requestHdr);
    requestContext.put(JMSConstants.CLIENT_RECEIVE_TIMEOUT, new Long(1000));
}
```

The code in [Example 42.15, “Setting JMS Properties using the Request Context”](#) does the following:

Gets the **InvocationHandler** for the proxy whose JMS properties you want to change.

Checks to see if the **InvocationHandler** is a **BindingProvider**.

Casts the returned **InvocationHandler** object into a **BindingProvider** object to retrieve the request context.

Gets the request context.

Creates a **JMSMessageHeadersType** object to hold the new message header values.

Sets the Correlation ID.

Sets the Expiration property to 60 minutes.

Creates a new **JMSPropertyType** object.

Sets the values for the optional property.

Adds the optional property to the message header.

Sets the JMS message header values into the request context.

Sets the client receive timeout property to 1 second.

CHAPTER 43. WRITING HANDLERS

Abstract

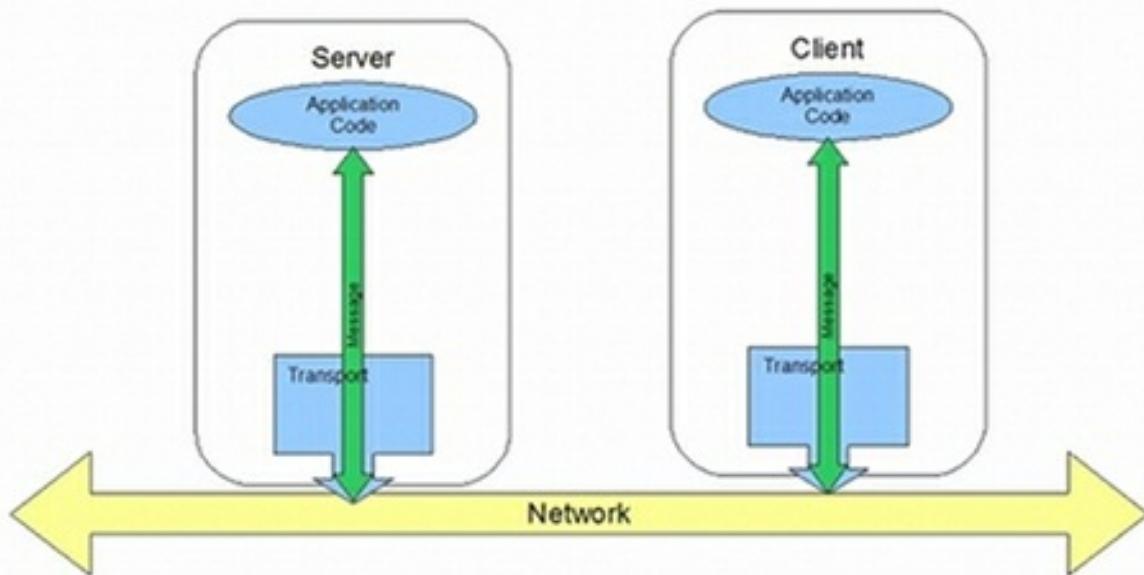
JAX-WS provides a flexible plug-in framework for adding message processing modules to an application. These modules, known as handlers, are independent of the application level code and can provide low-level message processing capabilities.

43.1. HANDLERS: AN INTRODUCTION

Overview

When a service proxy invokes an operation on a service, the operation's parameters are passed to Apache CXF where they are built into a message and placed on the wire. When the message is received by the service, Apache CXF reads the message from the wire, reconstructs the message, and then passes the operation parameters to the application code responsible for implementing the operation. When the application code is finished processing the request, the reply message undergoes a similar chain of events on its trip to the service proxy that originated the request. This is shown in [Figure 43.1, "Message Exchange Path"](#).

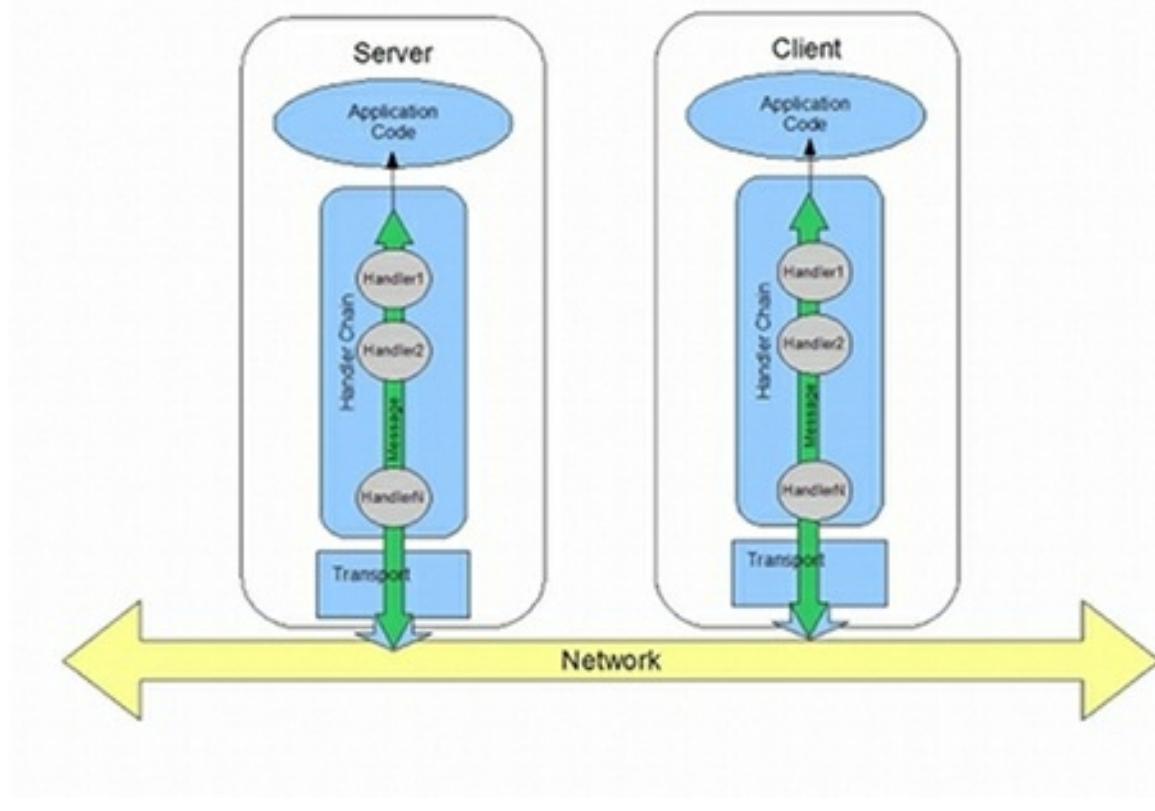
Figure 43.1. Message Exchange Path



JAX-WS defines a mechanism for manipulating the message data between the application level code and the network. For example, you might want the message data passed over the open network to be encrypted using a proprietary encryption mechanism. You could write a JAX-WS handler that encrypted and decrypted the data. Then you could insert the handler into the message processing chains of all clients and servers.

As shown in [Figure 43.2, "Message Exchange Path with Handlers"](#), the handlers are placed in a chain that is traversed between the application level code and the transport code that places the message onto the network.

Figure 43.2. Message Exchange Path with Handlers



Handler types

The JAX-WS specification defines two basic handler types:

- **Logical Handler** Logical handlers can process the message payload and the properties stored in the message context. For example, if the application uses pure XML messages, the logical handlers have access to the entire message. If the application uses SOAP messages, the logical handlers have access to the contents of the SOAP body. They do not have access to either the SOAP headers or any attachments unless they were placed into the message context.
Logical handlers are placed closest to the application code on the handler chain. This means that they are executed first when a message is passed from the application code to the transport. When a message is received from the network and passed back to the application code, the logical handlers are executed last.
- **Protocol Handler** Protocol handlers can process the entire message received from the network and the properties stored in the message context. For example, if the application uses SOAP messages, the protocol handlers would have access to the contents of the SOAP body, the SOAP headers, and any attachments.
Protocol handlers are placed closest to the transport on the handler chain. This means that they are executed first when a message is received from the network. When a message is sent to the network from the application code, the protocol handlers are executed last.



NOTE

The only protocol handler supported by Apache CXF is specific to SOAP.

Implementation of handlers

The differences between the two handler types are very subtle and they share a common base interface. Because of their common parentage, logical handlers and protocol handlers share a number of methods that must be implemented, including:

- `handleMessage()` The **handleMessage()** method is the central method in any handler. It is the method responsible for processing normal messages.
- `handleFault()` **handleFault()** is the method responsible for processing fault messages.
- `close()` **close()** is called on all executed handlers in a handler chain when a message has reached the end of the chain. It is used to clean up any resources consumed during message processing.

The differences between the implementation of a logical handler and the implementation of a protocol handler revolve around the following:

- The specific interface that is implemented
All handlers implement an interface that derives from the Handler interface. Logical handlers implement the LogicalHandler interface. Protocol handlers implement protocol specific extensions of the Handler interface. For example, SOAP handlers implement the SOAPHandler interface.
- The amount of information available to the handler
Protocol handlers have access to the contents of messages and all of the protocol specific information that is packaged with the message content. Logical handlers can only access the contents of the message. Logical handlers have no knowledge of protocol details.

Adding handlers to an application

To add a handler to an application you must do the following:

1. Determine whether the handler is going to be used on the service providers, the consumers, or both.
2. Determine which type of handler is the most appropriate for the job.
3. Implement the proper interface.
To implement a logical handler see [Section 43.2, “Implementing a Logical Handler”](#).
To implement a protocol handler see [Section 43.4, “Implementing a Protocol Handler”](#).
4. Configure your endpoint(s) to use the handlers. See [Section 43.10, “Configuring Endpoints to Use Handlers”](#).

43.2. IMPLEMENTING A LOGICAL HANDLER

Overview

Logical handlers implement the javax.xml.ws.handler.LogicalHandler interface. The LogicalHandler interface, shown in [Example 43.1, “LogicalHandler Synopsis”](#) passes a **LogicalMessageContext** object to the **handleMessage()** method and the **handleFault()** method. The context object provides access to the **body** of the message and to any properties set into the message exchange’s context.

Example 43.1. LogicalHandler Synopsis

```

public interface LogicalHandler extends Handler
{
    boolean handleMessage(LogicalMessageContext context);
    boolean handleFault(LogicalMessageContext context);
    void close(LogicalMessageContext context);
}

```

Procedure

To implement a logical hander you do the following:

1. Implement any [Section 43.6, “Initializing a Handler”](#) logic required by the handler.
2. Implement the [Section 43.3, “Handling Messages in a Logical Handler”](#) logic.
3. Implement the [Section 43.7, “Handling Fault Messages”](#) logic.
4. Implement the logic for [Section 43.8, “Closing a Handler”](#) the handler when it is finished.
5. Implement any logic for [Section 43.9, “Releasing a Handler”](#) the handler’s resources before it is destroyed.

43.3. HANDLING MESSAGES IN A LOGICAL HANDLER

Overview

Normal message processing is handled by the **handleMessage()** method.

The **handleMessage()** method receives a **LogicalMessageContext** object that provides access to the message body and any properties stored in the message context.

The **handleMessage()** method returns either true or false depending on how message processing is to continue. It can also throw an exception.

Getting the message data

The LogicalMessageContext object passed into logical message handlers allows access to the message body using the context’s **getMessage()** method. The **getMessage()** method, shown in [Example 43.2, “Method for Getting the Message Payload in a Logical Handler”](#), returns the message payload as a LogicalMessage object.

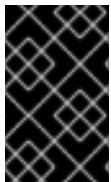
Example 43.2. Method for Getting the Message Payload in a Logical Handler

LogicalMessage**getMessage**

Once you have the LogicalMessage object, you can use it to manipulate the message body. The LogicalMessage interface, shown in [Example 43.3, “Logical Message Holder”](#), has getters and setters for working with the actual message body.

Example 43.3. Logical Message Holder

```
LogicalMessageSource getPayloadObject()  
getPayloadJAXBContext context()  
setPayloadObject(Object payload)  
setPayloadJAXBContext(JAXBContext context())  
setPayloadSource(payload)
```



IMPORTANT

The contents of the message payload are determined by the type of binding in use. The SOAP binding only allows access to the SOAP body of the message. The XML binding allows access to the entire message body.

Working with the message body as an XML object

One pair of getters and setters of the logical message work with the message payload as a **javax.xml.transform.dom.DOMSource** object.

The **getPayload()** method that has no parameters returns the message payload as a **DOMSource** object. The returned object is the actual message payload. Any changes made to the returned object change the message body immediately.

You can replace the body of the message with a **DOMSource** object using the **setPayload()** method that takes the single Source object.

Working with the message body as a JAXB object

The other pair of getters and setters allow you to work with the message payload as a JAXB object. They use a **JAXBContext** object to transform the message payload into JAXB objects.

To use the JAXB objects you do the following:

1. Get a **JAXBContext** object that can manage the data types in the message body.
For information on creating a **JAXBContext** object see [Chapter 39, Using A **JAXBContext** Object](#).
2. Get the message body as shown in [Example 43.4, "Getting the Message Body as a JAXB Object"](#).

Example 43.4. Getting the Message Body as a JAXB Object

```
JAXBContext jaxbc = JAXBContext(myObjectFactory.class);
Object body = message.getPayload(jaxbc);
```

3. Cast the returned object to the proper type.
4. Manipulate the message body as needed.
5. Put the updated message body back into the context as shown in [Example 43.5, "Updating the Message Body Using a JAXB Object"](#).

Example 43.5. Updating the Message Body Using a JAXB Object

```
message.setPayload(body, jaxbc);
```

Working with context properties

The logical message context passed into a logical handler is an instance of the application's message context and can access all of the properties stored in it. Handlers have access to properties at both the **APPLICATION** scope and the **HANDLER** scope.

Like the application's message context, the logical message context is a subclass of Java Map. To access the properties stored in the context, you use the **get()** method and **put()** method inherited from the Map interface.

By default, any properties you set in the message context from inside a logical handler are assigned a scope of **HANDLER**. If you want the application code to be able to access the property you need to use the context's **setScope()** method to explicitly set the property's scope to **APPLICATION**.

For more information on working with properties in the message context see [Section 42.1, "Understanding Contexts"](#).

Determining the direction of the message

It is often important to know the direction a message is passing through the handler chain. For example, you would want to retrieve a security token from incoming requests and attach a security token to an outgoing response.

The direction of the message is stored in the message context's outbound message property. You retrieve the outbound message property from the message context using the `MessageContext.MESSAGE_OUTBOUND_PROPERTY` key as shown in [Example 43.6, "Getting the Message's Direction from the SOAP Message Context"](#).

Example 43.6. Getting the Message's Direction from the SOAP Message Context

```
Boolean outbound;
outbound = (Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
```

The property is stored as a **Boolean** object. You can use the object's **booleanValue()** method to determine the property's value. If the property is set to true, the message is outbound. If the property is set to false the message is inbound.

Determining the return value

How the **handleMessage()** method completes its message processing has a direct impact on how message processing proceeds. It can complete by doing one of the following actions:

1. Return true—Returning true signals to the Apache CXF runtime that message processing should continue normally. The next handler, if any, has its **handleMessage()** invoked.
2. Return false—Returning false signals to the Apache CXF runtime that normal message processing must stop. How the runtime proceeds depends on the message exchange pattern in use for the **current message**.

For request-response message exchanges the following happens:

- a. The direction of message processing is reversed.

For example, if a request is being processed by a service provider, the message stops progressing toward the service's implementation object. Instead, it is sent back towards the binding for return to the consumer that originated the request.

- b. Any message handlers that reside along the handler chain in the new processing direction have their **handleMessage()** method invoked in the order in which they reside in the chain.
- c. When the message reaches the end of the handler chain it is dispatched.
For one-way message exchanges the following happens:
 - d. Message processing stops.
 - e. All previously invoked message handlers have their **close()** method invoked.
 - f. The message is dispatched.
- 3. Throw a ProtocolException exception—Throwing a ProtocolException exception, or a subclass of this exception, signals the Apache CXF runtime that fault message processing is beginning. How the runtime proceeds depends on the message exchange pattern in use for the **current message**.
For request-response message exchanges the following happens:
 - a. If the handler has not already created a fault message, the runtime wraps the message in a fault message.
 - b. The direction of message processing is reversed.
For example, if a request is being processed by a service provider, the message stops progressing toward the service's implementation object. Instead, it is sent back towards the binding for return to the consumer that originated the request.
 - c. Any message handlers that reside along the handler chain in the new processing direction have their **handleFault()** method invoked in the order in which they reside in the chain.
 - d. When the fault message reaches the end of the handler chain it is dispatched.
For one-way message exchanges the following happens:
 - e. If the handler has not already created a fault message, the runtime wraps the message in a fault message.
 - f. Message processing stops.
 - g. All previously invoked message handlers have their **close()** method invoked.
 - h. The fault message is dispatched.
- 4. Throw any other runtime exception—Throwing a runtime exception other than a ProtocolException exception signals the Apache CXF runtime that message processing is to stop. All previously invoked message handlers have the **close()** method invoked and the exception is dispatched. If the message is part of a request-response message exchange, the exception is dispatched so that it is returned to the consumer that originated the request.

Example

[Example 43.7, “Logical Message Handler Message Processing”](#) shows an implementation of **handleMessage()** message for a logical message handler that is used by a service consumer. It processes requests before they are sent to the service provider.

Example 43.7. Logical Message Handler Message Processing

```
public class SmallNumberHandler implements LogicalHandler<LogicalMessageContext>
```

```

{
    public final boolean handleMessage(LogicalMessageContext messageContext)
    {
        try
        {
            boolean outbound =
(Boolean)messageContext.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

            if (outbound)
            {
                LogicalMessage msg = messageContext.getMessage();

                JAXBContext jaxbContext = JAXBContext.newInstance(ObjectFactory.class);
                Object payload = msg.getPayload(jaxbContext);
                if (payload instanceof JAXBElement)
                {
                    payload = ((JAXBElement)payload).getValue();
                }

                if (payload instanceof AddNumbers)
                {
                    AddNumbers req = (AddNumbers)payload;

                    int a = req.getArg0();
                    int b = req.getArg1();
                    int answer = a + b;

                    if (answer < 20)
                    {
                        AddNumbersResponse resp = new AddNumbersResponse();
                        resp.setReturn(answer);
                        msg.setPayload(new ObjectFactory().createAddNumbersResponse(resp),
                                      jaxbContext);

                        return false;
                    }
                }
                else
                {
                    throw new WebServiceException("Bad Request");
                }
            }
            return true;
        }
        catch (JAXBException ex)
        {
            throw new ProtocolException(ex);
        }
    }
}
...
}

```

The code in [Example 43.7, “Logical Message Handler Message Processing”](#) does the following:

Checks if the message is an outbound request.

If the message is an outbound request, the handler does additional message processing.

Gets the LogicalMessage representation of the message payload from the message context.

Gets the actual message payload as a JAXB object.

Checks to make sure the request is of the correct type.

If it is, the handler continues processing the message.

Checks the value of the sum.

If it is less than the threshold of 20 then it builds a response and returns it to the client.

Builds the response.

Returns false to stop message processing and return the response to the client.

Throws a runtime exception if the message is not of the correct type.

This exception is returned to the client.

Returns true if the message is an inbound response or the sum does not meet the threshold.

Message processing continues normally.

Throws a ProtocolException if a JAXB marshalling error is encountered.

The exception is passed back to the client after it is processed by the **handleFault()** method of the handlers between the current handler and the client.

43.4. IMPLEMENTING A PROTOCOL HANDLER

Overview

Protocol handlers are specific to the protocol in use. Apache CXF provides the SOAP protocol handler as specified by JAX-WS. A SOAP protocol handler implements the `javax.xml.ws.handler.soap.SOAPHandler` interface.

The `SOAPHandler` interface, shown in [Example 43.8, “SOAPHandler Synopsis”](#), uses a SOAP specific message context that provides access to the message as a `SOAPMessage` object. It also allows you to access the SOAP headers.

Example 43.8. SOAPHandler Synopsis

```
public interface SOAPHandler extends Handler
{
    boolean handleMessage(SOAPMessageContext context);
    boolean handleFault(SOAPMessageContext context);
    void close(SOAPMessageContext context);
    Set<QName> getHeaders()
}
```

In addition to using a SOAP specific message context, SOAP protocol handlers require that you implement an additional method called **getHeaders()**. This additional method returns the QNames of the header blocks the handler can process.

Procedure

To implement a logical hander do the following:

1. Implement any [Section 43.6, “Initializing a Handler”](#) logic required by the handler.
2. Implement the [Section 43.5, “Handling Messages in a SOAP Handler”](#) logic.
3. Implement the [Section 43.7, “Handling Fault Messages”](#) logic.
4. Implement the **getHeaders()** method.
5. Implement the logic for [Section 43.8, “Closing a Handler”](#) the handler when it is finished.
6. Implement any logic for [Section 43.9, “Releasing a Handler”](#) the handler’s resources before it is destroyed.

Implementing the getHeaders() method

The **getHeaders()**, shown in [Example 43.9, “The SOAPHandler.getHeaders\(\) Method”](#), method informs the Apache CXF runtime what SOAP headers the handler is responsible for processing. It returns the QNames of the outer element of each SOAP header the handler understands.

Example 43.9. The SOAPHandler.getHeaders() Method

```
Set<QName>getHeaders
```

For many cases simply returning null is sufficient. However, if the application uses the **mustUnderstand** attribute of any of the SOAP headers, then it is important to specify the headers understood by the application’s SOAP handlers. The runtime checks the set of SOAP headers that all of the registered handlers understand against the list of headers with the **mustUnderstand** attribute set to **true**. If any of the flagged headers are not in the list of understood headers, the runtime rejects the message and throws a SOAP must understand exception.

43.5. HANDLING MESSAGES IN A SOAP HANDLER

Overview

Normal message processing is handled by the **handleMessage()** method.

The **handleMessage()** method receives a **SOAPMessageContext** object that provides access to the message body as a **SOAPMessage** object and the SOAP headers associated with the message. In addition, the context provides access to any properties stored in the message context.

The **handleMessage()** method returns either true or false depending on how message processing is to continue. It can also throw an exception.

Working with the message body

You can get the SOAP message using the SOAP message context's **getMessage()** method. It returns the message as a live **SOAPMessage** object. Any changes to the message in the handler are automatically reflected in the message stored in the context.

If you wish to replace the existing message with a new one, you can use the context's **setMessage()** method. The **setMessage()** method takes a **SOAPMessage** object.

Getting the SOAP headers

You can access the SOAP message's headers using the **SOAPMessage** object's **getHeader()** method. This will return the SOAP header as a **SOAPHeader** object that you will need to inspect to find the header elements you wish to process.

The SOAP message context provides a **getHeaders()** method, shown in [Example 43.10, "The SOAPMessageContext.getHeaders\(\) Method"](#), that will return an array containing JAXB objects for the specified SOAP headers.

Example 43.10. The **SOAPMessageContext.getHeaders()** Method

```
Object[]getHeaders(QNameheaderJAXBContextcontextbooleanallRoles
```

You specify the headers using the QName of their element. You can further limit the headers that are returned by setting the **allRoles** parameter to false. That instructs the runtime to only return the SOAP headers that are applicable to the active SOAP roles.

If no headers are found, the method returns an empty array.

For more information about instantiating a **JAXBContext** object see [Chapter 39, Using A **JAXBContext** Object](#).

Working with context properties

The SOAP message context passed into a logical handler is an instance of the application's message context and can access all of the properties stored in it. Handlers have access to properties at both the **APPLICATION** scope and the **Handler** scope.

Like the application's message context, the SOAP message context is a subclass of Java Map. To access the properties stored in the context, you use the **get()** method and **put()** method inherited from the Map interface.

By default, any properties you set in the context from inside a logical handler will be assigned a scope of **HANDLER**. If you want the application code to be able to access the property you need to use the context's **setScope()** method to explicitly set the property's scope to **APPLICATION**.

For more information on working with properties in the message context see [Section 42.1, "Understanding Contexts"](#).

Determining the direction of the message

It is often important to know the direction a message is passing through the handler chain. For example, you would want to add headers to an outgoing message and strip headers from an incoming message.

The direction of the message is stored in the message context's outbound message property. You retrieve the outbound message property from the message context using the

MessageContext.MESSAGE_OUTBOUND_PROPERTY key as shown in [Example 43.11, “Getting the Message’s Direction from the SOAP Message Context”](#).

Example 43.11. Getting the Message’s Direction from the SOAP Message Context

```
Boolean outbound;  
outbound = (Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
```

The property is stored as a **Boolean** object. You can use the object’s **booleanValue()** method to determine the property’s value. If the property is set to true, the message is outbound. If the property is set to false the message is inbound.

Determining the return value

How the **handleMessage()** method completes its message processing has a direct impact on how message processing proceeds. It can complete by doing one of the following actions:

1. return true—Returning true signals to the Apache CXF runtime that message processing should continue normally. The next handler, if any, has its **handleMessage()** invoked.
2. return false—Returning false signals to the Apache CXF runtime that normal message processing is to stop. How the runtime proceeds depends on the message exchange pattern in use for the **current message**.

For request-response message exchanges the following happens:

- a. The direction of message processing is reversed.
For example, if a request is being processed by a service provider, the message will stop progressing toward the service’s implementation object. It will instead be sent back towards the binding for return to the consumer that originated the request.

- b. Any message handlers that reside along the handler chain in the new processing direction have their **handleMessage()** method invoked in the order in which they reside in the chain.

- c. When the message reaches the end of the handler chain it is dispatched.

For one-way message exchanges the following happens:

- d. Message processing stops.

- e. All previously invoked message handlers have their **close()** method invoked.

- f. The message is dispatched.

3. throw a ProtocolException exception—Throwing a ProtocolException exception, or a subclass of this exception, signals the Apache CXF runtime that fault message processing is to start. How the runtime proceeds depends on the message exchange pattern in use for the **current message**.

For request-response message exchanges the following happens:

- a. If the handler has not already created a fault message, the runtime wraps the message in a fault message.

- b. The direction of message processing is reversed.

For example, if a request is being processed by a service provider, the message will stop progressing toward the service's implementation object. It will be sent back towards the binding for return to the consumer that originated the request.

- c. Any message handlers that reside along the handler chain in the new processing direction have their **handleFault()** method invoked in the order in which they reside in the chain.
- d. When the fault message reaches the end of the handler chain it is dispatched.
For one-way message exchanges the following happens:
 - e. If the handler has not already created a fault message, the runtime wraps the message in a fault message.
 - f. Message processing stops.
 - g. All previously invoked message handlers have their **close()** method invoked.
 - h. The fault message is dispatched.
- 4. throw any other runtime exception—Throwing a runtime exception other than a `ProtocolException` exception signals the Apache CXF runtime that message processing is to stop. All previously invoked message handlers have the **close()** method invoked and the exception is dispatched. If the message is part of a request-response message exchange the exception is dispatched so that it is returned to the consumer that originated the request.

Example

[Example 43.12, “Handling a Message in a SOAP Handler”](#) shows a **handleMessage()** implementation that prints the SOAP message to the screen.

Example 43.12. Handling a Message in a SOAP Handler

```
public boolean handleMessage(SOAPMessageContext smc)
{
    PrintStream out;

    Boolean outbound =
(Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

    if (outbound.booleanValue())
    {
        out.println("\nOutbound message:");
    }
    else
    {
        out.println("\nInbound message:");
    }

    SOAPMessage message = smc.getMessage();

    message.writeTo(out);
    out.println();

    return true;
}
```

The code in [Example 43.12, “Handling a Message in a SOAP Handler”](#) does the following:

Retrieves the outbound property from the message context.

Tests the messages direction and prints the appropriate message.

Retrieves the SOAP message from the context.

Prints the message to the console.

43.6. INITIALIZING A HANDLER

Overview

When the runtime creates an instance of a handler, it creates all of the resources the hander needs to process messages. While you can place all of the logic for doing this in the handler’s constructor, it may not be the most appropriate place. The handler framework performs a number of optional steps when it instantiates a handler. You can add resource injection and other initialization logic that will be executed during the optional steps.

You do not have to provide any initialization methods for a handler.

Order of initialization

The Apache CXF runtime initializes a handler in the following manner:

1. The handler’s constructor is called.
2. Any resources that are specified by the **@Resource** annotation are injected.
3. The method decorated with **@PostConstruct** annotation, if it is present, is called.



NOTE

Methods decorated with the **@PostConstruct** annotation must have a **void** return type and have no parameters.

4. The handler is place in the **Ready** state.

43.7. HANDLING FAULT MESSAGES

Overview

Handlers use the **handleFault()** method for processing fault messages when a `ProtocolException` exception is thrown during message processing.

The **handleFault()** method receives either a **LogicalMessageContext** object or **SOAPMessageContext** object depending on the type of handler. The received context gives the handler’s implementation access to the message payload.

The **handleFault()** method returns either true or false, depending on how fault message processing is to proceed. It can also throw an exception.

Getting the message payload

The context object received by the **handleFault()** method is similar to the one received by the **handleMessage()** method. You use the context's **getMessage()** method to access the message payload in the same way. The only difference is the payload contained in the context.

For more information on working with a **LogicalMessageContext** see [Section 43.3, “Handling Messages in a Logical Handler”](#).

For more information on working with a **SOAPMessageContext** see [Section 43.5, “Handling Messages in a SOAP Handler”](#).

Determining the return value

How the **handleFault()** method completes its message processing has a direct impact on how message processing proceeds. It completes by performing one of the following actions:

Return true

Returning true signals that fault processing should continue normally. The **handleFault()** method of the next handler in the chain will be invoked.

Return false

Returning false signals that fault processing stops. The **close()** method of the handlers that were invoked in processing the current message are invoked and the fault message is dispatched.

Throw an exception

Throwing an exception stops fault message processing. The **close()** method of the handlers that were invoked in processing the current message are invoked and the exception is dispatched.

Example

[Example 43.13, “Handling a Fault in a Message Handler”](#) shows an implementation of **handleFault()** that prints the message body to the screen.

Example 43.13. Handling a Fault in a Message Handler

```
public final boolean handleFault(LogicalMessageContext messageContext)
{
    System.out.println("handleFault() called with message:");

    LogicalMessage msg=messageContext.getMessage();
    System.out.println(msg.getPayload());

    return true;
}
```

43.8. CLOSING A HANDLER

When a handler chain is finished processing a message, the runtime calls each executed handler's **close()** method. This is the appropriate place to clean up any resources that were used by the handler during message processing or resetting any properties to a default state.

If a resource needs to persist beyond a single message exchange, you should not clean it up during in the handler's **close()** method.

43.9. RELEASING A HANDLER

Overview

The runtime releases a handler when the service or service proxy to which the handler is bound is shutdown. The runtime will invoke an optional release method before invoking the handler's destructor. This optional release method can be used to release any resources used by the handler or perform other actions that would not be appropriate in the handler's destructor.

You do not have to provide any clean-up methods for a handler.

Order of release

The following happens when the handler is released:

1. The handler finishes processing any active messages.
2. The runtime invokes the method decorated with the **@PreDestroy** annotation.
This method should clean up any resources used by the handler.
3. The handler's destructor is called.

43.10. CONFIGURING ENDPOINTS TO USE HANDLERS

43.10.1. Programmatic Configuration

43.10.1.1. Adding a Handler Chain to a Consumer

Overview

Adding a handler chain to a consumer involves explicitly building the chain of handlers. Then you set the handler chain directly on the service proxy's **Binding** object.



IMPORTANT

Any handler chains configured using the Spring configuration override the handler chains configured programmatically.

Procedure

To add a handler chain to a consumer you do the following:

1. Create a **List<Handler>** object to hold the handler chain.
2. Create an instance of each handler that will be added to the chain.
3. Add each of the instantiated handler objects to the list in the order they are to be invoked by the runtime.
4. Get the Binding object from the service proxy.

Apache CXF provides an implementation of the Binding interface called **org.apache.cxf.jaxws.binding.DefaultBindingImpl**.

5. Set the handler chain on the proxy using the Binding object's **setHandlerChain()** method.

Example

[Example 43.14, "Adding a Handler Chain to a Consumer"](#) shows code for adding a handler chain to a consumer.

Example 43.14. Adding a Handler Chain to a Consumer

```
import javax.xml.ws.BindingProvider;
import javax.xml.ws.handler.Handler;
import java.util.ArrayList;
import java.util.List;

import org.apache.cxf.jaxws.binding.DefaultBindingImpl;
...
SmallNumberHandler sh = new SmallNumberHandler();
List<Handler> handlerChain = new ArrayList<Handler>();
handlerChain.add(sh);

DefaultBindingImpl binding = ((BindingProvider)proxy).getBinding();
binding.getBinding().setHandlerChain(handlerChain);
```

The code in [Example 43.14, "Adding a Handler Chain to a Consumer"](#) does the following:

Instantiates a handler.

Creates a **List** object to hold the chain.

Adds the handler to the chain.

Gets the Binding object from the proxy as a **DefaultBindingImpl** object.

Assigns the handler chain to the proxy's binding.

43.10.1.2. Adding a Handler Chain to a Service Provider

Overview

You add a handler chain to a service provider by decorating either the SEI or the implementation class with the **@HandlerChain** annotation. The annotation points to a meta-data file defining the handler chain used by the service provider.

Procedure

To add handler chain to a service provider you do the following:

1. Decorate the provider's implementation class with the **@HandlerChain** annotation.
2. Create a handler configuration file that defines the handler chain.

The @HandlerChain annotation

The **javax.jws.HandlerChain** annotation decorates service provider's implementation class. It instructs the runtime to load the handler chain configuration file specified by its **file** property.

The annotation's **file** property supports two methods for identifying the handler configuration file to load:

- a URL
- a relative path name

[Example 43.15, "Service Implementation that Loads a Handler Chain"](#) shows a service provider implementation that will use the handler chain defined in a file called **handlers.xml**. **handlers.xml** must be located in the directory from which the service provider is run.

Example 43.15. Service Implementation that Loads a Handler Chain

```
import javax.jws.HandlerChain;
import javax.jws.WebService;
...
@WebService(name = "AddNumbers",
            targetNamespace = "http://apache.org/handlers",
            portName = "AddNumbersPort",
            endpointInterface = "org.apache.handlers.AddNumbers",
            serviceName = "AddNumbersService")
@HandlerChain(file = "handlers.xml")
public class AddNumbersImpl implements AddNumbers
{
    ...
}
```

Handler configuration file

The handler configuration file defines a handler chain using the XML grammar that accompanies JSR 109 (Web Services for Java EE, Version 1.2). This grammar is defined in the <http://java.sun.com/xml/ns/javaee>.

The root element of the handler configuration file is the **handler-chains** element. The **handler-chains** element has one or more **handler-chain** elements.

The **handler-chain** element define a handler chain. [Table 43.1, "Elements Used to Define a Server-Side Handler Chain"](#) describes the **handler-chain** element's children.

Table 43.1. Elements Used to Define a Server-Side Handler Chain

Element	Description
handler	Contains the elements that describe a handler.

Element	Description
service-name-pattern	Specifies the QName of the WSDL service element defining the service to which the handler chain is bound. You can use * as a wildcard when defining the QName.
port-name-pattern	Specifies the QName of the WSDL port element defining the endpoint to which the handler chain is bound. You can use * as a wildcard when defining the QName.
protocol-binding	Specifies the message binding for which the handler chain is used. The binding is specified as a URI or using one of the following aliases: ##SOAP11_HTTP,##SOAP11_HTTP_MTOM,##SOAP12_HTTP,##SOAP12_HTTP_MTOM, or ##XML_HTTP. For more information about message binding URLs see Chapter 23, Apache CXF Binding IDs .

The **handler-chain** element is only required to have a single **handler** element as a child. It can, however, support as many **handler** elements as needed to define the complete handler chain. The handlers in the chain are executed in the order they specified in the handler chain definition.



IMPORTANT

The final order of execution will be determined by sorting the specified handlers into logical handlers and protocol handlers. Within the groupings, the order specified in the configuration will be used.

The other children, such as **protocol-binding**, are used to limit the scope of the defined handler chain. For example, if you use the **service-name-pattern** element, the handler chain will only be attached to service providers whose WSDL **port** element is a child of the specified WSDL **service** element. You can only use one of these limiting children in a **handler** element.

The **handler** element defines an individual handler in a handler chain. Its **handler-class** child element specifies the fully qualified name of the class implementing the handler. The **handler** element can also have an optional **handler-name** element that specifies a unique name for the handler.

[Example 43.16, "Handler Configuration File"](#) shows a handler configuration file that defines a single handler chain. The chain is made up of two handlers.

Example 43.16. Handler Configuration File

```
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee">
<handler-chain>
    <handler>
        <handler-name>LoggingHandler</handler-name>
```

```

<handler-class>demo.handlers.common.LoggingHandler</handler-class>
</handler>
<handler>
  <handler-name>AddHeaderHandler</handler-name>
  <handler-class>demo.handlers.common.AddHeaderHandler</handler-class>
</handler>
</handler-chain>
</handler-chains>

```

43.10.2. Spring Configuration

Overview

The easiest way to configure an endpoint to use a handler chain is to define the chain in the endpoint's configuration. This is done by adding a **jaxws:handlers** child to the element configuring the endpoint.



IMPORTANT

A handler chain added through the configuration file takes precedence over a handler chain configured programmatically.

Procedure

To configure an endpoint to load a handler chain you do the following:

1. If the endpoint does not already have a configuration element, add one.
For more information on configuring Apache CXF endpoints see [Chapter 17, Configuring JAX-WS Endpoints](#).
2. Add a **jaxws:handlers** child element to the endpoint's configuration element.
3. For each handler in the chain, add a **bean** element specifying the class that implements the handler.
If your handler implementation is used in more than one place you can reference a **bean** element using the **ref** element.

The handlers element

The **jaxws:handlers** element defines a handler chain in an endpoint's configuration. It can appear as a child to all of the JAX-WS endpoint configuration elements. These are:

- **jaxws:endpoint** configures a service provider.
- **jaxws:server** also configures a service provider.
- **jaxws:client** configures a service consumer.

You add handlers to the handler chain in one of two ways:

- add a **bean** element defining the implementation class
- use a **ref** element to refer to a named **bean** element from elsewhere in the configuration file

The order in which the handlers are defined in the configuration is the order in which they will be executed. The order may be modified if you mix logical handlers and protocol handlers. The run time will sort them into the proper order while maintaining the basic order specified in the configuration.

Example

[Example 43.17, “Configuring an Endpoint to Use a Handler Chain In Spring”](#) shows the configuration for a service provider that loads a handler chain.

Example 43.17. Configuring an Endpoint to Use a Handler Chain In Spring

```
<beans ...  
    xmlns:jaxws="http://cxf.apache.org/jaxws"  
    ...  
    schemaLocation="...  
        http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd  
    ...">  
    <jaxws:endpoint id="HandlerExample"  
        implementor="org.apache.cxf.example.DemoImpl"  
        address="http://localhost:8080/demo">  
        <jaxws:handlers> <bean class="demo.handlers.common.LoggingHandler" /> <bean  
        class="demo.handlers.common.AddHeaderHandler" /> </jaxws:handlers>  
    </jaxws:endpoint>  
</beans>
```

CHAPTER 44. MAVEN TOOLING REFERENCE

44.1. PLUG-IN SETUP

Abstract

before you can use the Apache CXF plug-ins, you must first add the proper dependencies and repositories to your POM.

Dependencies

You need to add the following dependencies to your project's POM:

- the JAX-WS frontend

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxfrt-frontend-jaxws</artifactId>
  <version>version</version>
</dependency>
```

- the HTTP transport

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxfrt-transports-http</artifactId>
  <version>version</version>
</dependency>
```

- the Undertow transport

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxfrt-transports-http-undertow</artifactId>
  <version>version</version>
</dependency>
```

44.2. CXF-CODEGEN-PLUGIN

Abstract

Generates JAX-WS compliant Java code from a WSDL document

Overview

Basic example

The following POM extract shows a simple example of how to configure the Maven **cxf-codegen-plugin** to process the **myService.wsdl** WSDL file:

```
<plugin>
```

```

<groupId>org.apache.cxf</groupId>
<artifactId>cxfrs-codegen-plugin</artifactId>
<version>3.2.7.fuse-750027-redhat-00001</version>
<executions>
  <execution>
    <id>generate-sources</id>
    <phase>generate-sources</phase>
    <configuration>
      <sourceRoot>target/generated/src/main/java</sourceRoot>
      <wsdlOptions>
        <wsdlOption>
          <wsdl>src/main/resources/wsdl/myService.wsdl</wsdl>
        </wsdlOption>
      </wsdlOptions>
    </configuration>
    <goals>
      <goal>wsdl2java</goal>
    </goals>
  </execution>
</executions>
</plugin>

```

Basic configuration settings

In the preceding example, you can customize the following configuration settings

configuration/sourceRoot

Specifies the directory where the generated Java files will be stored. Default is **target/generated-sources/cxf**.

configuration/wsdlOptions/wsdlOption/wsdl

Specifies the location of the WSDL file.

Description

The **wsdl2java** task takes a WSDL document and generates fully annotated Java code from which to implement a service. The WSDL document must have a valid **portType** element, but it does not need to contain a **binding** element or a **service** element. Using the optional arguments you can customize the generated code.

WSDL options

At least one **wsdlOptions** element is required to configure the plug-in. The **wsdlOptions** element's **wsdl** child is required and specifies a WSDL document to be processed by the plug-in. In addition to the **wsdl** element, the **wsdlOptions** element can take a number of children that can customize how the WSDL document is processed.

More than one **wsdlOptions** element can be listed in the plug-in configuration. Each element configures a single WSDL document for processing.

Default options

The **defaultOptions** element is an optional element. It can be used to set options that are used across all of the specified WSDL documents.



IMPORTANT

If an option is duplicated in the **wsdlOptions** element, the value in the **wsdlOptions** element takes precedent.

Specifying code generation options

To specify generic code generation options (corresponding to the switches supported by the Apache CXF **wsdl2java** command-line tool), you can add the **extraargs** element as a child of a **wsdlOption** element. For example, you can add the **-impl** option and the **-verbose** option as follows:

```
...
<configuration>
  <sourceRoot>target/generated/src/main/java</sourceRoot>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
      <!-- you can set the options of wsdl2java command by using the <extraargs> -->
      <extraargs>
        <extraarg>-impl</extraarg>
        <extraarg>-verbose</extraarg>
      </extraargs>
    </wsdlOption>
  </wsdlOptions>
</configuration>
...
```

If a switch takes arguments, you can specify these using subsequent **extraarg** elements. For example, to specify the **jibx** data binding, you can configure the plug-in as follows:

```
...
<configuration>
  <sourceRoot>target/generated/src/main/java</sourceRoot>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
      <extraargs>
        <extraarg>-databinding</extraarg>
        <extraarg>jibx</extraarg>
      </extraargs>
    </wsdlOption>
  </wsdlOptions>
</configuration>
...
```

Specifying binding files

To specify the location of one or more JAX-WS binding files, you can add the **bindingFiles** element as a child of **wsdlOption**—for example:

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
```

```

<wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
<bindingFiles>
  <bindingFile>${basedir}/src/main/resources/wsdl/async_binding.xml</bindingFile>
</bindingFiles>
</wsdlOption>
</wsdlOptions>
</configuration>
...

```

Generating code for a specific WSDL service

To specify the name of the WSDL service for which code is to be generated, you can add the **serviceName** element as a child of **wsdlOption** (the default behaviour is to generate code for every service in the WSDL document)—for example:

```

...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
      <serviceName>MyWSDLService</serviceName>
    </wsdlOption>
  </wsdlOptions>
</configuration>
...

```

Generating code for multiple WSDL files

To generate code for multiple WSDL files, simply insert additional **wsdlOption** elements for the WSDL files. If you want to specify some common options that apply to all of the WSDL files, put the common options into the **defaultOptions** element as shown:

```

<configuration>
  <defaultOptions>
    <bindingFiles>
      <bindingFile>${basedir}/src/main/jaxb/bindings.xml</bindingFile>
    </bindingFiles>
    <noAddressBinding>true</noAddressBinding>
  </defaultOptions>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
      <serviceName>MyWSDLService</serviceName>
    </wsdlOption>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myOtherService.wsdl</wsdl>
      <serviceName>MyOtherWSDLService</serviceName>
    </wsdlOption>
  </wsdlOptions>
</configuration>

```

It is also possible to specify multiple WSDL files using wildcard matching. In this case, specify the directory containing the WSDL files using the **wsdlRoot** element and then select the required WSDL files using an **include** element, which supports wildcarding with the * character. For example, to select all

of the WSDL files ending in **Service.wsdl** from the **src/main/resources/wsdl** root directory, you could configure the plug-in as follows:

```
<configuration>
  <defaultOptions>
    <bindingFiles>
      <bindingFile>${basedir}/src/main/jaxb/bindings.xml</bindingFile>
    </bindingFiles>
    <noAddressBinding>true</noAddressBinding>
  </defaultOptions>
  <wsdlRoot>${basedir}/src/main/resources/wsdl</wsdlRoot>
  <includes>
    <include>*Service.wsdl</include>
  </includes>
</configuration>
```

Downloading WSDL from a Maven repository

To download a WSDL file directly from a Maven repository, add a **wsdlArtifact** element as a child of the **wsdlOption** element and specify the coordinates of the Maven artifact, as follows:

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdlArtifact>
        <groupId>org.apache.pizza</groupId>
        <artifactId>PizzaService</artifactId>
        <version>1.0.0</version>
      </wsdlArtifact>
    </wsdlOption>
  </wsdlOptions>
</configuration>
...
```

Encoding

(Requires JAXB 2.2) To specify the character encoding (Charset) used for the generated Java files, add an **encoding** element as a child of the **configuration** element, as follows:

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
    </wsdlOption>
  </wsdlOptions>
  <encoding>UTF-8</encoding>
</configuration>
...
```

Forking a separate process

You can configure the codegen plug-in to fork a separate JVM for code generation, by adding the **fork** element as a child of the **configuration** element. The fork element can be set to one of the following values:

once

Fork a single new JVM to process all of the WSDL files specified in the codegen plug-in's configuration.

always

Fork a new JVM to process each WSDL file specified in the codegen plug-in's configuration.

false

(Default) Disables forking.

If the codegen plug-in is configured to fork a separate JVM (that is, the **fork** option is set to a non-false value), you can specify additional JVM arguments to the forked JVM through the **additionalJvmArgs** element. For example, the following fragment configures the codegen plug-in to fork a single JVM, which is restricted to access XML schemas from the local file system only (by setting the **javax.xml.accessExternalSchema** system property):

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
    </wsdlOption>
  </wsdlOptions>
  <fork>once</fork>
  <additionalJvmArgs>-Djavax.xml.accessExternalSchema=jar,file</additionalJvmArgs>
</configuration>
...
```

Options reference

The options used to manage the code generation process are reviewed in the following table.

Option	Interpretation
[option]`-fe	-frontend <i>frontend</i> `
Specifies the front end used by the code generator. Possible values are jaxws , jaxws21 , and cxf . The jaxws21 frontend is used to generate JAX-WS 2.1 compliant code. The cxf frontend, which can optionally be used instead of the jaxws frontend, provides an extra constructor for Service classes. This constructor conveniently enables you to specify the Bus instance for configuring the service. Default is jaxws .	[option]`-db

Option	Interpretation
<code>-databinding <i>databinding</i></code>	Specifies the data binding used by the code generator. Possible values are: jaxb , xmlbeans , sdo (sdo-static and sdo-dynamic), and jibx . Default is jaxb .
-wv <i>wsdlVersion</i>	Specifies the WSDL version expected by the tool. Default is 1.1 . ^[a]
-p <i>wsdlNamespace=PackageName</i>	Specifies zero, or more, package names to use for the generated code. Optionally specifies the WSDL namespace to package name mapping.
-b <i>bindingName</i>	Specifies one or more JAXWS or JAXB binding files. Use a separate -b flag for each binding file.
-sn <i>serviceName</i>	Specifies the name of the WSDL service for which code is to be generated. The default is to generate code for every service in the WSDL document.
-reserveClass <i>classname</i>	Used with -autoNameResolution , defines a class names for wsdl-to-java not to use when generating classes. Use this option multiple times for multiple classes.
-catalog <i>catalogUrl</i>	Specifies the URL of an XML catalog to use for resolving imported schemas and WSDL documents.
-d <i>output-directory</i>	Specifies the directory into which the generated code files are written.
-compile	Compiles generated Java files.
-classdir <i>compile-class-dir</i>	Specifies the directory into which the compiled class files are written.
-clientjar <i>jar-file-name</i>	Generates the JAR file that contains all the client classes and the WSDL. The specified wsdlLocation does not work when this option is specified.
-client	Generates starting point code for a client mainline.
-server	Generates starting point code for a server mainline.
-impl	Generates starting point code for an implementation object.

Option	Interpretation
-all	Generates all starting point code: types, service proxy, service interface, server mainline, client mainline, implementation object, and an Ant build.xml file.
-ant	Generates the Ant build.xml file.
-autoNameResolution	Automatically resolve naming conflicts without requiring the use of binding customizations.
-defaultValues = <i>DefaultValueProvider</i>	Instructs the tool to generate default values for the generated client and the generated implementation. Optionally, you can also supply the name of the class used to generate the default values. By default, the RandomValueProvider class is used.
-exclude schema-namespace= <i>java-packagename</i>	Ignore the specified WSDL schema namespace when generating code. This option may be specified multiple times. Also, optionally specifies the Java package name used by types described in the excluded namespace(s).
-exsh (true/false)	Enables or disables processing of extended soap header message binding. Default is false.
-noTypes	Turns off generating types.
-dns (true/false)	Enables or disables the loading of the default namespace package name mapping. Default is true.
-dex (true/false)	Enables or disables the loading of the default excludes namespace mapping. Default is true.
-xjcargs	Specifies a comma separated list of arguments to be passed to directly to the XJC when the JAXB data binding is being used. To get a list of all possible XJC arguments use the -xjc-X .
-noAddressBinding	Instructs the tool to use the Apache CXF proprietary WS-Addressing type instead of the JAX-WS 2.1 compliant mapping.
[option]`-validate [=all	basic
none]`	Instructs the tool to validate the WSDL document before attempting to generate any code.
-keep	Instructs the tool to not overwrite any existing files.

Option	Interpretation
-wsdlLocation wsdlLocation	Specifies the value of the @WebService annotation's wsdlLocation property.
-v	Displays the version number for the tool.
[option]`-verbose	-V`
Displays comments during the code generation process.	-quiet
Suppresses comments during the code generation process.	-allowElementReferences[=true], -aer[=true]
If true , disregards the rule given in section 2.3.1.2(v) of the JAX-WS 2.2 specification disallowing element references when using wrapper-style mapping. Default is false .	-asyncMethods[=method1,method2,...]
List of subsequently generated Java class methods to allow for client-side asynchronous calls; similar to enableAsyncMapping in a JAX-WS binding file.	-bareMethods[=method1,method2,...]
List of subsequently generated Java class methods to have wrapper style (see below), similar to enableWrapperStyle in JAX-WS binding file.	-mimeMethods[=method1,method2,...]
List of subsequently generated Java class methods to enable mime:content mapping, similar to enableMIMEContent in JAX-WS binding file.	-faultSerialVersionUID fault-serialVersionUID
How to generate uid of fault exceptions. Possible values are: NONE , TIMESTAMP , FQCN , or a specific number. Default is NONE .	-encoding encoding
Specifies the Charset encoding to use when generating Java code.	-exceptionSuper
Superclass for fault beans generated from wsdl:fault elements (defaults to java.lang.Exception).	-seiSuper interfaceName
Specifies a base interface for the generated SEI interfaces. For example, this option can be used to add the Java 7 AutoCloseable interface as a super interface.	-mark-generated

Option	Interpretation
[a]	Currently, Apache CXF only provides WSDL 1.1 support for the code generator.

44.3. JAVA2WS

Abstract

generates a WSDL document from Java code

Synopsis

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-java2ws-plugin</artifactId>
  <version>version</version>
  <executions>
    <execution>
      <id>process-classes</id>
      <phase>process-classes</phase>
      <configuration>
        <className>className</className>
        <option>...</option>
        ...
      </configuration>
      <goals>
        <goal>java2ws</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Description

The **java2ws** task takes a service endpoint implementation (SEI) and generates the support files used to implement a Web service. It can generate the following:

- a WSDL document
- the server code needed to deploy the service as a POJO
- client code for accessing the service
- wrapper and fault beans

Required configuration

The plug-in requires that the **className** configuration element is present. The element's value is the fully qualified name of the SEI to be processed.

Optional configuration

The configuration element's listed in the following table can be used to fine tune the WSDL generation.

Element	Description
frontend	Specifies front end to use for processing the SEI and generating the support classes. jaxws is the default. simple is also supported.
databinding	Specifies the data binding used for processing the SEI and generating the support classes. The default when using the JAX-WS front end is jaxb . The default when using the simple frontend is aegis .
genWSDL	Instructs the tool to generate a WSDL document when set to true .
genWrapperbean	Instructs the tool to generate the wrapper bean and the fault beans when set to true .
genClient	Instructs the tool to generate client code when set to true .
genServer	Instructs the tool to generate server code when set to true .
outputFile	Specifies the name of the generated WSDL file.
classpath	Specifies the classpath searched when processing the SEI.
soap12	Specifies that the generated WSDL document is to include a SOAP 1.2 binding when set to true .
targetNamespace	Specifies the target namespace to use in the generated WSDL file.
serviceName	Specifies the value of the generated service element's name attribute.

PART VI. DEVELOPING RESTFUL WEB SERVICES

This guide describes how to use the JAX-RS APIs to implement Web services.

CHAPTER 45. INTRODUCTION TO RESTFUL WEB SERVICES

Abstract

Representational State Transfer (REST) is a software architecture style that centers around the transmission of data over HTTP, using only the four basic HTTP verbs. It also eschews the use of any additional wrappers such as a SOAP envelope and the use of any state data.

OVERVIEW

Representational State Transfer (REST) is an architectural style first described in a doctoral dissertation by a researcher named Roy Fielding. In RESTful systems, servers expose resources using a URI, and clients access these resources using the four HTTP verbs. As clients receive representations of a resource they are placed in a state. When they access a new resource, typically by following a link, they change, or transition, their state. In order to work, REST assumes that resources are capable of being represented using a pervasive standard grammar.

The World Wide Web is the most ubiquitous example of a system designed on REST principles. Web browsers act as clients accessing resources hosted on Web servers. The resources are represented using HTML or XML grammars that all Web browsers can consume. The browsers can also easily follow the links to new resources.

The advantages of RESTful systems is that they are highly scalable and highly flexible. Because the resources are accessed and manipulated using the four HTTP verbs, the resources are exposed using a URIs, and the resources are represented using standard grammars, clients are not as affected by changes to the servers. Also, RESTful systems can take full advantage of the scalability features of HTTP such as caching and proxies.

BASIC REST PRINCIPLES

RESTful architectures adhere to the following basic principles:

- Application state and functionality are divided into resources.
- Resources are addressable using standard URIs that can be used as hypermedia links.
- All resources use only the four HTTP verbs.
 - **DELETE**
 - **GET**
 - **POST**
 - **PUT**
- All resources provide information using the MIME types supported by HTTP.
- The protocol is stateless.
- Responses are cacheable.
- The protocol is layered.

RESOURCES

Resources are central to REST. A resource is a source of information that can be addressed using a URI. In the early days of the Web, resources were largely static documents. In the modern Web, a resource can be any source of information. For example a Web service can be a resource if it can be accessed using a URI.

RESTful endpoints exchange *representations* of the resources they address. A representation is a document containing the data provided by the resource. For example, the method of a Web service that provides access to a customer record would be a resource, the copy of the customer record exchanged between the service and the consumer is a representation of the resource.

REST BEST PRACTICES

When designing RESTful Web services it is helpful to keep in mind the following:

- Provide a distinct URI for each resource you wish to expose.
For example, if you are building a system that deals with driving records, each record should have a unique URI. If the system also provides information on parking violations and speeding fines, each type of resource should also have a unique base. For example, speeding fines could be accessed through `/speedingfines/driverID` and parking violations could be accessed through `/parkingfines/driverID`.
- Use nouns in your URIs.
Using nouns highlights the fact that resources are things and not actions. URIs such as `/ordering` imply an action, whereas `/orders` implies a thing.
- Methods that map to **GET** should not change any data.
- Use links in your responses.
Putting links to other resources in your responses makes it easier for clients to follow a chain of data. For example, if your service returns a collection of resources, it would be easier for a client to access each of the individual resources using the provided links. If links are not included, a client needs to have additional logic to follow the chain to a specific node.
- Make your service stateless.
Requiring the client or the service to maintain state information forces a tight coupling between the two. Tight couplings make upgrading and migrating more difficult. Maintaining state can also make recovery from communication errors more difficult.

DESIGNING A RESTFUL WEB SERVICE

Regardless of the framework you use to implement a RESTful Web service, there are a number of steps that should be followed:

1. Define the resources the service will expose.

In general, a service will expose one or more resources that are organized as a tree. For example, a driving record service could be organized into three resources:

- `/license/driverID`
- `/license/driverID/speedingfines`
- `/license/driverID/parkingfines`

2. Define what actions you want to be able to perform on each resource.
For example, you may want to be able to update a diver's address or remove a parking ticket from a driver's record.
3. Map the actions to the appropriate HTTP verbs.

Once you have defined the service, you can implement it using Apache CXF.

IMPLEMENTING REST WITH APACHE CXF

Apache CXF provides an implementation of the *Java API for RESTful Web Services* (JAX-RS). JAX-RS provides a standardized way to map POJOs to resources using annotations.

When moving from the abstract service definition to a RESTful Web service implemented using JAX-RS, you need to do the following:

1. Create a root resource class for the resource that represents the top of the service's resource tree.
See [Section 46.3, "Root resource classes"](#).
2. Map the service's other resources into sub-resources.
See [Section 46.5, "Working with sub-resources"](#).
3. Create methods to implement each of the HTTP verbs used by each of the resources.
See [Section 46.4, "Working with resource methods"](#).



NOTE

Apache CXF continues to support the old HTTP binding to map Java interfaces into RESTful Web services. The HTTP binding provides basic functionality and has a number of limitations. Developers are encouraged to update their applications to use JAX-RS.

DATA BINDINGS

By default, Apache CXF uses Java Architecture for XML Binding (JAXB) objects to map the resources and their representations to Java objects. Provides clean, well defined mappings between Java objects and XML elements.

The Apache CXF JAX-RS implementation also supports exchanging data using *JavaScript Object Notation* (JSON). JSON is a popular data format used by Ajax developers. The marshaling of data between JSON and JAXB is handled by the Apache CXF runtime.

CHAPTER 46. CREATING RESOURCES

Abstract

In RESTful Web services all requests are handled by resources. The JAX-RS APIs implement resources as a Java class. A resource class is a Java class that is annotated with one, or more, JAX-RS annotations. The core of a RESTful Web service implemented using JAX-RS is a root resource class. The root resource class is the entry point to the resource tree exposed by a service. It may handle all requests itself, or it may provide access to sub-resources that handle requests.

46.1. INTRODUCTION

Overview

RESTful Web services implemented using JAX-RS APIs provide responses as representations of a resource implemented by Java classes. A *resource class* is a class that uses JAX-RS annotations to implement a resource. For most RESTful Web services, there is a collection of resources that need to be accessed. The resource class' annotations provide information such as the URI of the resources and which HTTP verb each operation handles.

Types of resources

The JAX-RS APIs allow you to create two basic types of resources:

- A [Section 46.3, "Root resource classes"](#) is the entry point to a service's resource tree. It is decorated with the **@Path** annotation to define the base URI for the resources in the service.
- [Section 46.5, "Working with sub-resources"](#) are accessed through the root resource. They are implemented by methods that are decorated with the **@Path** annotation. A sub-resource's **@Path** annotation defines a URI relative to the base URI of a root resource.

Example

[Example 46.1, "Simple resource class"](#) shows a simple resource class.

Example 46.1. Simple resource class

```
package demo.jaxrs.server;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;

@Path("/customerservice")
public class CustomerService
{
    public CustomerService()
    {

    }

    @GET
    public Customer getCustomer(@QueryParam("id") String id)
    {
```

```

    ...
}

...
}

```

Two items make the class defined in [Example 46.1, “Simple resource class”](#) a resource class:

The **@Path** annotation specifies the base URI for the resource.

The **@GET** annotation specifies that the method implements the HTTP **GET** method for the resource.

46.2. BASIC JAX-RS ANNOTATIONS

Overview

The most basic pieces of information required by a RESTful Web service implementation are:

- the URI of the service's resources
- how the class' methods are mapped to the HTTP verbs

JAX-RS defines a set of annotations that provide this basic information. All resource classes must have at least one of these annotations.

Setting the path

The **@Path** annotation specifies the URI of a resource. The annotation is defined by the javax.ws.rs.Path interface and it can be used to decorate either a resource class or a resource method. It takes a string value as its only parameter. The string value is a URI template that specifies the location of an implemented resource.

The URI template specifies a relative location for the resource. As shown in [Example 46.2, “URI template syntax”](#), the template can contain the following:

- unprocessed path components
- parameter identifiers surrounded by { }



NOTE

Parameter identifiers can include regular expressions to alter the default path processing.

Example 46.2. URI template syntax

```
@Path("resourceName/{param1}../{paramN}")
```

For example, the URI template **widgets/{color}/{number}** would map to **widgets/blue/12**. The value of the **color** parameter is assigned to **blue**. The value of the **number** parameter is assigned **12**.

How the URI template is mapped to a complete URI depends on what the **@Path** annotation is decorating. If it is placed on a root resource class, the URI template is the root URI of all resources in the tree and it is appended directly to the URI at which the service is published. If the annotation decorates a sub-resource, it is relative to the root resource URI.

Specifying HTTP verbs

JAX-RS uses five annotations for specifying the HTTP verb that will be used for a method:

- javax.ws.rs.DELETE specifies that the method maps to a **DELETE**.
- javax.ws.rs.GET specifies that the method maps to a **GET**.
- javax.ws.rs.POST specifies that the method maps to a **POST**.
- javax.ws.rs.PUT specifies that the method maps to a **PUT**.
- javax.ws.rs.HEAD specifies that the method maps to a **HEAD**.

When you map your methods to HTTP verbs, you must ensure that the mapping makes sense. For example, if you map a method that is intended to submit a purchase order, you would map it to a **PUT** or a **POST**. Mapping it to a **GET** or a **DELETE** would result in unpredictable behavior.

46.3. ROOT RESOURCE CLASSES

Overview

A root resource class is the entry point into a JAX-RS implemented RESTful Web service. It is decorated with a **@Path** that specifies the root URI of the resources implemented by the service. Its methods either directly implement operations on the resource or provide access to sub-resources.

Requirements

In order for a class to be a root resource class it must meet the following criteria:

- The class must be decorated with the **@Path** annotation.
The specified path is the root URI for all of the resources implemented by the service. If the root resource class specifies that its path is **widgets** and one of its methods implements the **GET** verb, then a **GET** on **widgets** invokes that method. If a sub-resource specifies that its URI is **{id}**, then the full URI template for the sub-resource is **widgets/{id}** and it will handle requests made to URLs like **widgets/12** and **widgets/42**.
- The class must have a public constructor for the runtime to invoke.
The runtime must be able to provide values for all of the constructor's parameters. The constructor's parameters can include parameters decorated with the JAX-RS parameter annotations. For more information on the parameter annotations see [Chapter 47, Passing Information into Resource Classes and Methods](#).
- At least one of the classes methods must either be decorated with an HTTP verb annotation or the **@Path** annotation.

Example

[Example 46.3, "Root resource class"](#) shows a root resource class that provides access to a sub-resource.

Example 46.3. Root resource class

```
package demo.jaxrs.server;

import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Response;

@Path("/customerservice/")
public class CustomerService
{
    public CustomerService()
    {
        ...
    }

    @GET
    public Customer getCustomer(@QueryParam("id") String id)
    {
        ...
    }

    @DELETE
    public Response deleteCustomer(@QueryParam("id") String id)
    {
        ...
    }

    @PUT
    public Response updateCustomer(Customer customer)
    {
        ...
    }

    @POST
    public Response addCustomer(Customer customer)
    {
        ...
    }

    @Path("/orders/{orderId}/")
    public Order getOrder(@PathParam("orderId") String orderId)
    {
        ...
    }
}
```

The class in [Example 46.3, “Root resource class”](#) meets all of the requirements for a root resource class.

The class is decorated with the **@Path** annotation. The root URI for the resources exposed by the service is **customerservice**.

The class has a public constructor. In this case the no argument constructor is used for simplicity.

The class implements each of the four HTTP verbs for the resource.

The class also provides access to a sub-resource through the **getOrder()** method. The URI for the sub-resource, as specified using the the **@Path** annotation, is **customerservice/order/id**. The sub-resource is implemented by the **Order** class.

For more information on implementing sub-resources see [Section 46.5, "Working with sub-resources"](#).

46.4. WORKING WITH RESOURCE METHODS

Overview

Resource methods are annotated using JAX-RS annotations. They have one of the HTTP method annotation specifying the types of requests that the method processes. JAX-RS places several constraints on resource methods.

General constraints

All resource methods must meet the following conditions:

- It must be public.
- It must be decorated with one of the HTTP method annotations described in [the section called "Specifying HTTP verbs"](#).
- It must not have more than one entity parameter as described in [the section called "Parameters"](#).

Parameters

Resource method parameters take two forms:

- **entity parameters**—Entity parameters are not annotated. Their value is mapped from the request entity body. An entity parameter can be of any type for which your application has an entity provider. Typically they are JAXB objects.



IMPORTANT

A resource method can have **only one** entity parameter.

For more information on entity providers see [Chapter 51, Entity Support](#).

- **annotated parameters**—Annotated parameters use one of the JAX-RS annotations that specify how the value of the parameter is mapped from the request. Typically, the value of the parameter is mapped from portions of the request URI.
For more information about using the JAX-RS annotations for mapping request data to method parameters see [Chapter 47, Passing Information into Resource Classes and Methods](#).

[Example 46.4, “Resource method with a valid parameter list”](#) shows a resource method with a valid parameter list.

Example 46.4. Resource method with a valid parameter list

```
@POST
@Path("disaster/monster/giant/{id}")
public void addDaikaiju(Kaiju kaiju,
                        @PathParam("id") String id)
{
...
}
```

[Example 46.5, “Resource method with an invalid parameter list”](#) shows a resource method with an invalid parameter list. It has two parameters that are not annotated.

Example 46.5. Resource method with an invalid parameter list

```
@POST
@Path("disaster/monster/giant/")
public void addDaikaiju(Kaiju kaiju,
                        String id)
{
...
}
```

Return values

Resource methods can return one of the following:

- **void**
- any Java class for which the application has an entity provider
For more information on entity providers see [Chapter 51, Entity Support](#).
- a **Response** object
For more information on **Response** objects see [Section 48.3, “Fine tuning an application’s responses”](#).
- a **GenericEntity<T>** object
For more information on **GenericEntity<T>** objects see [Section 48.4, “Returning entities with generic type information”](#).

All resource methods return an HTTP status code to the requester. When the return type of the method is **void** or the value being returned is **null**, the resource method sets the HTTP status code to **204**. When the resource method returns any value other than **null**, it sets the HTTP status code to **200**.

46.5. WORKING WITH SUB-RESOURCES

Overview

It is likely that a service will need to be handled by more than one resource. For example, in an order processing service best-practices suggests that each customer would be handled as a unique resource. Each order would also be handled as a unique resource.

Using the JAX-RS APIs, you would implement the customer resources and the order resources as *sub-resources*. A sub-resource is a resource that is accessed through a root resource class. They are defined by adding a **@Path** annotation to a resource class' method. Sub-resources can be implemented in one of two ways:

- **Sub-resource method**—directly implements an HTTP verb for a sub-resource and is decorated with one of the annotations described in [the section called "Specifying HTTP verbs"](#).
- **Sub-resource locator**—points to a class that implements the sub-resource.

Specifying a sub-resource

Sub-resources are specified by decorating a method with the **@Path** annotation. The URI of the sub-resource is constructed as follows:

1. Append the value of the sub-resource's **@Path** annotation to the value of the sub-resource's parent resource's **@Path** annotation.
The parent resource's **@Path** annotation maybe located on a method in a resource class that returns an object of the class containing the sub-resource.
2. Repeat the previous step until the root resource is reached.
3. The assembled URI is appended to the base URI at which the service is deployed.

For example the URI of the sub-resource shown in [Example 46.6, "Order sub-resource"](#) could be `baseURI/customerservice/order/12`.

Example 46.6. Order sub-resource

```
...
@Path("/customerservice/")
public class CustomerService
{
    ...
    @Path("/orders/{orderId}/")
    @GET
    public Order getOrder(@PathParam("orderId") String orderId)
    {
        ...
    }
}
```

Sub-resource methods

A sub-resource method is decorated with both a **@Path** annotation and one of the HTTP verb annotations. The sub-resource method is directly responsible for handling a request made on the resource using the specified HTTP verb.

[Example 46.7, "Sub-resource methods"](#) shows a resource class with three sub-resource methods:

- **getOrder()** handles HTTP **GET** requests for resources whose URI matches `/customerservice/orders/{orderId}/`.
- **updateOrder()** handles HTTP **PUT** requests for resources whose URI matches `/customerservice/orders/{orderId}/`.
- **newOrder()** handles HTTP **POST** requests for the resource at `/customerservice/orders/`.

Example 46.7. Sub-resource methods

```
...
@Path("/customerservice/")
public class CustomerService
{
    ...
    @Path("/orders/{orderId}/")
    @GET
    public Order getOrder(@PathParam("orderId") String orderId)
    {
        ...
    }

    @Path("/orders/{orderId}/")
    @PUT
    public Order updateOrder(@PathParam("orderId") String orderId,
                            Order order)
    {
        ...
    }

    @Path("/orders/")
    @POST
    public Order newOrder(Order order)
    {
        ...
    }
}
```



NOTE

Sub-resource methods with the same URI template are equivalent to resource class returned by a sub-resource locator.

Sub-resource locators

Sub-resource locators are not decorated with one of the HTTP verb annotations and do not directly handle a request on the sub-resource. Instead, a sub-resource locator returns an instance of a resource class that can handle the request.

In addition to not having an HTTP verb annotation, sub-resource locators also cannot have any entity parameters. All of the parameters used by a sub-resource locator method must use one of the annotations described in [Chapter 47, Passing Information into Resource Classes and Methods](#).

As shown in [Example 46.8, "Sub-resource locator returning a specific class"](#), sub-resource locator allows

you to encapsulate a resource as a reusable class instead of putting all of the methods into one super class. The **processOrder()** method is a sub-resource locator. When a request is made on a URI matching the URI template `/orders/{orderId}/` it returns an instance of the **Order** class. The **Order** class has methods that are decorated with HTTP verb annotations. A **PUT** request is handled by the **updateOrder()** method.

Example 46.8. Sub-resource locator returning a specific class

```
...
@Path("/customerservice/")
public class CustomerService
{
    ...
    @Path("/orders/{orderId}/")
    public Order processOrder(@PathParam("orderId") String orderId)
    {
        ...
    }
    ...
}

public class Order
{
    ...
    @GET
    public Order getOrder(@PathParam("orderId") String orderId)
    {
        ...
    }

    @PUT
    public Order updateOrder(@PathParam("orderId") String orderId,
                           Order order)
    {
        ...
    }
}
```

Sub-resource locators are processed at runtime so that they can support polymorphism. The return value of a sub-resource locator can be a generic **Object**, an abstract class, or the top of a class hierarchy. For example, if your service needed to process both PayPal orders and credit card orders, the **processOrder()** method's signature from [Example 46.8, "Sub-resource locator returning a specific class"](#) could remain unchanged. You would simply need to implement two classes, **ppOrder** and **ccOrder**, that extended the **Order** class. The implementation of **processOrder()** would instantiate the desired implementation of the sub-resource based on what ever logic is required.

46.6. RESOURCE SELECTION METHOD

Overview

It is possible for a given URI to map to one or more resource methods. For example the URI

`customerservice/12/ma` could match the templates `@Path("customerservice/{id}")` or `@Path("customerservice/{id}/{state}")`. JAX-RS specifies a detailed algorithm for matching a resource method to a request. The algorithm compares the normalized URI, the HTTP verb, and the media types of the request and response entities to the annotations on the resource classes.

The basic selection algorithm

The JAX-RS selection algorithm is broken down into three stages:

1. Determine the root resource class.

The request URI is matched against all of the classes decorated with the `@Path` annotation. The classes whose `@Path` annotation matches the request URI are determined.

If the value of the resource class' `@Path` annotation matches the entire request URI, the class' methods are used as input into the third stage.

2. Determine the object will handle the request.

If the request URI is longer than the value of the selected class' `@Path` annotation, the values of the resource methods' `@Path` annotations are used to look for a sub-resource that can process the request.

If one or more sub-resource methods match the request URI, these methods are used as input for the third stage.

If the only matches for the request URI are sub-resource locators, the resource methods of the object created by the sub-resource locator are used to match the request URI. This stage is repeated until a sub-resource method matches the request URI.

3. Select the resource method that will handle the request.

The resource method whose HTTP verb annotation matches the HTTP verb in the request. In addition, the selected resource method must accept the media type of the request entity body and be capable of producing a response that conforms to the media type(s) specified in the request.

Selecting from multiple resource classes

The first two stages of the selection algorithm determine the resource that will handle the request. In some cases the resource is implemented by a resource class. In other cases, it is implemented by one or more sub-resources that use the same URI template. When there are multiple resources that match a request URI, resource classes are preferred over sub-resources.

If more than one resource still matches the request URI after sorting between resource classes and sub-resources, the following criteria are used to select a single resource:

1. Prefer the resource with the most literal characters in its URI template.

Literal characters are characters that are not part of a template variable. For example, `/widgets/{id}/{color}` has ten literal characters and `/widgets/1/{color}` has eleven literal characters. So, the request URI `/widgets/1/red` would be matched to the resource with `/widgets/1/{color}` as its URI template.



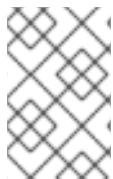
NOTE

A trailing slash (/) counts as a literal character. So `/joefred/` will be preferred over `/joefred`.

2. Prefer the resource with the most variables in its URI template.
The request URI `/widgets/30/green` could match both `/widgets/{id}/{color}` and `/widgets/{amount}/`. However, the resource with the URI template `/widgets/{id}/{color}` will be selected because it has two variables.
3. Prefer the resource with the most variables containing regular expressions.
The request URI `/widgets/30/green` could match both `/widgets/{number}/{color}` and `/widgets/{id:}/{color}*`. However, the resource with the URI template `*/widgets/{id:}/{color}` will be selected because it has a variable containing a regular expression.

Selecting from multiple resource methods

In many cases, selecting a resource that matches the request URI results in a single resource method that can process the request. The method is determined by matching the HTTP verb specified in the request with a resource method's HTTP verb annotation. In addition to having the appropriate HTTP verb annotation, the selected method must also be able to handle the request entity included in the request and be able to produce the proper type of response specified in the request's metadata.



NOTE

The type of request entity a resource method can handle is specified by the **@Consumes** annotation. The type of responses a resource method can produce are specified using the **@Produces** annotation.

When selecting a resource produces multiple methods that can handle a request the following criteria is used to select the resource method that will handle the request:

1. Prefer resource methods over sub-resources.
2. Prefer sub-resource methods over sub-resource locators.
3. Prefer methods that use the most specific values in the **@Consumes** annotation and the **@Produces** annotation.
For example, a method that has the annotation `@Consumes("text/xml")` would be preferred over a method that has the annotation `@Consumes("text/*")`. Both methods would be preferred over a method without an **@Consumes** annotation or the annotation `@Consumes("*/")`.
4. Prefer methods that most closely match the content type of the request body entity.



NOTE

The content type of the request body entity is specified in the HTTP **Content-Type** property.

5. Prefer methods that most closely match the content type accepted as a response.



NOTE

The content types accepted as a response are specified in the HTTP **Accept** property.

Customizing the selection process

In some cases, developers have reported the algorithm being somewhat restrictive in the way multiple resource classes are selected. For example, if a given resource class has been matched and if this class has no matching resource method, then the algorithm stops executing. It never checks the remaining matching resource classes.

Apache CXF provides the org.apache.cxf.jaxrs.ext.ResourceComparator interface which can be used to customize how the runtime handles multiple matching resource classes. The ResourceComparator interface, shown in [Example 46.9, “Interface for customizing resource selection”](#), has two methods that need to be implemented. One compares two resource classes and the other compares two resource methods.

Example 46.9. Interface for customizing resource selection

```
package org.apache.cxf.jaxrs.ext;

import org.apache.cxf.jaxrs.model.ClassResourceInfo;
import org.apache.cxf.jaxrs.model.OperationResourceInfo;
import org.apache.cxf.message.Message;

public interface ResourceComparator
{
    int compare(ClassResourceInfo cri1,
               ClassResourceInfo cri2,
               Message message);

    int compare(OperationResourceInfo oper1,
               OperationResourceInfo oper2,
               Message message);
}
```

Custom implementations select between the two resources as follows:

- Return **1** if the first parameter is a better match than the second parameter
- Return **-1** if the second parameter is a better match than the first parameter

If **0** is returned then the runtime will proceed with the default selection algorithm

You register a custom ResourceComparator implementation by adding a **resourceComparator** child to the service's **jaxrs:server** element.

CHAPTER 47. PASSING INFORMATION INTO RESOURCE CLASSES AND METHODS

Abstract

JAX-RS specifies a number of annotations that allow the developer to control where the information passed into resources come from. The annotations conform to common HTTP concepts such as matrix parameters in a URI. The standard APIs allow the annotations to be used on method parameters, bean properties, and resource class fields. Apache CXF provides an extension that allows for the injection of a sequence of parameters to be injected into a bean.

47.1. BASICS OF INJECTING DATA

Overview

Parameters, fields, and bean properties that are initialized using data from the HTTP request message have their values injected into them by the runtime. The specific data that is injected is specified by a set of annotations described in [Section 47.2, “Using JAX-RS APIs”](#).

The JAX-RS specification places a few restrictions on when the data is injected. It also places a few restrictions on the types of objects into which request data can be injected.

When data is injected

Request data is injected into objects when they are instantiated due to a request. This means that only objects that directly correspond to a resource can use the injection annotations. As discussed in [Chapter 46, Creating Resources](#), these objects will either be a root resource decorated with the `@Path` annotation or an object returned from a sub-resource locator method.

Supported data types

The specific set of data types that data can be injected into depends on the annotation used to specify the source of the injected data. However, all of the injection annotations support at least the following set of data types:

- primitives such as `int`, `char`, or `long`
- Objects that have a constructor that accepts a single `String` argument
- Objects that have a static `valueOf()` method that accepts a single `String` argument
- `List<T>`, `Set<T>`, or `SortedSet<T>` objects where `T` satisfies the other conditions in the list



NOTE

Where injection annotations have different requirements for supported data types, the differences will be highlighted in the discussion of the annotation.

47.2. USING JAX-RS APIs

47.2.1. JAX-RS Annotation Types

The standard JAX-RS API specifies annotations that can be used to inject values into fields, bean properties, and method parameters. The annotations can be split up into three distinct types:

- [Section 47.2.2, "Injecting data from a request URI"](#)
- [Section 47.2.3, "Injecting data from the HTTP message header"](#)
- [Section 47.2.4, "Injecting data from HTML forms"](#)

47.2.2. Injecting data from a request URI

Overview

One of the best practices for designing a RESTful Web service is that each resource should have a unique URI. A developer can use this principle to provide a good deal of information to the underlying resource implementation. When designing URI templates for a resource, a developer can build the templates to include parameter information that can be injected into the resource implementation. Developers can also leverage query and matrix parameters for feeding information into the resource implementations.

Getting data from the URI's path

One of the more common mechanisms for getting information about a resource is through the variables used in creating the URI templates for a resource. This is accomplished using the **javax.ws.rsPathParam** annotation. The **@PathParam** annotation has a single parameter that identifies the URI template variable from which the data will be injected.

In [Example 47.1, "Injecting data from a URI template variable"](#) the **@PathParam** annotation specifies that the value of the URI template variable **color** is injected into the **itemColor** field.

Example 47.1. Injecting data from a URI template variable

```
import javax.ws.rs.Path;
import javax.ws.rsPathParam
...
@Path("/boxes/{shape}/{color}")
class Box
{
    ...
    @PathParam("color")
    String itemColor;
    ...
}
```

The data types supported by the **@PathParam** annotation are different from the ones described in [the section called "Supported data types"](#). The entity into which the **@PathParam** annotation injects data must be of one of the following types:

- PathSegment
The value will be the final segment of the matching part of the path.

- `List<PathSegment>`
The value will be a list of `PathSegment` objects corresponding to the path segment(s) that matched the named template parameter.
- primitives such as `int`, `char`, or `long`
- Objects that have a constructor that accepts a single `String` argument
- Objects that have a static `valueOf()` method that accepts a single `String` argument

Using query parameters

A common way of passing information on the Web is to use *query parameters* in a URI. Query parameters appear at the end of the URI and are separated from the resource location portion of the URI by a question mark(`?`). They consist of one, or more, name value pairs where the name and value are separated by an equal sign(`=`). When more than one query parameter is specified, the pairs are separated from each other by either a semicolon(`;`) or an ampersand(`&`). [Example 47.2, “URI with a query string”](#) shows the syntax of a URI with query parameters.

Example 47.2. URI with a query string

```
http://fusesource.org?name=value;name2=value2;...
```



NOTE

You can use **either** the semicolon or the ampersand to separate query parameters, but not both.

The `javax.ws.rs.QueryParam` annotation extracts the value of a query parameter and injects it into a JAX-RS resource. The annotation takes a single parameter that identifies the name of the query parameter from which the value is extracted and injected into the specified field, bean property, or parameter. The `@QueryParam` annotation supports the types described in [the section called “Supported data types”](#).

[Example 47.3, “Resource method using data from a query parameter”](#) shows a resource method that injects the value of the query parameter `id` into the method’s `id` parameter.

Example 47.3. Resource method using data from a query parameter

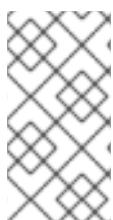
```
import javax.ws.rsQueryParam;
import javax.ws.rsPathParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...
@Path("/monstersforhire/")
public class MonsterService
{
    ...
    @POST
    @Path("/{type}")
    public void updateMonster(@PathParam("type") String type,
                            @QueryParam("id") String id)
```

```
{
...
}
...
}
```

To process an HTTP **POST** to `/monstersforhire/daikaiju?id=jonas` the **updateMonster()** method's **type** is set to **daikaiju** and the **id** is set to **jonas**.

Using matrix parameters

URI matrix parameters, like URI query parameters, are name/value pairs that can provide additional information selecting a resource. Unlike query parameters, matrix parameters can appear anywhere in a URI and they are separated from the hierarchical path segments of the URI using a semicolon(;).
`/mostersforhire/daikaiju;id=jonas` has one matrix parameter called **id** and
`/monstersforhire/japan;type=daikaiju/flying;wingspan=40` has two matrix parameters called **type** and **wingspan**.



NOTE

Matrix parameters are not evaluated when computing a resource's URI. So, the URI used to locate the proper resource to handle the request URI
`/monstersforhire/japan;type=daikaiju/flying;wingspan=40` is
`/monstersforhire/japan/flying`.

The value of a matrix parameter is injected into a field, parameter, or bean property using the **javax.ws.rs.MatrixParam** annotation. The annotation takes a single parameter that identifies the name of the matrix parameter from which the value is extracted and injected into the specified field, bean property, or parameter. The **@MatrixParam** annotation supports the types described in [the section called "Supported data types"](#).

[Example 47.4, "Resource method using data from matrix parameters"](#) shows a resource method that injects the value of the matrix parameters **type** and **id** into the method's parameters.

Example 47.4. Resource method using data from matrix parameters

```
import javax.ws.rs.MatrixParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...
@Path("/monstersforhire/")
public class MonsterService
{
    ...
    @POST
    public void updateMonster(@MatrixParam("type") String type,
                             @MatrixParam("id") String id)
    {
        ...
    }
    ...
}
```



To process an HTTP **POST** to `/monstersforhire;type=daikaiju;id=whale` the **updateMonster()** method's **type** is set to **daikaiju** and the **id** is set to **whale**.



NOTE

JAX-RS evaluates all of the matrix parameters in a URI at once, so it cannot enforce constraints on a matrix parameters location in a URI. For example `/monstersforhire/japan;type=daikaiju/flying;wingspan=40`, `/monstersforhire/japan/flying;type=daikaiju;wingspan=40`, and `/monstersforhire/japan;type=daikaiju;wingspan=40/flying` are all treated as equivalent by a RESTful Web service implemented using the JAX-RS APIs.

Disabling URI decoding

By default all request URIs are decoded. So the URI `/monster/night%20stalker` and the URI `/monster/night stalker` are equivalent. The automatic URI decoding makes it easy to send characters outside of the ASCII character set as parameters.

If you do not wish to have URI automatically decoded, you can use the **javax.ws.rs.Encoded** annotation to deactivate the URI decoding. The annotation can be used to deactivate URI decoding at the following levels:

- class level—Decorating a class with the **@Encoded** annotation deactivates the URI decoding for all parameters, field, and bean properties in the class.
- method level—Decorating a method with the **@Encoded** annotation deactivates the URI decoding for all parameters of the class.
- parameter/field level—Decorating a parameter or field with the **@Encoded** annotation deactivates the URI decoding for all parameters of the class.

[Example 47.5, "Disabling URI decoding"](#) shows a resource whose **getMonster()** method does not use URI decoding. The **addMonster()** method only disables URI decoding for the **type** parameter.

Example 47.5. Disabling URI decoding

```

@Path("/monstersforhire/")
public class MonsterService
{
    ...
    ...
    @GET
    @Encoded
    @Path("/{type}")
    public Monster getMonster(@PathParam("type") String type,
                             @QueryParam("id") String id)
    {
        ...
    }
    ...
    ...
}

@PUT
@Path("/{id}")
public void addMonster(@Encoded @PathParam("type") String type,
                      @QueryParam("id") String id)

```

```

{
...
}
...
}
```

Error handling

If an error occurs when attempting to inject data using one of the URI injection annotations a `WebApplicationException` exception wrapping the original exception is generated. The `WebApplicationException` exception's status is set to **404**.

47.2.3. Injecting data from the HTTP message header

Overview

In normal usage the HTTP headers in a request message pass along generic information about the message, how it is to be handled in transit, and details about the expected response. While a few standard headers are commonly recognized and used, the HTTP specification allows for any name/value pair to be used as an HTTP header. The JAX-RS APIs provide an easy mechanism for injecting HTTP header information into a resource implementation.

One of the most commonly used HTTP headers is the cookie. Cookies allow HTTP clients and servers to share static information across multiple request/response sequences. The JAX-RS APIs provide an annotation inject data directly from a cookie into a resource implementation.

Injecting information from the HTTP headers

The **`javax.ws.rs.HeaderParam`** annotation is used to inject the data from an HTTP header field into a parameter, field, or bean property. It has a single parameter that specifies the name of the HTTP header field from which the value is extracted and injected into the resource implementation. The associated parameter, field, or bean property must conform to the data types described in [the section called "Supported data types"](#).

[Injecting the If-Modified-Since header](#) shows code for injecting the value of the HTTP **If-Modified-Since** header into a class' `oldestDate` field.

Injecting the If-Modified-Since header

```

import javax.ws.rs.HeaderParam;
...
class RecordKeeper
{
...
    @HeaderParam("If-Modified-Since")
    String oldestDate;
...
}
```

Injecting information from a cookie

Cookies are a special type of HTTP header. They are made up of one or more name/value pairs that are

passed to the resource implementation on the first request. After the first request, the cookie is passed back and forth between the provider and consumer with each message. Only the consumer, because they generate requests, can change the cookie. Cookies are commonly used to maintain session across multiple request/response sequences, storing user settings, and other data that can persist.

The **javax.ws.rs.CookieParam** annotation extracts the value from a cookie's field and injects it into a resource implementation. It takes a single parameter that specifies the name of the cookie's field from which the value is to be extracted. In addition to the data types listed in [the section called "Supported data types"](#), entities decorated with the **@CookieParam** can also be a **Cookie** object.

[Example 47.6, "Injecting a cookie"](#) shows code for injecting the value of the **handle** cookie into a field in the **CB** class.

Example 47.6. Injecting a cookie

```
import javax.ws.rs.CookieParam;
...
class CB
{
...
    @CookieParam("handle")
    String handle;
...
}
```

Error handling

If an error occurs when attempting to inject data using one of the HTTP message injection annotations a **WebApplicationException** exception wrapping the original exception is generated. The **WebApplicationException** exception's status is set to **400**.

47.2.4. Injecting data from HTML forms

Overview

HTML forms are an easy means of getting information from a user and they are also easy to create. Form data can be used for HTTP **GET** requests and HTTP **POST** requests:

GET

When form data is sent as part of an HTTP **GET** request the data is appended to the URI as a set of query parameters. Injecting data from query parameters is discussed in [the section called "Using query parameters"](#).

POST

When form data is sent as part of an HTTP **POST** request the data is placed in the HTTP message body. The form data can be handled using a regular entity parameter that supports the form data. It can also be handled by using the **@FormParam** annotation to extract the data and inject the pieces into resource method parameters.

Using the **@FormParam** annotation to inject form data

The **javax.ws.rs.FormParam** annotation extracts field values from form data and injects the value into resource method parameters. The annotation takes a single parameter that specifies the key of the field

from which it extracts the values. The associated parameter must conform to the data types described in [the section called “Supported data types”](#).



IMPORTANT

The JAX-RS API Javadoc states that the **@FormParam** annotation can be placed on fields, methods, and parameters. However, the **@FormParam** annotation is only meaningful when placed on resource method parameters.

Example

[Injecting form data into resource method parameters](#) shows a resource method that injects form data into its parameters. The method assumes that the client’s form includes three fields—**title**, **tags**, and **body**—that contain string data.

Injecting form data into resource method parameters

```
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;

...
@POST
public boolean updatePost(@FormParam("title") String title,
                          @FormParam("tags") String tags,
                          @FormParam("body") String post)
{
    ...
}
```

47.2.5. Specifying a default value to inject

Overview

To provide for a more robust service implementation, you may want to ensure that any optional parameters can be set to a default value. This can be particularly useful for values that are taken from query parameters and matrix parameters since entering long URI strings is highly error prone. You may also want to set a default value for a parameter extracted from a cookie since it is possible for a requesting system not have the proper information to construct a cookie with all the values.

The **javax.ws.rs.DefaultValue** annotation can be used in conjunction with the following injection annotations:

- **@PathParam**
- **@QueryParam**
- **@MatrixParam**
- **@FormParam**
- **@HeaderParam**
- **@CookieParam**

The **@DefaultValue** annotation specifies a default value to be used when the data corresponding to the injection annotation is not present in the request.

Syntax

[Syntax for setting the default value of a parameter](#) shows the syntax for using the **@DefaultValue** annotation.

Syntax for setting the default value of a parameter

```
import javax.ws.rs.DefaultValue;
...
void resourceMethod(@MatrixParam("matrix")
    @DefaultValue("value")
    int someValue, ... )
...
...
```

The annotation must come before the parameter, bean, or field, it will effect. The position of the **@DefaultValue** annotation relative to the accompanying injection annotation does not matter.

The **@DefaultValue** annotation takes a single parameter. This parameter is the value that will be injected into the field if the proper data cannot be extracted based on the injection annotation. The value can be any **String** value. The value should be compatible with type of the associated field. For example, if the associated field is of type **int**, a default value of **blue** results in an exception.

Dealing with lists and sets

If the type of the annotated parameter, bean or field is List, Set, or SortedSet then the resulting collection will have a single entry mapped from the supplied default value.

Example

[Setting default values](#) shows two examples of using the **@DefaultValue** to specify a default value for a field whose value is injected.

Setting default values

```
import javax.ws.rs.DefaultValue;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("/monster")
public class MonsterService
{
    @Get
    public Monster getMonster(@QueryParam("id") @DefaultValue("42") int id,
        @QueryParam("type") @DefaultValue("bogeyman") String type)
    {
        ...
    }
}
```

```
...  
}
```

The `getMonster()` method in [Setting default values](#) is invoked when a **GET** request is sent to `baseURI/monster`. The method expects two query parameters, `id` and `type`, appended to the URI. So a **GET** request using the URI `baseURI/monster?id=1&type=fomóiri` would return the Fomóiri with the id of one.

Because the `@DefaultValue` annotation is placed on both parameters, the `getMonster()` method can function if the query parameters are omitted. A **GET** request sent to `baseURI/monster` is equivalent to a **GET** request using the URI `baseURI/monster?id=42&type=bogeyman`.

47.2.6. Injecting Parameters into a Java Bean

Overview

When posting HTML forms over REST, a common pattern on the server side is to create a Java bean to encapsulate all of the data received in the form (and possibly data from other parameters and HTML headers, as well). Normally, creating this Java bean would be a two step process: a resource method receives the form values by injection (for example, by adding `@FormParam` annotations to its method parameters), and the resource method then calls the bean's constructor, passing in the form data.

Using the JAX-RS 2.0 `@BeanParam` annotation, it is possible to implement this pattern in a single step. The form data can be injected directly into the fields of the bean class and the bean itself is created automatically by the JAX-RS runtime. This is most easily explained by example.

Injection target

The `@BeanParam` annotation can be attached to resource method parameters, resource fields, or bean properties. A parameter target is the only kind of target that can be used with all resource class lifecycles, however. The other kinds of target are restricted to the per-request lifecycle. This situation is summarized in [Table 47.1, "@BeanParam Injection Targets"](#).

Table 47.1. @BeanParam Injection Targets

Target	Resource Class Lifecycles
PARAMETER	All
FIELD	Per-request (default)
METHOD (bean property)	Per-request (default)

Example without BeanParam annotation

The following example shows how you might go about capturing form data in a Java bean using the conventional approach (without using `@BeanParam`):

```
// Java
import javax.ws.rs.POST;
import javax.ws.rs.FormParam;
```

```

import javax.ws.rs.core.Response;
...
@POST
public Response orderTable(@FormParam("orderId") String orderId,
                           @FormParam("color") String color,
                           @FormParam("quantity") String quantity,
                           @FormParam("price") String price)
{
    ...
    TableOrder bean = new TableOrder(orderId, color, quantity, price);
    ...
    return Response.ok().build();
}

```

In this example, the **orderTable** method processes a form that is used to order a quantity of tables from a furniture Web site. When the order form is posted, the form values are injected into the parameters of the **orderTable** method, and the **orderTable** method explicitly creates an instance of the **TableOrder** class, using the injected form data.

Example with BeanParam annotation

The previous example can be refactored to take advantage of the **@BeanParam** annotation. When using the **@BeanParam** approach, the form parameters can be injected directly into the fields of the bean class, **TableOrder**. In fact, you can use any of the standard JAX-RS parameter annotations in the bean class: including **@PathParam**, **@QueryParam**, **@FormParam**, **@MatrixParam**, **@CookieParam**, and **@HeaderParam**. The code for processing the form can be refactored as follows:

```

// Java
import javax.ws.rs.POST;
import javax.ws.rs.FormParam;
import javax.ws.rs.core.Response;
...
public class TableOrder {
    @FormParam("orderId")
    private String orderId;

    @FormParam("color")
    private String color;

    @FormParam("quantity")
    private String quantity;

    @FormParam("price")
    private String price;

    // Define public getter/setter methods
    // (Not shown)
    ...
}

...
@POST
public Response orderTable(@BeanParam TableOrder orderBean)
{
    ...
    // Do whatever you like with the 'orderBean' bean
}

```

```

...
    return Response.ok().build();
}

```

Now that the form annotations have been added to the bean class, `TableOrder`, you can replace all of the `@FormParam` annotations in the signature of the resource method with just a single `@BeanParam` annotation, as shown. Now, when the form is posted to the `orderTable` resource method, the JAX-RS runtime automatically creates a `TableOrder` instance, `orderBean`, and injects all of the data specified by the parameter annotations on the bean class.

47.3. PARAMETER CONVERTERS

Overview

Using parameter converters, it is possible to inject a parameter (of `String` type) into **any** type of field, bean property, or resource method argument. By implementing and binding a suitable parameter converter, you can extend the JAX-RS runtime so that it is capable of converting the parameter String value to the target type.

Automatic conversions

Parameters are received as instances of `String`, so you can always inject them directly into fields, bean properties, and method parameters of `String` type. In addition, the JAX-RS runtime has the capability to convert parameter strings automatically to the following types:

1. Primitive types.
2. Types that have a constructor that accepts a single `String` argument.
3. Types that have a static method named `valueOf` or `fromString` with a single `String` argument that returns an instance of the type.
4. `List<T>`, `Set<T>`, or `SortedSet<T>`, if `T` is one of the types described in 2 or 3.

Parameter converters

In order to inject a parameter into a type not covered by automatic conversion, you can define a custom *parameter converter* for the type. A parameter converter is a JAX-RS extension that enables you to define conversion from `String` to a custom type, and also in the reverse direction, from the custom type to a `String`.

Factory pattern

The JAX-RS parameter converter mechanism uses a factory pattern. So, instead of registering a parameter converter directly, you must register a parameter converter provider (of type, `javax.ws.rs.ext.ParamConverterProvider`), which creates a parameter converter (of type, `javax.ws.rs.ext.ParamConverter`) on demand.

ParamConverter interface

The `javax.ws.rs.ext.ParamConverter` interface is defined as follows:

```
// Java
```

```

package javax.ws.rs.ext;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.ws.rs.DefaultValue;

public interface ParamConverter<T> {

    @Target({ElementType.TYPE})
    @Retention(RetentionPolicy.RUNTIME)
    @Documented
    public static @interface Lazy {}

    public T fromString(String value);

    public String toString(T value);
}

```

To implement your own **ParamConverter** class, you must implement this interface, overriding the **fromString** method (to convert the parameter string to your target type) and the **toString** method (to convert your target type back to a string).

ParamConverterProvider interface

The **javax.ws.rs.ext.ParamConverterProvider** interface is defined as follows:

```

// Java
package javax.ws.rs.ext;

import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

public interface ParamConverterProvider {
    public <T> ParamConverter<T> getConverter(Class<T> rawType, Type genericType, Annotation annotations[]);
}

```

To implement your own **ParamConverterProvider** class, you must implement this interface, overriding the **getConverter** method, which is a factory method that creates **ParamConverter** instances.

Binding the parameter converter provider

To *bind* the parameter converter provider to the JAX-RS runtime (thus making it available to your application), you must annotate your implementation class with the **@Provider** annotation, as follows:

```

// Java
...
import javax.ws.rs.ext.ParamConverterProvider;
import javax.ws.rs.ext.Provider;

@Provider

```

```
public class TargetTypeProvider implements ParamConverterProvider {
    ...
}
```

This annotation ensures that your parameter converter provider is automatically registered during the scanning phase of deployment.

Example

The following example shows how to implement a **ParamConverterProvider** and a **ParamConverter** which has the capability to convert parameter strings to and from the **TargetType** type:

```
// Java
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

import javax.ws.rs.ext.ParamConverter;
import javax.ws.rs.ext.ParamConverterProvider;
import javax.ws.rs.ext.Provider;

@Provider
public class TargetTypeProvider implements ParamConverterProvider {

    @Override
    public <T> ParamConverter<T> getConverter(
        Class<T> rawType,
        Type genericType,
        Annotation[] annotations
    ) {
        if (rawType.getName().equals(TargetType.class.getName())) {
            return new ParamConverter<T>() {

                @Override
                public T fromString(String value) {
                    // Perform conversion of value
                    // ...
                    TargetType convertedValue = // ... ;
                    return convertedValue;
                }

                @Override
                public String toString(T value) {
                    if (value == null) { return null; }
                    // Assuming that TargetType.toString is defined
                    return value.toString();
                }
            };
        }
        return null;
    }
}
```

Using the parameter converter

Now that you have defined a parameter converter for **TargetType**, it is possible to inject parameters directly into **TargetType** fields and arguments, for example:

```
// Java
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;
...
@POST
public Response updatePost(@FormParam("target") TargetType target)
{
...
}
```

Lazy conversion of default value

If you specify default values for your parameters (using the **@DefaultValue** annotation), you can choose whether the default value is converted to the target type right away (default behaviour), or whether the default value should be converted only when required (lazy conversion). To select lazy conversion, add the **@ParamConverter.Lazy** annotation to the target type. For example:

```
// Java
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;
import javax.ws.rs.DefaultValue;
import javax.ws.rs.ext.ParamConverter.Lazy;
...
@POST
public Response updatePost(
    @FormParam("target")
    @DefaultValue("default val")
    @ParamConverter.Lazy
    TargetType target)
{
...
}
```

47.4. USING APACHE CXF EXTENSIONS

Overview

Apache CXF provides an extension to the standard JAX-WS injection mechanism that allows developers to replace a sequence of injection annotations with a single annotation. The single annotation is placed on a bean containing fields for the data that is extracted using the annotation. For example, if a resource method is expecting a request URI to include three query parameters called **id**, **type**, and **size**, it could use a single **@QueryParam** annotation to inject all of the parameters into a bean with corresponding fields.



NOTE

Consider using the **@BeanParam** annotation instead (available since JAX-RS 2.0). The standardized **@BeanParam** approach is more flexible than the proprietary Apache CXF extension, and is thus the recommended alternative. For details, see [Section 47.2.6, "Injecting Parameters into a Java Bean"](#).

Supported injection annotations

This extension does not support all of the injection parameters. It only supports the following ones:

- **@PathParam**
- **@QueryParam**
- **@MatrixParam**
- **@FormParam**

Syntax

To indicate that an annotation is going to use serial injection into a bean, you need to do two things:

1. Specify the annotation's parameter as an empty string. For example **@PathParam("")** specifies that a sequence of URI template variables are to be serialized into a bean.
2. Ensure that the annotated parameter is a bean with fields that match the values being injected.

Example

[Example 47.7, "Injecting query parameters into a bean"](#) shows an example of injecting a number of Query parameters into a bean. The resource method expect the request URI to include two query parameters: **type** and **id**. Their values are injected into the corresponding fields of the **Monster** bean.

Example 47.7. Injecting query parameters into a bean

```
import javax.ws.rs.QueryParam;
import javax.ws.rsPathParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...
@Path("/monstersforhire/")
public class MonsterService
{
    ...
    @POST
    public void updateMonster(@QueryParam("") Monster bean)
    {
        ...
    }
    ...
}

public class Monster
{
    String type;
    String id;

    ...
}
```

CHAPTER 48. RETURNING INFORMATION TO THE CONSUMER

Abstract

RESTful requests require that at least an HTTP response code be returned to the consumer. In many cases, a request can be satisfied by returning a plain JAXB object or a **GenericEntity** object. When the resource method needs to return additional metadata along with the response entity, JAX-RS resource methods can return a **Response** object containing any needed HTTP headers or other metadata.

48.1. RETURN TYPES

The information returned to the consumer determines the exact type of object a resource method returns. This may seem obvious, but the mapping between Java return objects and what is returned to a RESTful consumer is not one-to-one. At a minimum, RESTful consumers need to be returned a valid HTTP return code in addition to any response entity body. The mapping of the data contained within a Java object to a response entity is effected by the MIME types a consumer is willing to accept.

To address the issues involved in mapping Java object to RESTful response messages, resource methods are allowed to return four types of Java constructs:

- [Section 48.2, “Returning plain Java constructs”](#) return basic information with HTTP return codes determined by the JAX-RS runtime.
- [Section 48.2, “Returning plain Java constructs”](#) return complex information with HTTP return codes determined by the JAX-RS runtime.
- [Section 48.3, “Fine tuning an application’s responses”](#) return complex information with a programmatically determined HTTP return status. The **Response** object also allows HTTP headers to be specified.
- [Section 48.4, “Returning entities with generic type information”](#) return complex information with HTTP return codes determined by the JAX-RS runtime. The **GenericEntity** object provides more information to the runtime components serializing the data.

48.2. RETURNING PLAIN JAVA CONSTRUCTS

Overview

In many cases a resource class can return a standard Java type, a JAXB object, or any object for which the application has an entity provider. In these cases the runtime determines the MIME type information using the Java class of the object being returned. The runtime also determines the appropriate HTTP return code to send to the consumer.

Returnable types

Resource methods can return **void** or any Java type for which an entity writer is provided. By default, the runtime has providers for the following:

- the Java primitives
- the **Number** representations of the Java primitives

- JAXB objects

[the section called “Natively supported types”](#) lists all of the return types supported by default. [the section called “Custom writers”](#) describes how to implement a custom entity writer.

MIME types

The runtime determines the MIME type of the returned entity by first checking the resource method and resource class for a **@Produces** annotation. If it finds one, it uses the MIME type specified in the annotation. If it does not find one specified by the resource implementation, it relies on the entity providers to determine the proper MIME type.

By default the runtime assign MIME types as follows:

- Java primitives and their **Number** representations are assigned a MIME type of **application/octet-stream**.
- JAXB objects are assigned a MIME type of **application/xml**.

Applications can use other mappings by implementing custom entity providers as described in [the section called “Custom writers”](#).

Response codes

When resource methods return plain Java constructs, the runtime automatically sets the response’s status code if the resource method completes without throwing an exception. The status code is set as follows:

- **204**(No Content)–the resource method’s return type is **void**
- **204**(No Content)–the value of the returned entity is **null**
- **200**(OK)–the value of the returned entity is not **null**

If an exception is thrown before the resource method completes the return status code is set as described in [Chapter 50, Handling Exceptions](#).

48.3. FINE TUNING AN APPLICATION’S RESPONSES

48.3.1. Basics of building responses

Overview

RESTful services often need more precise control over the response returned to a consumer than is allowed when a resource method returns a plain Java construct. The JAX-RS **Response** class allows a resource method to have some control over the return status sent to the consumer and to specify HTTP message headers and cookies in the response.

Response objects wrap the object representing the entity that is returned to the consumer. **Response** objects are instantiated using the **ResponseBuilder** class as a factory.

The **ResponseBuilder** class also has many of the methods used to manipulate the response’s metadata. For instance the **ResponseBuilder** class contains the methods for setting HTTP headers and cache control directives.

Relationship between a response and a response builder

The **Response** class has a protected constructor, so they cannot be instantiated directly. They are created using the **ResponseBuilder** class enclosed by the **Response** class. The **ResponseBuilder** class is a holder for all of the information that will be encapsulated in the response created from it. The **ResponseBuilder** class also has all of the methods responsible for setting HTTP header properties on the message.

The **Response** class does provide some methods that ease setting the proper response code and wrapping the entity. There are methods for each of the common response status codes. The methods corresponding to status that include an entity body, or required metadata, include versions that allow for directly setting the information into the associated response builder.

The **ResponseBuilder** class' **build()** method returns a response object containing the information stored in the response builder at the time the method is invoked. After the response object is returned, the response builder is returned to a clean state.

Getting a response builder

There are two ways to get a response builder:

- Using the static methods of the **Response** class as shown in [Getting a response builder using the Response class](#).

Getting a response builder using the Response class

```
import javax.ws.rs.core.Response;
Response r = Response.ok().build();
```

When getting a response builder this way you do not get access to an instance you can manipulate in multiple steps. You must string all of the actions into a single method call.

- Using the Apache CXF specific **ResponseBuilderImpl** class. This class allows you to work directly with a response builder. However, it requires that you manually set all of the response builders information manually.

[Example 48.1, "Getting a response builder using the ResponseBuilderImpl class"](#) shows how [Getting a response builder using the Response class](#) could be rewritten using the **ResponseBuilderImpl** class.

Example 48.1. Getting a response builder using theResponseBuilderImpl class

```
import javax.ws.rs.core.Response;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.status(200);
Response r = builder.build();
```



NOTE

You could also simply assign the **ResponseBuilder** returned from a **Response** class' method to a **ResponseBuilderImpl** object.

More information

For more information about the **Response** class see the [Response class' Javadoc](#).

For more information about the **ResponseBuilder** class see the [ResponseBuilder class' Javadoc](#).

For more information on the Apache CXF **ResponseBuilderImpl** class see the [ResponseBuilderImpl Javadoc](#).

48.3.2. Creating responses for common use cases

Overview

The **Response** class provides shortcut methods for handling the more common responses that a RESTful service will need. These methods handle setting the proper headers using either provided values or default values. They also handle populating the entity body when appropriate.

Creating responses for successful requests

When a request is successfully processed the application needs to send a response to acknowledge that the request has been fulfilled. That response may contain an entity.

The most common response when successfully completing a response is **OK**. An **OK** response typically contains an entity that corresponds to the request. The **Response** class has an overloaded **ok()** method that sets the response status to **200** and adds a supplied entity to the enclosed response builder. There are five versions of the **ok()** method. The most commonly used variant are:

- **Response.ok()**—creates a response with a status of **200** and an empty entity body.
- **Response.ok(java.lang.Object entity)**—creates a response with a status of **200**, stores the supplied object in the responses entity body, and determines the entities media type by introspecting the object.

[Creating a response with an 200 response](#) shows an example of creating a response with an **OK** status.

Creating a response with an 200 response

```
import javax.ws.rs.core.Response;
import demo.jaxrs.server.Customer;
...
Customer customer = new Customer("Jane", 12);
return Response.ok(customer).build();
```

For cases where the requester is not expecting an entity body, it may be more appropriate to send a **204 No Content** status instead of an **200 OK** status. The **Response.noContent()** method will create an appropriate response object.

[Creating a response with a 204 status](#) shows an example of creating a response with an **204** status.

Creating a response with a 204 status

```
import javax.ws.rs.core.Response;
return Response.noContent().build();
```

Creating responses for redirection

The **Response** class provides methods for handling three of the redirection response statuses.

303 See Other

The **303 See Other** status is useful when the requested resource needs to permanently redirect the consumer to a new resource to process the request.

The **Response** classes **seeOther()** method creates a response with a **303** status and places the new resource URI in the message's **Location** field. The **seeOther()** method takes a single parameter that specifies the new URI as a **java.net.URI** object.

304 Not Modified

The **304 Not Modified** status can be used for different things depending on the nature of the request. It can be used to signify that the requested resource has not changed since a previous **GET** request. It can also be used to signify that a request to modify the resource did not result in the resource being changed.

The **Response** classes **notModified()** methods creates a response with a **304** status and sets the modified date property on the HTTP message. There are three versions of the **notModified()** method:

- **notModified**
- **notModifiedjavax.ws.rs.core.Entitytag**
- **notModifiedjava.lang.Stringtag**

307 Temporary Redirect

The **307 Temporary Redirect** status is useful when the requested resource needs to direct the consumer to a new resource, but wants the consumer to continue using this resource to handle future requests.

The **Response** classes **temporaryRedirect()** method creates a response with a **307** status and places the new resource URI in the message's **Location** field. The **temporaryRedirect()** method takes a single parameter that specifies the new URI as a **java.net.URI** object.

[Creating a response with a **304** status](#) shows an example of creating a response with an **304** status.

Creating a response with a **304** status

```
import javax.ws.rs.core.Response;
return Response.notModified().build();
```

Creating responses to signal errors

The **Response** class provides methods to create responses for two basic processing errors:

- **serverError**—creates a response with a status of **500 Internal Server Error**.

- **notAcceptable(java.util.List<javax.ws.rs.core.Variant>variants)**—creates a response with a **406 Not Acceptable** status and an entity body containing a list of acceptable resource types.

[Creating a response with a 500 status](#) shows an example of creating a response with an **500** status.

Creating a response with a 500 status

```
import javax.ws.rs.core.Response;
return Response.serverError().build();
```

48.3.3. Handling more advanced responses

Overview

The **Response** class methods provide short cuts for creating responses for common cases. When you need to address more complicated cases such as specifying cache control directives, adding custom HTTP headers, or sending a status not handled by the **Response** class, you need to use the **ResponseBuilder** classes methods to populate the response before using the **build()** method to generate the response object.

As discussed in [the section called “Getting a response builder”](#), you can use the Apache CXF **ResponseBuilderImpl** class to create a response builder instance that can be manipulated directly.

Adding custom headers

Custom headers are added to a response using the **ResponseBuilder** class' **header()** method. The **header()** method takes two parameters:

- **name**—a string specifying the name of the header
- **value**—a Java object containing the data stored in the header

You can set multiple headers on the message by calling the **header()** method repeatedly.

[Adding a header to a response](#) shows code for adding a header to a response.

Adding a header to a response

```
import javax.ws.rs.core.Response;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.header("username", "joe");
Response r = builder.build();
```

Adding a cookie

Custom headers are added to a response using the **ResponseBuilder** class' **cookie()** method. The **cookie()** method takes one or more cookies. Each cookie is stored in a **javax.ws.rs.core.NewCookie** object. The easiest of the **NewCookie** class' contructors to use takes two parameters:

- **name**—a string specifying the name of the cookie

- **value**—a string specifying the value of the cookie

You can set multiple cookies by calling the **cookie()** method repeatedly.

[Adding a cookie to a response](#) shows code for adding a cookie to a response.

Adding a cookie to a response

```
import javax.ws.rs.core.Response;
import javax.ws.rs.core.NewCookie;

NewCookie cookie = new NewCookie("username", "joe");

Response r = Response.ok().cookie(cookie).build();
```



WARNING

Calling the **cookie()** method with a **null** parameter list erases any cookies already associated with the response.

Setting the response status

When you want to return a status other than one of the statuses supported by the **Response** class' helper methods, you can use the **ResponseBuilder** class' **status()** method to set the response's status code. The **status()** method has two variants. One takes an **int** that specifies the response code. The other takes a **Response.Status** object to specify the response code.

The **Response.Status** class is an enumeration enclosed in the **Response** class. It has entries for most of the defined HTTP response codes.

[Adding a header to a response](#) shows code for setting the response status to **404 Not Found**.

Adding a header to a response

```
import javax.ws.rs.core.Response;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.status(404);
Response r = builder.build();
```

Setting cache control directives

The **ResponseBuilder** class' **cacheControl()** method allows you to set the cache control headers on the response. The **cacheControl()** method takes a **javax.ws.rs.CacheControl** object that specifies the cache control directives for the response.

The **CacheControl** class has methods that correspond to all of the cache control directives supported by the HTTP specification. Where the directive is a simple on or off value the setter method takes a **boolean** value. Where the directive requires a numeric value, such as the **max-age** directive, the setter

takes an **int** value.

[Adding a header to a response](#) shows code for setting the **no-store** cache control directive.

Adding a header to a response

```
import javax.ws.rs.core.Response;
import javax.ws.rs.core.CacheControl;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

CacheControl cache = new CacheControl();
cache.setNoCache(true);

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.cacheControl(cache);
Response r = builder.build();
```

48.4. RETURNING ENTITIES WITH GENERIC TYPE INFORMATION

Overview

There are occasions where the application needs more control over the MIME type of the returned object or the entity provider used to serialize the response. The JAX-RS

javax.ws.rs.core.GenericEntity<T> class provides finer control over the serializing of entities by providing a mechanism for specifying the generic type of the object representing the entity.

Using a **GenericEntity<T>** object

One of the criteria used for selecting the entity provider that serializes a response is the generic type of the object. The generic type of an object represents the Java type of the object. When a common Java type or a JAXB object is returned, the runtime can use Java reflection to determine the generic type. However, when a JAX-RS **Response** object is returned, the runtime cannot determine the generic type of the wrapped entity and the actual Java class of the object is used as the Java type.

To ensure that the entity provider is provided with correct generic type information, the entity can be wrapped in a **GenericEntity<T>** object before being added to the **Response** object being returned.

Resource methods can also directly return a **GenericEntity<T>** object. In practice, this approach is rarely used. The generic type information determined by reflection of an unwrapped entity and the generic type information stored for an entity wrapped in a **GenericEntity<T>** object are typically the same.

Creating a **GenericEntity<T>** object

There are two ways to create a **GenericEntity<T>** object:

1. Create a subclass of the **GenericEntity<T>** class using the entity being wrapped. [Creating a GenericEntity<T> object using a subclass](#) shows how to create a **GenericEntity<T>** object containing an entity of type **List<String>** whose generic type will be available at runtime.

Creating a **GenericEntity<T>** object using a subclass

```
import javax.ws.rs.core.GenericEntity;
List<String> list = new ArrayList<String>();
```

```

...
GenericEntity<List<String>> entity =
    new GenericEntity<List<String>>(list) {};
Response response = Response.ok(entity).build();

```

The subclass used to create a **GenericEntity<T>** object is typically anonymous.

2. Create an instance directly by supplying the generic type information with the entity.

[Example 48.2, “Directly instantiating a GenericEntity<T> object”](#) shows how to create a response containing an entity of type **AtomicInteger**.

Example 48.2. Directly instantiating a GenericEntity<T> object

```

import javax.ws.rs.core.GenericEntity;

AtomicInteger result = new AtomicInteger(12);
GenericEntity<AtomicInteger> entity =
    new GenericEntity<AtomicInteger>(result,
        result.getClass().getGenericSuperclass());
Response response = Response.ok(entity).build();

```

48.5. ASYNCHRONOUS RESPONSE

48.5.1. Asynchronous Processing on the Server

Overview

The purpose of asynchronous processing of invocations on the server side is to enable more efficient use of threads and, ultimately, to avoid the scenario where client connection attempts are refused because all of the server’s request threads are blocked. When an invocation is processed asynchronously, the request thread is freed up almost immediately.



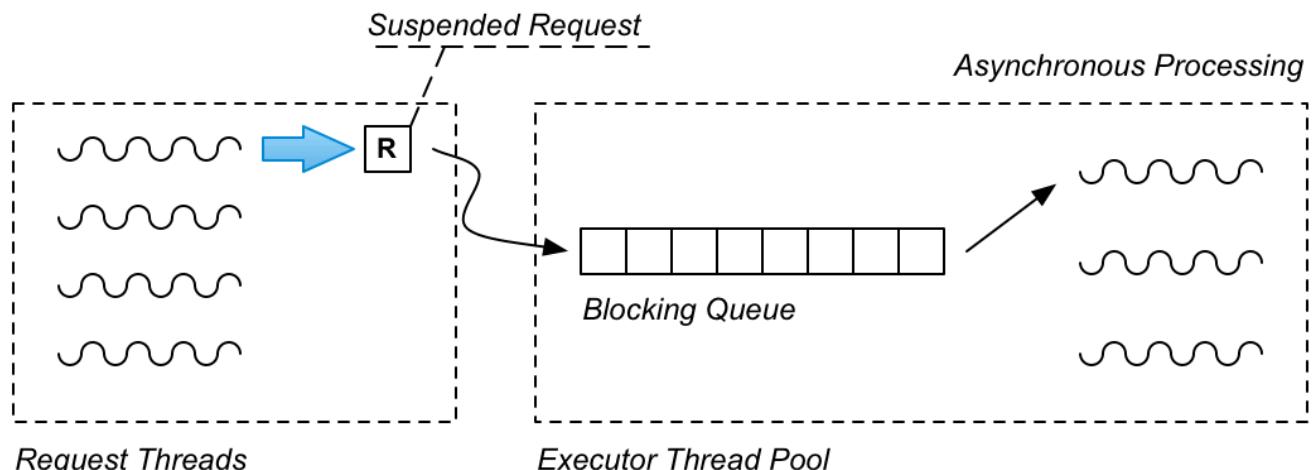
NOTE

Note that even when asynchronous processing is enabled on the server side, a client **will still remain blocked** until it receives a response from the server. If you want to see asynchronous behaviour on the client side, you must implement client-side asynchronous processing. See [Section 49.6, “Asynchronous Processing on the Client”](#).

Basic model for asynchronous processing

[Figure 48.1, “Threading Model for Asynchronous Processing”](#) shows an overview of the basic model for asynchronous processing on the server side.

Figure 48.1. Threading Model for Asynchronous Processing



In outline, a request is processed as follows in the asynchronous model:

1. An asynchronous resource method is invoked within a request thread (and receives a reference to an **AsyncResponse** object, which will be needed later to send back the response).
2. The resource method encapsulates the suspended request in a **Runnable** object, which contains all of the information and processing logic required to process the request.
3. The resource method pushes the **Runnable** object onto the blocking queue of the executor thread pool.
4. The resource method can now return, thus freeing up the request thread.
5. When the **Runnable** object gets to the top of the queue, it is processed by one of the threads in the executor thread pool. The encapsulated **AsyncResponse** object is then used to send the response back to the client.

Thread pool implementation with Java executor

The **java.util.concurrent** API is a powerful API that enables you to create a complete thread pool implementation very easily. In the terminology of the Java concurrency API, a thread pool is called an **executor**. It requires only a single line of code to create a complete working thread pool, including the working threads and the blocking queue that feeds them.

For example, to create a complete working thread pool like the **Executor Thread Pool** shown in Figure 48.1, "Threading Model for Asynchronous Processing", create a **java.util.concurrent.Executor** instance, as follows:

```
Executor executor = new ThreadPoolExecutor(
    5,                                // Core pool size
    5,                                // Maximum pool size
    0,                                // Keep-alive time
    TimeUnit.SECONDS,                  // Time unit
    new ArrayBlockingQueue<Runnable>(10) // Blocking queue
);
```

This constructor creates a new thread pool with five threads, fed by a single blocking queue with which can hold up to 10 **Runnable** objects. To submit a task to the thread pool, call the **executor.execute** method, passing in a reference to a **Runnable** object (which encapsulates the asynchronous task).

Defining an asynchronous resource method

To define a resource method that is asynchronous, inject an argument of type **javax.ws.rs.container.AsyncResponse** using the **@Suspended** annotation and make sure that the method returns **void**. For example:

```
// Java
...
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rsPathParam;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;

@Path("/bookstore")
public class BookContinuationStore {
    ...
    @GET
    @Path("{id}")
    public void handleRequestInPool(@PathParam("id") String id,
                                    @Suspended AsyncResponse response) {
        ...
    }
    ...
}
```

Note that the resource method must return **void**, because the injected **AsyncResponse** object will be used to return the response at a later time.

AsyncResponse class

The **javax.ws.rs.container.AsyncResponse** class provides an abstract handle on an incoming client connection. When an **AsyncResponse** object is injected into a resource method, the underlying TCP client connection is initially in a **suspended** state. At a later time, when you are ready to return the response, you can re-activate the underlying TCP client connection and pass back the response, by calling **resume** on the **AsyncResponse** instance. Alternatively, if you need to abort the invocation, you could call **cancel** on the **AsyncResponse** instance.

Encapsulating a suspended request as a Runnable

In the asynchronous processing scenario shown in [Figure 48.1, “Threading Model for Asynchronous Processing”](#), you push the suspended request onto a queue, from where it can be processed at a later time by a dedicated thread pool. In order for this approach to work, however, you need to have some way of **encapsulating** the suspended request in an object. The suspended request object needs to encapsulate the following things:

- Parameters from the incoming request (if any).
- The **AsyncResponse** object, which provides a handle on the incoming client connection and a way of sending back the response.
- The logic of the invocation.

A convenient way to encapsulate these things is to define a **Runnable** class to represent the suspended request, where the **Runnable.run()** method encapsulates the logic of the invocation. The most elegant way to do this is to implement the **Runnable** as a local class, as shown in the following example.

Example of asynchronous processing

To implement the asynchronous processing scenario, the implementation of the resource method must pass a **Runnable** object (representing the suspended request) to the executor thread pool. In Java 7 and 8, you can exploit some novel syntax to define the **Runnable** class as a local class, as shown in the following example:

```
// Java
package org.apache.cxf.systest.jaxrs;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.Executor;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

import javax.ws.rs.GET;
import javax.ws.rs.NotFoundException;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.CompletionCallback;
import javax.ws.rs.container.ConnectionCallback;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.container.TimeoutHandler;

import org.apache.cxf.phase.PhaseInterceptorChain;

@Path("/bookstore")
public class BookContinuationStore {

    private Map<String, String> books = new HashMap<String, String>();
    private Executor executor = new ThreadPoolExecutor(5, 5, 0, TimeUnit.SECONDS,
        new ArrayBlockingQueue<Runnable>(10));

    public BookContinuationStore() {
        init();
    }
    ...
    @GET
    @Path("{id}")
    public void handleRequestInPool(final @PathParam("id") String id,
        final @Suspended AsyncResponse response) {
        executor.execute(new Runnable() {
            public void run() {
                // Retrieve the book data for 'id'
                // which is presumed to be a very slow, blocking operation
                // ...
                bookdata = ...
                // Re-activate the client connection with 'resume'
                // and send the 'bookdata' object as the response
                response.resume(bookdata);
            }
        });
    }
}
```

```

    });
}
...
}

```

Note how the resource method arguments, **id** and **response**, are passed straight into the definition of the **Runnable** local class. This special syntax enables you to use the resource method arguments directly in the **Runnable.run()** method, without having to define corresponding fields in the local class.



IMPORTANT

In order for this special syntax to work, the resource method parameters **must** be declared as **final** (which implies that they must not be changed in the method implementation).

48.5.2. Timeouts and Timeout Handlers

Overview

The asynchronous processing model also provides support for imposing timeouts on REST invocations. By default, a timeout results in a HTTP error response being sent back to the client. But you also have the option of registering a timeout handler callback, which enables you to customize the response to a timeout event.

Example of setting a timeout without a handler

To define a simple invocation timeout, without specifying a timeout handler, call the **setTimeout** method on the **AsyncResponse** object, as shown in the following example:

```

// Java
// Java
...
import java.util.concurrent.TimeUnit;
...
import javax.ws.rs.GET;
import javax.ws.rs.NotFoundException;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.container.TimeoutHandler;

@Path("/bookstore")
public class BookContinuationStore {
    ...
    @GET
    @Path("/books/defaulttimeout")
    public void getBookDescriptionWithTimeout(@Suspended AsyncResponse async) {
        async.setTimeout(2000, TimeUnit.MILLISECONDS);
        // Optionally, send request to executor queue for processing
        // ...
    }
}

```

```

    }
    ...
}
```

Note that you can specify the timeout value using any time unit from the **java.util.concurrent.TimeUnit** class. The preceding example does not show the code for sending the request to the executor thread pool. If you just wanted to test the timeout behaviour, you could include just the call to **async.setTimeout** in the resource method body, and the timeout would be triggered on every invocation.

The **AsyncResponse.NO_TIMEOUT** value represents an infinite timeout.

Default timeout behaviour

By default, if the invocation timeout is triggered, the JAX-RS runtime raises a **ServiceUnavailableException** exception and sends back a HTTP error response with the status **503**.

TimeoutHandler interface

If you want to customize the timeout behaviour, you must define a timeout handler, by implementing the **TimeoutHandler** interface:

```

// Java
package javax.ws.rs.container;

public interface TimeoutHandler {
    public void handleTimeout(AsyncResponse asyncResponse);
}
```

When you override the **handleTimeout** method in your implementation class, you can choose between the following approaches to dealing with the timeout:

- Cancel the response, by calling the **asyncResponse.cancel** method.
- Send a response, by calling the **asyncResponse.resume** method with the response value.
- Extend the waiting period, by calling the **asyncResponse.setTimeout** method. (For example, to wait for a further 10 seconds, you could call **asyncResponse.setTimeout(10, TimeUnit.SECONDS)**).

Example of setting a timeout with a handler

To define an invocation timeout with a timeout handler, call both the **setTimeout** method and the **setTimeoutHandler** method on the **AsyncResponse** object, as shown in the following example:

```

// Java
...
import javax.ws.rs.GET;
import javax.ws.rs.NotFoundException;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.container.TimeoutHandler;
```

```

@Path("/bookstore")
public class BookContinuationStore {

    ...
    @GET
    @Path("/books/cancel")
    public void getBookDescriptionWithCancel(@PathParam("id") String id,
                                             @Suspended AsyncResponse async) {
        async.setTimeout(2000, TimeUnit.MILLISECONDS);
        async.setTimeoutHandler(new CancelTimeoutHandlerImpl());
        // Optionally, send request to executor queue for processing
        // ...
    }
    ...
}

```

Where this example registers an instance of the **CancelTimeoutHandlerImpl** timeout handler to handle the invocation timeout.

Using a timeout handler to cancel the response

The **CancelTimeoutHandlerImpl** timeout handler is defined as follows:

```

// Java
...
import javax.ws.rs.container.AsyncResponse;
...
import javax.ws.rs.container.TimeoutHandler;

@Path("/bookstore")
public class BookContinuationStore {

    ...
    private class CancelTimeoutHandlerImpl implements TimeoutHandler {

        @Override
        public void handleTimeout(AsyncResponse asyncResponse) {
            asyncResponse.cancel();
        }
    }
    ...
}

```

The effect of calling **cancel** on the **AsyncResponse** object is to send a HTTP 503 (**Service unavailable**) error response to the client. You can optionally specify an argument to the **cancel** method (either an **int** or a **java.util.Date** value), which would be used to set a **Retry-After:** HTTP header in the response message. Clients often ignore the **Retry-After:** header, however.

Dealing with a cancelled response in the Runnable instance

If you have encapsulated a suspended request as a **Runnable** instance, which is queued for processing in an executor thread pool, you might find that the **AsyncResponse** has been cancelled by the time the thread pool gets around to processing the request. For this reason, you ought to add some code to your **Runnable** instance, which enables it to cope with a cancelled **AsyncResponse** object. For example:

```
// Java
...
@Path("/bookstore")
public class BookContinuationStore {
    ...
    private void sendRequestToThreadPool(final String id, final AsyncResponse response) {
        executor.execute(new Runnable() {
            public void run() {
                if ( !response.isCancelled() ) {
                    // Process the suspended request ...
                    // ...
                }
            }
        });
    }
    ...
}
```

48.5.3. Handling Dropped Connections

Overview

It is possible to add a callback to deal with the case where the client connection is lost.

ConnectionCallback interface

To add a callback for dropped connections, you must implement the [javax.ws.rs.container.ConnectionCallback](#) interface, which is defined as follows:

```
// Java
package javax.ws.rs.container;

public interface ConnectionCallback {
    public void onDisconnect(AsyncResponse disconnected);
}
```

Registering a connection callback

After implementing a connection callback, you must register it with the current **AsyncResponse** object, by calling one of the **register** methods. For example, to register a connection callback of type, **MyConnectionCallback**:

```
asyncResponse.register(new MyConnectionCallback());
```

Typical scenario for connection callback

Typically, the main reason for implementing a connection callback would be to free up resources associated with the dropped client connection (where you could use the **AsyncResponse** instance as the key to identify the resources that need to be freed).

48.5.4. Registering Callbacks

Overview

You can optionally add a callback to an **AsyncResponse** instance, in order to be notified when the invocation has completed. There are two alternative points in the processing when this callback can be invoked, either:

- After the request processing is finished and the response has already been sent back to the client, or
- After the request processing is finished and an unmapped **Throwable** has been propagated to the hosting I/O container.

CompletionCallback interface

To add a completion callback, you must implement the [javax.ws.rs.container.CompletionCallback](#) interface, which is defined as follows:

```
// Java
package javax.ws.rs.container;

public interface CompletionCallback {
    public void onComplete(Throwable throwable);
}
```

Usually, the **throwable** argument is **null**. However, if the request processing resulted in an unmapped exception, **throwable** contains the unmapped exception instance.

Registering a completion callback

After implementing a completion callback, you must register it with the current **AsyncResponse** object, by calling one of the **register** methods. For example, to register a completion callback of type, **MyCompletionCallback**:

```
asyncResponse.register(new MyCompletionCallback());
```

CHAPTER 49. JAX-RS 2.0 CLIENT API

Abstract

JAX-RS 2.0 defines a full-featured client API which can be used for making REST invocations or any HTTP client invocations. This includes a fluent API (to simplify building up requests), a framework for parsing messages (based on a type of plug-in known as an *entity provider*), and support for asynchronous invocations on the client side.

49.1. INTRODUCTION TO THE JAX-RS 2.0 CLIENT API

Overview

JAX-RS 2.0 defines a fluent API for JAX-RS clients, which enables you to build up a HTTP request step-by-step and then invoke the request using the appropriate HTTP verb (GET, POST, PUT, or DELETE).



NOTE

It is also possible to define a JAX-RS client in Blueprint XML or Spring XML (using the **jaxrs:client** element). For details of this approach, see [Section 18.2, "Configuring JAX-RS Client Endpoints"](#).

Dependencies

To use the JAX-RS 2.0 client API in your application, you must add the following Maven dependency to your project's **pom.xml** file:

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxfrt-rs-client</artifactId>
  <version>3.2.7.fuse-750027-redhat-00001</version>
</dependency>
```

If you plan to use the asynchronous invocation feature (see [Section 49.6, "Asynchronous Processing on the Client"](#)), you also need the following Maven dependency:

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxfrt-transports-http-hc</artifactId>
  <version>3.2.7.fuse-750027-redhat-00001</version>
</dependency>
```

Client API package

The JAX-RS 2.0 client interfaces and classes are located in the following Java package:

```
javax.ws.rs.client
```

When developing JAX-RS 2.0 Java clients, you also typically need to access classes from the core package:

```
javax.ws.rs.core
```

Example of a simple client request

The following code fragment shows a simple example, where the JAX-RS 2.0 client API is used to make an invocation on the <http://example.org/bookstore> JAX-RS service, invoking with the GET HTTP method:

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
Response res = client.target("http://example.org/bookstore/books/123")
    .request("application/xml").get();
```

Fluent API

The JAX-RS 2.0 client API is designed as a *fluent API* (sometimes called a Domain Specific Language). In the fluent API, a chain of Java methods is invoked in a single statement, in such a way that the Java methods look like the commands from a simple language. In JAX-RS 2.0, the fluent API is used to build and invoke a REST request.

Steps to make a REST invocation

Using the JAX-RS 2.0 client API, a client invocation is built and invoked in a series of steps, as follows:

1. Bootstrap the client.
2. Configure the target.
3. Build and make the invocation.
4. Parse the response.

Bootstrap the client

The first step is to bootstrap the client, by creating a `javax.ws.rs.client.Client` object. This **Client** instance is a relatively heavyweight object, which represents the stack of technologies required to support a JAX-RS client (possibly including, interceptors and additional CXF features). Ideally, you should re-use client objects when you can, instead of creating new ones.

To create a new **Client** object, invoke a static method on the **ClientBuilder** class, as follows:

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
...
Client client = ClientBuilder.newClient();
...
```

Configure the target

By configuring the target, you effectively define the URI that will be used for the REST invocation. The following example shows how you can define a base URI, **base**, and then add additional path segments to the base URI, using the **path(String)** method:

```
// Java
import javax.ws.rs.client.WebTarget;
...
WebTarget base = client.target("http://example.org/bookstore/");
WebTarget books = base.path("books").path("{id}");
...
```

Build and make the invocation

This is really two steps rolled up into one: firstly, you build up the HTTP request (including headers, accepted media types, and so on); and secondly, you invoke the relevant HTTP method (optionally providing a request message body, if one is required).

For example, to create and invoke a request that accepts the **application/xml** media type:

```
// Java
import javax.ws.rs.core.Response;
...
Response resp = books.resolveTemplate("id", "123").request("application/xml").get();
```

Parse the response

Finally, you need to parse the response, **resp**, obtained in the previous step. Usually, the response is returned in the form of a **javax.ws.rs.core.Response** object, which encapsulates HTTP headers, along with other HTTP metadata, and the HTTP message body (if any).

If you want to access the returned HTTP message in **String** format, you can easily do so by invoking the **readEntity** method with a **String.class** argument, as follows:

```
// Java
...
String msg = resp.readEntity(String.class);
```

You can always access the message body of a response as a **String**, by specifying **String.class** as the argument to **readEntity**. For more general transformations or conversions of the message body, you can provide an *entity provider* to perform the conversion. For more details, see [Section 49.4, “Parsing Requests and Responses”](#).

49.2. BUILDING THE CLIENT TARGET

Overview

After creating the initial **Client** instance, the next step is to build up the request URI. The **WebTarget** builder class enables you to configure all aspects of the URI, including the URI path and query parameters.

WebTarget builder class

The **javax.ws.rs.client.WebTarget** builder class provides the part of the fluent API that enables you to build up the REST URI for the request.

Create the client target

To create a **WebTarget** instance, invoke one of the **target** methods on a **javax.ws.rs.client.Client** instance. For example:

```
// Java
import javax.ws.rs.client.WebTarget;
...
WebTarget base = client.target("http://example.org/bookstore/");
```

Base path and path segments

You can specify the complete path all in one go, using the **target** method; or you can specify a base path, and then add path segments piece by piece, using a combination of the **target** method and the **path** methods. The advantage of combining a base path with path segments is that you can easily reuse the base path **WebTarget** object for multiple invocations on slightly different targets. For example:

```
// Java
import javax.ws.rs.client.WebTarget;
...
WebTarget base = client.target("http://example.org/bookstore/");
WebTarget headers = base.path("bookheaders");
// Now make some invocations on the 'headers' target...
...
WebTarget collections = base.path("collections");
// Now make some invocations on the 'collections' target...
...
```

URI template parameters

The syntax of the target path also supports URI template parameters. That is, a path segment can be initialized with a template parameter, **{param}**, which subsequently gets resolved to a specify value. For example:

```
// Java
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
...
WebTarget base = client.target("http://example.org/bookstore/");
WebTarget books = base.path("books").path("{id}");
...
Response resp = books.resolveTemplate("id", "123").request("application/xml").get();
```

Where the **resolveTemplate** method replaces the path segment, **{id}**, with the value **123**.

Define query parameters

Query parameters can be appended to the URI path, where the beginning of the query parameters is marked by a single **?** character. This mechanism enables you to set a series of name/value pairs, using the syntax: **?name1=value1&name2=value2&...**

A **WebTarget** instance enables you to define query parameters using the **QueryParam** method, as follows:

```
// Java
WebTarget target = client.target("http://example.org/bookstore/")
    .queryParam("userId","Agamemnon")
    .queryParam("lang","gr");
```

Define matrix parameters

Matrix parameters are somewhat similar to query parameters, but are not as widely supported and use a different syntax. To define a matrix parameter on a **WebTarget** instance, invoke the **matrixParam(String, Object)** method.

49.3. BUILDING THE CLIENT INVOCATION

Overview

After building the target URI, using the **WebTarget** builder class, the next step is to configure the other aspects of the request—such as HTTP headers, cookies, and so on—using the **Invocation.Builder** class. The final step in building the invocation is to invoke the appropriate HTTP verb (GET, POST, PUT, or DELETE) and provide a message body, if required.

Invocation.Builder class

The **javax.ws.rs.client.Invocation.Builder** builder class provides the part of the fluent API that enables you to build up the contents of the HTTP message and to invoke a HTTP method.

Create the invocation builder

To create an **Invocation.Builder** instance, invoke one of the **request** methods on a **javax.ws.rs.client.WebTarget** instance. For example:

```
// Java
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.client.Invocation.Builder;
...
WebTarget books = client.target("http://example.org/bookstore/books/123");
Invocation.Builder invbuilder = books.request();
```

Define HTTP headers

You can add a HTTP header to the request message using the **header** method, as follows:

```
Invocation.Builder invheader = invbuilder.header("From", "fionn@example.org");
```

Define cookies

You can add a cookie to the request message using the **cookie** method, as follows:

```
Invocation.Builder invcookie = invbuilder.cookie("myrestclient", "123xyz");
```

Define properties

You can set a property in the context of this request using the `property` method, as follows:

```
Invocation.Builder invproperty = invbuilder.property("Name", "Value");
```

Define accepted media types, languages, or encodings

You can define accepted media types, languages, or encodings, as follows:

```
Invocation.Builder invmedia = invbuilder.accept("application/xml")
    .acceptLanguage("en-US")
    .acceptEncoding("gzip");
```

Invoke HTTP method

The process of building a REST invocation is terminated by invoking a HTTP method, which performs the HTTP invocation. The following methods (inherited from the [javax.ws.rs.client.SyncInvoker](#) base class) can be invoked:

```
get
post
delete
put
head
trace
options
```

If the specific HTTP verb you want to invoke is not on this list, you can use the generic `method` method to invoke any HTTP method.

Typed responses

All of the HTTP invocation methods are provided with an untyped variant and a typed variant (which takes an extra argument). If you invoke a request using the default `get()` method (taking no arguments), a [javax.ws.rs.core.Response](#) object is returned from the invocation. For example:

```
Response res = client.target("http://example.org/bookstore/books/123")
    .request("application/xml").get();
```

It is also possible, however, to ask for the response to be returned as a specific type, using the `get(Class<T>)` method. For example, to invoke a request and ask for the response to be returned as a [BookInfo](#) object:

```
BookInfo res = client.target("http://example.org/bookstore/books/123")
    .request("application/xml").get(BookInfo.class);
```

In order for this to work, however, you must register a suitable *entity provider* with the [Client](#) instance, which is capable of mapping the response format, [application/xml](#), to the requested type. For more details about entity providers, see [Section 49.4, “Parsing Requests and Responses”](#).

Specifying the outgoing message in post or put

For HTTP methods that include a message body in the request (such as POST or PUT), you must specify the message body as the first argument of the method. The message body must be specified as a **javax.ws.rs.client.Entity** object, where the **Entity** encapsulates the message contents and its associated media type. For example, to invoke a POST method, where the message contents are provided as a **String** type:

```
import javax.ws.rs.client.Entity;
...
Response res = client.target("http://example.org/bookstore/registerbook")
    .request("application/xml")
    .put(Entity.entity("Red Hat Install Guide", "text/plain"));
```

If necessary, the **Entity.entity()** constructor method will automatically map the supplied message instance to the specified media type, using the registered entity providers. It is always possible to specify the message body as a simple **String** type.

Delayed invocation

Instead of invoking the HTTP request right away (for example, by invoking the **get()** method), you have the option of creating an **javax.ws.rs.client.Invocation** object, which can be invoked at a later time. The **Invocation** object encapsulates all of the details of the pending invocation, including the HTTP method.

The following methods can be used to build an **Invocation** object:

```
buildGet
buildPost
buildDelete
buildPut
build
```

For example, to create a GET **Invocation** object and invoke it at a later time, you can use code like the following:

```
import javax.ws.rs.client.Invocation;
import javax.ws.rs.core.Response;
...
Invocation getBookInfo = client.target("http://example.org/bookstore/books/123")
    .request("application/xml").buildGet();
...
// Later on, in some other part of the application:
Response = getBookInfo.invoke();
```

Asynchronous invocation

The JAX-RS 2.0 client API supports asynchronous invocations on the client side. To make an asynchronous invocation, simply invoke the **async()** method in the chain of methods following **request()**. For example:

```
Future<Response> res = client.target("http://example.org/bookstore/books/123")
    .request("application/xml")
    .async()
    .get();
```

When you make an asynchronous invocation, the returned value is a **java.util.concurrent.Future** object. For more details about asynchronous invocations, see [Section 49.6, “Asynchronous Processing on the Client”](#).

49.4. PARSING REQUESTS AND RESPONSES

Overview

An essential aspect of making HTTP invocations is that the client must be able to parse the outgoing request messages and the incoming responses. In JAX-RS 2.0, the key concept is the **Entity** class, which represents a raw message tagged with a media type. In order to parse the raw message, you can register multiple *entity providers*, which have the capability to convert media types to and from particular Java types.

In other words, in the context of JAX-RS 2.0, an **Entity** is the representation of a raw message and an entity provider is the plug-in that provides the capability to parse the raw message (based on the media type).

Entities

An **Entity** is a message body augmented by metadata (media type, language, and encoding). An **Entity** instance holds the message in a raw format and is associated with a specific media type. To convert the contents of an **Entity** object to a Java object you require an *entity provider*, which is capable of mapping the given media type to the required Java type.

Variants

A **javax.ws.rs.core.Variant** object encapsulates the metadata associated with an **Entity**, as follows:

- Media type,
- Language,
- Encoding.

Effectively, you can think of an **Entity** as consisting of the HTTP message contents, augmented by **Variant** metadata.

Entity providers

An entity provider is a class that provides the capability of mapping between a media type and a Java type. Effectively, you can think of an entity provider as a class that provides the ability to parse messages of a particular media type (or possibly of multiple media types). There are two different varieties of entity provider:

MessageBodyReader

Provides the capability of mapping from media type(s) to a Java type.

MessageBodyWriter

Provides the capability of mapping from a Java type to a media type.

Standard entity providers

Entity providers for the following Java and media type combinations are provided as standard:

byte[]

All media types (`/*`).

java.lang.String

All media types (`/*`).

java.io.InputStream

All media types (`/*`).

java.io.Reader

All media types (`/*`).

java.io.File

All media types (`/*`).

javax.activation.DataSource

All media types (`/*`).

javax.xml.transform.Source

XML types (`text/xml`, `application/xml`, and media types of the form `application/*+xml`).

javax.xml.bind.JAXBElement and application-supplied JAXB classes

XML types (`text/xml`, `application/xml`, and media types of the form `application/*+xml`).

MultivaluedMap<String, String>

Form content (`application/x-www-form-urlencoded`).

StreamingOutput

All media types (`/*`), **MessageBodyWriter** only.

java.lang.Boolean, java.lang.Character, java.lang.Number

Only for **text/plain**. Corresponding primitive types supported through boxing/unboxing conversion.

Response object

The default return type is the **javax.ws.rs.core.Response** type, which represents an untyped response. The **Response** object provides access to the complete HTTP response, including the message body, HTTP status, HTTP headers, media type, and so on.

Accessing the response status

You can access the response status, either through the **getStatus** method (which returns the HTTP status code):

```
int status = resp.getStatus();
```

Or though the **getStatusInfo** method, which also provides a description string:

```
String statusReason = resp.getStatusInfo().getReasonPhrase();
```

Accessing the returned headers

You can access the HTTP headers using any of the following methods:

```
MultivaluedMap<String, Object>
getHeaders()
```

```
MultivaluedMap<String,String>
getStringHeaders()

String
getHeaderString(String name)
```

For example, if you know that the **Response** has a **Date** header, you could access it as follows:

```
String dateAsString = resp.getHeaderString("Date");
```

Accessing the returned cookies

You can access any new cookies set on the **Response** using the **getCookies** method, as follows:

```
import javax.ws.rs.core.NewCookie;
...
java.util.Map<String,NewCookie> cookieMap = resp.get Cookies();
java.util.Collection<NewCookie> cookieCollection = cookieMap.values();
```

Accessing the returned message content

You can access the returned message content by invoking one of the **readEntity** methods on the **Response** object. The **readEntity** method automatically invokes the available entity providers to convert the message to the requested type (specified as the first argument of **readEntity**). For example, to access the message content as a **String** type:

```
String messageBody = resp.readEntity(String.class);
```

Collection return value

If you need to access the returned message as a Java generic type—for example, as a **List** or **Collection** type—you can specify the request message type using the **javax.ws.rs.core.GenericType<T>** construction. For example:

```
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import javax.ws.rs.core.GenericType;
import java.util.List;
...
GenericType<List<String>> stringListType = new GenericType<List<String>>() {};

Client client = ClientBuilder.newClient();
List<String> bookNames = client.target("http://example.org/bookstore/booknames")
    .request("text/plain")
    .get(stringListType);
```

49.5. CONFIGURING THE CLIENT ENDPOINT

Overview

It is possible to augment the functionality of the base **javax.ws.rs.client.Client** object by registering and configuring features and providers.

Example

The following example shows a client configured to have a logging feature, a custom entity provider, and to set the **prettyLogging** property to **true**:

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import org.apache.cxf.feature.LoggingFeature;
...
Client client = ClientBuilder.newClient();
client.register(LoggingFeature.class)
    .register(MyCustomEntityProvider.class)
    .property("LoggingFeature.prettyLogging","true");
```

Configurable API for registering objects

The **Client** class supports the **Configurable** API for registering objects, which provides several variants of the **register** method. In most cases, you would register either a class or an object instance, as shown in the following examples:

```
client.register(LoggingFeature.class)
client.register(new LoggingFeature())
```

For more details about the **register** variants, see the reference documentation for [Configurable](#).

What can you configure on the client?

You can configure the following aspects of a client endpoint:

- Features
- Providers
- Properties
- Filters
- Interceptors

Features

A [javax.ws.rs.core.Feature](#) is effectively a plug-in that adds an extra feature or functionality to a JAX-RS client. Often, a feature installs one or more interceptors in order to provide the required functionality.

Providers

A provider is a particular kind of client plug-in that provides a mapping capability. The JAX-RS 2.0 specification defines the following kinds of provider:

Entity providers

An *entity provider* provides the capability of mapping between a specific media type a Java type. For more details, see [Section 49.4, "Parsing Requests and Responses"](#).

Exception mapping providers

An *exception mapping provider* maps a checked runtime exception to an instance of **Response**.

Context providers

A context provider is used on the server side, to supply context to resource classes and other service providers.

Filters

A JAX-RS 2.0 filter is a plug-in that gives you access to the URI, headers, and miscellaneous context data at various points (extension points) of the message processing pipeline. For details, see [Chapter 61, JAX-RS 2.0 Filters and Interceptors](#).

Interceptors

A JAX-RS 2.0 interceptor is a plug-in that gives you access to the message body of a request or response as it is being read or written. For details, see [Chapter 61, JAX-RS 2.0 Filters and Interceptors](#).

Properties

By setting one or more properties on the client, you can customize the configuration of a registered feature or a registered provider.

Other configurable types

It is possible, not only to configure a **javax.ws.rs.client.Client** (and **javax.ws.rs.client.ClientBuilder**) object, but also a **WebTarget** object. When you change the configuration of a **WebTarget** object, the underlying client configuration is deep copied to give the new **WebTarget** configuration. Hence, it is possible to change the configuration of the **WebTarget** object without changing the configuration of the original **Client** object.

49.6. ASYNCHRONOUS PROCESSING ON THE CLIENT

Overview

JAX-RS 2.0 supports asynchronous processing of invocations on the client side. Two different styles of asynchronous processing are supported: either using a **java.util.concurrent.Future<V>** return value; or by registering an invocation callback.

Asynchronous invocation with Future<V> return value

Using the **Future<V>** approach to asynchronous processing, you can invoke a client request asynchronously, as follows:

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import java.util.concurrent.Future;
import javax.ws.rs.core.Response;
```

```

...
Client client = ClientBuilder.newClient();
Future<Response> futureResp = client.target("http://example.org/bookstore/books/123")
    .request("application/xml")
    .async()
    .get();
...
// At a later time, check (and wait) for the response:
Response resp = futureResp.get();

```

You can use a similar approach for typed responses. For example, to get a response of type, **BookInfo**:

```

Client client = ClientBuilder.newClient();
Future<BookInfo> futureResp = client.target("http://example.org/bookstore/books/123")
    .request("application/xml")
    .async()
    .get(BookInfo.class);
...
// At a later time, check (and wait) for the response:
BookInfo resp = futureResp.get();

```

Asynchronous invocation with invocation callback

Instead of accessing the return value using a Future<V> object, you can define an invocation callback (using [javax.ws.rs.client.InvocationCallback<RESPONSE>](#)), as follows:

```

// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import java.util.concurrent.Future;
import javax.ws.rs.core.Response;
import javax.ws.rs.client.InvocationCallback;
...
Client client = ClientBuilder.newClient();
Future<Response> futureResp = client.target("http://example.org/bookstore/books/123")
    .request("application/xml")
    .async()
    .get(
        new InvocationCallback<Response>() {
            @Override
            public void completed(final Response resp) {
                // Do something when invocation is complete
                ...
            }
            @Override
            public void failed(final Throwable throwable) {
                throwable.printStackTrace();
            }
        });
...

```

You can use a similar approach for typed responses:

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import java.util.concurrent.Future;
import javax.ws.rs.core.Response;
import javax.ws.rs.client.InvocationCallback;

...
Client client = ClientBuilder.newClient();
Future<BookInfo> futureResp = client.target("http://example.org/bookstore/books/123")
    .request("application/xml")
    .async()
    .get
new InvocationCallback<BookInfo>() {
    @Override
    public void completed(final BookInfo resp) {
        // Do something when invocation is complete
        ...
    }
    @Override
    public void failed(final Throwable throwable) {
        throwable.printStackTrace();
    }
});
...
...
```

CHAPTER 50. HANDLING EXCEPTIONS

Abstract

When possible, exceptions caught by a resource method should cause a useful error to be returned to the requesting consumer. JAX-RS resource methods can throw a `WebApplicationException` exception. You can also provide `ExceptionMapper<E>` implementations to map exceptions to appropriate responses.

50.1. OVERVIEW OF JAX-RS EXCEPTION CLASSES

Overview

In JAX-RS 1.x, the only available exception class is `WebApplicationException`. Since JAX-WS 2.0, however, a number of additional JAX-RS exception classes have been defined.

JAX-RS runtime level exceptions

The following exceptions are meant to be thrown by the JAX-RS runtime only (that is, you must **not** throw these exceptions from your application level code):

ProcessingException

(JAX-RS 2.0 only) The `javax.ws.rs.ProcessingException` can be thrown during request processing or during response processing in the JAX-RS runtime. For example, this error could be thrown due to errors in the filter chain or interceptor chain processing.

ResponseProcessingException

(JAX-RS 2.0 only) The `javax.ws.rs.client.ResponseProcessingException` is a subclass of `ProcessingException`, which can be thrown when errors occur in the JAX-RS runtime on the `client` side.

JAX-RS application level exceptions

The following exceptions are intended to be thrown (and caught) in your application level code:

WebApplicationException

The `javax.ws.rs.WebApplicationException` is a generic application level JAX-RS exception, which can be thrown in application code on the server side. This exception type can encapsulate a HTTP status code, an error message, and (optionally) a response message. For details, see [Section 50.2, "Using WebApplicationException exceptions to report"](#).

ClientErrorException

(JAX-RS 2.0 only) The `javax.ws.rs.ClientErrorException` exception class inherits from `WebApplicationException` and is used to encapsulate HTTP `4xx` status codes.

ServerErrorException

(JAX-RS 2.0 only) The `javax.ws.rs.ServerErrorException` exception class inherits from `WebApplicationException` and is used to encapsulate HTTP `5xx` status codes.

RedirectionException

(JAX-RS 2.0 only) The `javax.ws.rs.RedirectionException` exception class inherits from `WebApplicationException` and is used to encapsulate HTTP `3xx` status codes.

50.2. USING WEBAPPLICATIONEXCEPTION EXCEPTIONS TO REPORT

errors
indexterm:[WebApplicationException]

Overview

The JAX-RS API introduced the WebApplicationException runtime exception to provide an easy way for resource methods to create exceptions that are appropriate for RESTful clients to consume. WebApplicationException exceptions can include a **Response** object that defines the entity body to return to the originator of the request. It also provides a mechanism for specifying the HTTP status code to be returned to the client if no entity body is provided.

Creating a simple exception

The easiest means of creating a WebApplicationException exception is to use either the no argument constructor or the constructor that wraps the original exception in a WebApplicationException exception. Both constructors create a WebApplicationException with an empty response.

When an exception created by either of these constructors is thrown, the runtime returns a response with an empty entity body and a status code of **500 Server Error**.

Setting the status code returned to the client

When you want to return an error code other than **500**, you can use one of the four WebApplicationException constructors that allow you to specify the status. Two of these constructors, shown in [Example 50.1, “Creating a WebApplicationException with a status code”](#), take the return status as an integer.

Example 50.1. Creating a WebApplicationException with a status code

```
WebApplicationException int status WebApplicationException java.lang.Throwable cause int status
```

The other two, shown in [Example 50.2, “Creating a WebApplicationException with a status code”](#) take the response status as an instance of **Response.Status**.

Example 50.2. Creating a WebApplicationException with a status code

```
WebApplicationException javax.ws.rs.core.Response.Status status WebApplicationException java.lang.Throwable cause javax.ws.rs.core.Response.Status status
```

When an exception created by either of these constructors is thrown, the runtime returns a response with an empty entity body and the specified status code.

Providing an entity body

If you want a message to be sent along with the exception, you can use one of the WebApplicationException constructors that takes a **Response** object. The runtime uses the **Response** object to create the response sent to the client. The entity stored in the response is mapped to the

entity body of the message and the status field of the response is mapped to the HTTP status of the message.

[Example 50.3, "Sending a message with an exception"](#) shows code for returning a text message to a client containing the reason for the exception and sets the HTTP message status to **409 Conflict**.

Example 50.3. Sending a message with an exception

```
import javax.ws.rs.core.Response;
import javax.ws.rs.WebApplicationException;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

...
ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.status(Response.Status.CONFLICT);
builder.entity("The requested resource is conflicted.");
Response response = builder.build();
throw WebApplicationException(response);
```

Extending the generic exception

It is possible to extend the `WebApplicationException` exception. This would allow you to create custom exceptions and eliminate some boiler plate code.

[Example 50.4, "Extending WebApplicationException"](#) shows a new exception that creates a similar response to the code in [Example 50.3, "Sending a message with an exception"](#).

Example 50.4. Extending WebApplicationException

```
public class ConflictedException extends WebApplicationException
{
    public ConflictedException(String message)
    {
        ResponseBuilderImpl builder = new ResponseBuilderImpl();
        builder.status(Response.Status.CONFLICT);
        builder.entity(message);
        super(builder.build());
    }
}

...
throw ConflictedException("The requested resource is conflicted.");
```

50.3. JAX-RS 2.0 EXCEPTION TYPES

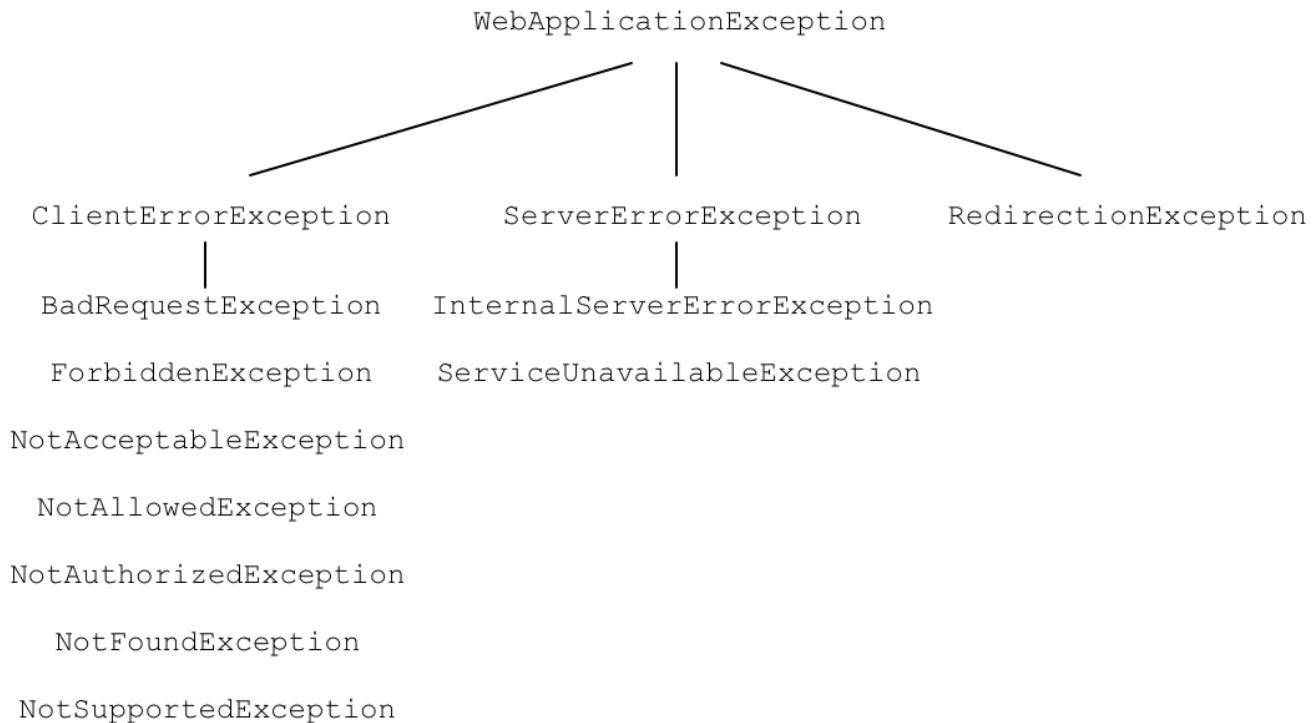
Overview

JAX-RS 2.0 introduces a number of specific HTTP exception types that you can throw (and catch) in your application code (in addition to the existing **WebApplicationException** exception type). These exception types can be used to wrap standard HTTP status codes, either for HTTP client errors (HTTP **4xx** status codes), or HTTP server errors (HTTP **5xx** status codes).

Exception hierarchy

Figure 50.1, “JAX-RS 2.0 Application Exception Hierarchy” shows the hierarchy of application level exceptions supported in JAX-RS 2.0.

Figure 50.1. JAX-RS 2.0 Application Exception Hierarchy



WebApplicationException class

The [javax.ws.rs.WebApplicationException](#) exception class (which has been available since JAX-RS 1.x) is at the base of the JAX-RS 2.0 exception hierarchy, and is described in detail in [Section 50.2, “Using WebApplicationException exceptions to report”](#).

ClientErrorException class

The [javax.ws.rs.ClientErrorException](#) exception class is used to encapsulate HTTP client errors (HTTP **4xx** status codes). In your application code, you can throw this exception or one of its subclasses.

ServerErrorException class

The [javax.ws.rs.ServerErrorException](#) exception class is used to encapsulate HTTP server errors (HTTP **5xx** status codes). In your application code, you can throw this exception or one of its subclasses.

RedirectionException class

The [javax.ws.rs.RedirectionException](#) exception class is used to encapsulate HTTP request redirection (HTTP **3xx** status codes). The constructors of this class take a URI argument, which specifies the redirect location. The redirect URI is accessible through the `getLocation()` method. Normally, HTTP redirection is transparent on the client side.

Client exception subclasses

You can raise the following HTTP client exceptions (HTTP **4xx** status codes) in a JAX-RS 2.0 application:

BadRequestException

Encapsulates the [400 Bad Request](#) HTTP error status.

ForbiddenException

Encapsulates the [403 Forbidden](#) HTTP error status.

NotAcceptableException

Encapsulates the [406 Not Acceptable](#) HTTP error status.

NotAllowedException

Encapsulates the [405 Method Not Allowed](#) HTTP error status.

NotAuthorizedException

Encapsulates the [401 Unauthorized](#) HTTP error status. This exception could be raised in either of the following cases:

- The client did not send the required credentials (in a HTTP **Authorization** header), or
- The client presented the credentials, but the credentials were not valid.

NotFoundException

Encapsulates the [404 Not Found](#) HTTP error status.

NotSupportedException

Encapsulates the [415 Unsupported Media Type](#) HTTP error status.

Server exception subclasses

You can raise the following HTTP server exceptions (HTTP **5xx** status codes) in a JAX-RS 2.0 application:

InternalServerErrorException

Encapsulates the [500 Internal Server Error](#) HTTP error status.

ServiceUnavailableException

Encapsulates the [503 Service Unavailable](#) HTTP error status.

50.4. MAPPING EXCEPTIONS TO RESPONSES

Overview

There are instances where throwing a `WebApplicationException` exception is impractical or impossible. For example, you may not want to catch all possible exceptions and then create a `WebApplicationException` for them. You may also want to use custom exceptions that make working with your application code easier.

To handle these cases the JAX-RS API allows you to implement a custom exception provider that generates a **Response** object to send to a client. Custom exception providers are created by implementing the `ExceptionMapper<E>` interface. When registered with the Apache CXF runtime, the custom provider will be used whenever an exception of type **E** is thrown.

How exception mappers are selected

Exception mappers are used in two cases:

- When any exception or one of its subclasses, is thrown, the runtime will check for an appropriate exception mapper. An exception mapper is selected if it handles the specific exception thrown. If there is not an exception mapper for the specific exception that was thrown, the exception mapper for the nearest superclass of the exception is selected.
- By default, a `WebApplicationException` will be handled by the default mapper, **WebApplicationExceptionMapper**. Even if an additional custom mapper is registered, which could potentially handle a `WebApplicationException` exception (for example, a custom **RuntimeException** mapper), the custom mapper will **not** be used and the **WebApplicationExceptionMapper** will be used instead.

This behaviour can be changed, however, by setting the `default.wae.mapper.least.specific` property to `true` on a **Message** object. When this option is enabled, the default **WebApplicationExceptionMapper** is relegated to the lowest priority, so that it becomes possible to handle a `WebApplicationException` exception with a custom exception mapper. For example, if this option is enabled, it would be possible to catch a `WebApplicationException` exception by registering a custom **RuntimeException** mapper. See the section called “[Registering an exception mapper for WebApplicationException](#)”.

If an exception mapper is not found for an exception, the exception is wrapped in an `ServletException` exception and passed onto the container runtime. The container runtime will then determine how to handle the exception.

Implementing an exception mapper

Exception mappers are created by implementing the `javax.ws.rs.ext.ExceptionMapper<E>` interface. As shown in [Example 50.5, “Exception mapper interface”](#), the interface has a single method, **toResponse()**, that takes the original exception as a parameter and returns a **Response** object.

Example 50.5. Exception mapper interface

```
public interface ExceptionMapper<E extends java.lang.Throwable>
{
    public Response toResponse(E exception);
}
```

The **Response** object created by the exception mapper is processed by the runtime just like any other **Response** object. The resulting response to the consumer will contain the status, headers, and entity body encapsulated in the **Response** object.

Exception mapper implementations are considered providers by the runtime. Therefore they must be decorated with the **@Provider** annotation.

If an exception occurs while the exception mapper is building the **Response** object, the runtime will send a response with a status of **500 Server Error** to the consumer.

[Example 50.6, “Mapping an exception to a response”](#) shows an exception mapper that intercepts Spring `AccessDeniedException` exceptions and generates a response with a **403 Forbidden** status and an empty entity body.

Example 50.6. Mapping an exception to a response

```
import javax.ws.rs.core.Response;
```

```

import javax.ws.rs.ext.ExceptionMapper;

import org.springframework.security.AccessDeniedException;

@Provider
public class SecurityExceptionMapper implements ExceptionMapper<AccessDeniedException>
{

    public Response toResponse(AccessDeniedException exception)
    {
        return Response.status(Response.Status.FORBIDDEN).build();
    }

}

```

The runtime will catch any AccessDeniedException exceptions and create a **Response** object with no entity body and a status of **403**. The runtime will then process the **Response** object as it would for a normal response. The result is that the consumer will receive an HTTP response with a status of **403**.

Registering an exception mapper

Before a JAX-RS application can use an exception mapper, the exception mapper must be registered with the runtime. Exception mappers are registered with the runtime using the **jaxrs:providers** element in the application's configuration file.

The **jaxrs:providers** element is a child of the **jaxrs:server** element and contains a list of **bean** elements. Each **bean** element defines one exception mapper.

[Example 50.7, “Registering exception mappers with the runtime”](#) shows a JAX-RS server configured to use a custom exception mapper, **SecurityExceptionMapper**.

Example 50.7. Registering exception mappers with the runtime

```

<beans ...>
    <jaxrs:server id="customerService" address="/">
        ...
        <jaxrs:providers>
            <bean id="securityException" class="com.bar.providers.SecurityExceptionMapper"/>
        </jaxrs:providers>
    </jaxrs:server>
</beans>

```

Registering an exception mapper for WebApplicationException

Registering an exception mapper for a **WebApplicationException** exception is a special case, because this exception type is automatically handled by the default **WebApplicationExceptionMapper**. Normally, even when you register a custom mapper that you would expect to handle **WebApplicationException**, it will continue to be handled by the default **WebApplicationExceptionMapper**. To change this default behaviour, you need to set the **default.wae.mapper.least.specific** property to **true**.

For example, the following XML code shows how to enable the **default.wae.mapper.least.specific** property on a JAX-RS endpoint:

```
<beans ...>
<jaxrs:server id="customerService" address="/">
...
<jaxrs:providers>
<bean id="securityException" class="com.bar.providers.SecurityExceptionMapper"/>
</jaxrs:providers>
<jaxrs:properties>
<entry key="default.wae.mapper.least.specific" value="true"/>
</jaxrs:properties>
</jaxrs:server>
</beans>
```

You can also set the **default.wae.mapper.least.specific** property in an interceptor, as shown in the following example:

```
// Java
public void handleMessage(Message message)
{
    m.put("default.wae.mapper.least.specific", true);
    ...
}
```

CHAPTER 51. ENTITY SUPPORT

Abstract

The Apache CXF runtime supports a limited number of mappings between MIME types and Java objects out of the box. Developers can extend the mappings by implementing custom readers and writers. The custom readers and writers are registered with the runtime at start-up.

OVERVIEW

The runtime relies on JAX-RS MessageBodyReader and MessageBodyWriter implementations to serialize and de-serialize data between the HTTP messages and their Java representations. The readers and writers can restrict the MIME types they are capable of processing.

The runtime provides readers and writers for a number of common mappings. If an application requires more advanced mappings, a developer can provide custom implementations of the MessageBodyReader interface and/or the MessageBodyWriter interface. Custom readers and writers are registered with the runtime when the application is started.

NATIVELY SUPPORTED TYPES

[Table 51.1, “Natively supported entity mappings”](#) lists the entity mappings provided by Apache CXF out of the box.

Table 51.1. Natively supported entity mappings

Java Type	MIME Type
primitive types	text/plain
java.lang.Number	text/plain
byte[]	*/*
java.lang.String	*/*
java.io.InputStream	*/*
java.io.Reader	*/*
java.io.File	*/*
javax.activation.DataSource	*/*
javax.xml.transform.Source	text/xml, application/xml, application/*+xml
javax.xml.bind.JAXBElement	text/xml, application/xml, application/*+xml
JAXB annotated objects	text/xml, application/xml, application/*+xml

Java Type	MIME Type
javax.ws.rs.core.MultivaluedMap<String, String>	application/x-www-form-urlencoded [a]
javax.ws.rs.core.StreamingOutput	*/* [b]

[a] This mapping is used for handling HTML form data.

[b] This mapping is only supported for returning data to a consumer.

CUSTOM READERS

Custom entity readers are responsible for mapping incoming HTTP requests into a Java type that a service's implementation can manipulate. They implement the javax.ws.rs.ext.MessageBodyReader interface.

The interface, shown in [Example 51.1, “Message reader interface”](#), has two methods that need implementing:

Example 51.1. Message reader interface

```
package javax.ws.rs.ext;

public interface MessageBodyReader<T>
{
    public boolean isReadable(java.lang.Class<?> type,
                           java.lang.reflect.Type genericType,
                           java.lang.annotation.Annotation[] annotations,
                           javax.ws.rs.core.MediaType mediaType);

    public T readFrom(java.lang.Class<T> type,
                     java.lang.reflect.Type genericType,
                     java.lang.annotation.Annotation[] annotations,
                     javax.ws.rs.core.MediaType mediaType,
                     javax.ws.rs.core.MultivaluedMap<String, String> httpHeaders,
                     java.io.InputStream entityStream)
    throws java.io.IOException, WebApplicationException;
}
```

isReadable()

The **isReadable()** method determines if the reader is capable of reading the data stream and creating the proper type of entity representation. If the reader can create the proper type of entity the method returns **true**.

[Table 51.2, “Parameters used to determine if a reader can produce an entity”](#) describes the **isReadable()** method's parameters.

Table 51.2. Parameters used to determine if a reader can produce an entity

Parameter	Type	Description
type	Class<T>	Specifies the actual Java class of the object used to store the entity.
genericType	Type	Specifies the Java type of the object used to store the entity. For example, if the message body is to be converted into a method parameter, the value will be the type of the method parameter as returned by the Method.getGenericParameterTypes() method.
annotations	Annotation[]	Specifies the list of annotations on the declaration of the object created to store the entity. For example if the message body is to be converted into a method parameter, this will be the annotations on that parameter returned by the Method.getParameterAnnotations() method.
mediaType	MediaType	Specifies the MIME type of the HTTP entity.

readFrom()

The **readFrom()** method reads the HTTP entity and converts it into the desired Java object. If the reading is successful the method returns the created Java object containing the entity. If an error occurs when reading the input stream the method should throw an IOException exception. If an error occurs that requires an HTTP error response, an WebApplicationException with the HTTP response should be thrown.

[Table 51.3, "Parameters used to read an entity"](#) describes the **readFrom()** method's parameters.

Table 51.3. Parameters used to read an entity

Parameter	Type	Description
type	Class<T>	Specifies the actual Java class of the object used to store the entity.

Parameter	Type	Description
genericType	Type	Specifies the Java type of the object used to store the entity. For example, if the message body is to be converted into a method parameter, the value will be the type of the method parameter as returned by the Method.getGenericParameterTypes() method.
annotations	Annotation[]	Specifies the list of annotations on the declaration of the object created to store the entity. For example if the message body is to be converted into a method parameter, this will be the annotations on that parameter returned by the Method.getParameterAnnotations() method.
mediaType	MediatType	Specifies the MIME type of the HTTP entity.
httpHeaders	MultivaluedMap<String, String>	Specifies the HTTP message headers associated with the entity.
entityStream	InputStream	Specifies the input stream containing the HTTP entity.



IMPORTANT

This method should not close the input stream.

Before an MessageBodyReader implementation can be used as an entity reader, it must be decorated with the **javax.ws.rs.ext.Provider** annotation. The **@Provider** annotation alerts the runtime that the supplied implementation provides additional functionality. The implementation must also be registered with the runtime as described in [the section called “Registering readers and writers”](#).

By default a custom entity provider handles all MIME types. You can limit the MIME types that a custom entity reader will handle using the **javax.ws.rs.Consumes** annotation. The **@Consumes** annotation specifies a comma separated list of MIME types that the custom entity provider reads. If an entity is not of a specified MIME type, the entity provider will not be selected as a possible reader.

[Example 51.2, “XML source entity reader”](#) shows an entity reader the consumes XML entities and stores them in a Source object.

Example 51.2. XML source entity reader

```
import java.io.IOException;
import java.io.InputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

import javax.ws.rs.Consumes;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.ext.MessageBodyReader;
import javax.ws.rs.ext.Provider;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Source;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamSource;

import org.w3c.dom.Document;
import org.apache.cxf.jaxrs.ext.xml.XMLSource;

@Provider
@Consumes({"application/xml", "application/*+xml", "text/xml", "text/html" })
public class SourceProvider implements MessageBodyReader<Object>
{
    public boolean isReadable(Class<?> type,
                             Type genericType,
                             Annotation[] annotations,
                             MediaType mt)
    {
        return Source.class.isAssignableFrom(type) || XMLSource.class.isAssignableFrom(type);
    }

    public Object readFrom(Class<Object> source,
                          Type genericType,
                          Annotation[] annotations,
                          MediaType mediaType,
                          MultivaluedMap<String, String> httpHeaders,
                          InputStream is)
    throws IOException
    {
        if (DOMSource.class.isAssignableFrom(source))
        {
            Document doc = null;
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder;
            try
            {
                builder = factory.newDocumentBuilder();
                doc = builder.parse(is);
            }
            catch (Exception e)
            {
                IOException ioex = new IOException("Problem creating a Source object");
                ioex.setStackTrace(e.getStackTrace());
                throw ioex;
            }
        }
    }
}
```

```

        return new DOMSource(doc);
    }
    else if (StreamSource.class.isAssignableFrom(source) ||
Source.class.isAssignableFrom(source))
    {
        return new StreamSource(is);
    }
    else if (XMLSource.class.isAssignableFrom(source))
    {
        return new XMLSource(is);
    }

    throw new IOException("Unrecognized source");
}
}

```

CUSTOM WRITERS

Custom entity writers are responsible for mapping Java types into HTTP entities. They implement the javax.ws.rs.ext.MessageBodyWriter interface.

The interface, shown in [Example 51.3, “Message writer interface”](#), has three methods that need implementing:

Example 51.3. Message writer interface

```

package javax.ws.rs.ext;

public interface MessageBodyWriter<T>
{
    public boolean isWriteable(java.lang.Class<?> type,
        java.lang.reflect.Type genericType,
        java.lang.annotation.Annotation[] annotations,
        javax.ws.rs.core.MediaType mediaType);

    public long getSize(T t,
        java.lang.Class<?> type,
        java.lang.reflect.Type genericType,
        java.lang.annotation.Annotation[] annotations,
        javax.ws.rs.core.MediaType mediaType);

    public void writeTo(T t,
        java.lang.Class<?> type,
        java.lang.reflect.Type genericType,
        java.lang.annotation.Annotation[] annotations,
        javax.ws.rs.core.MediaType mediaType,
        javax.ws.rs.core.MultivaluedMap<String, Object> httpHeaders,
        java.io.OutputStream entityStream)
    throws java.io.IOException, WebApplicationException;
}

```

isWritable()

The **isWritable()** method determines if the entity writer can map the Java type to the proper entity type. If the writer can do the mapping, the method returns **true**.

[Table 51.4, "Parameters used to read an entity"](#) describes the **isWritable()** method's parameters.

Table 51.4. Parameters used to read an entity

Parameter	Type	Description
type	Class<T>	Specifies the Java class of the object being written.
genericType	Type	Specifies the Java type of object to be written, obtained either by reflection of a resource method return type or via inspection of the returned instance. The GenericEntity class, described in Section 48.4, "Returning entities with generic type information" , provides support for controlling this value.
annotations	Annotation[]	Specifies the list of annotations on the method returning the entity.
mediaType	MediatType	Specifies the MIME type of the HTTP entity.

getSize()

The **getSize()** method is called before the **writeTo()**. It returns the length, in bytes, of the entity being written. If a positive value is returned the value is written into the HTTP message's **Content-Length** header.

[Table 51.5, "Parameters used to read an entity"](#) describes the **getSize()** method's parameters.

Table 51.5. Parameters used to read an entity

Parameter	Type	Description
t	generic	Specifies the instance being written.
type	Class<T>	Specifies the Java class of the object being written.

Parameter	Type	Description
genericType	Type	Specifies the Java type of object to be written, obtained either by reflection of a resource method return type or via inspection of the returned instance. The GenericEntity class, described in Section 48.4, “Returning entities with generic type information” , provides support for controlling this value.
annotations	Annotation[]	Specifies the list of annotations on the method returning the entity.
mediaType	MediaType	Specifies the MIME type of the HTTP entity.

writeTo()

The **writeTo()** method converts a Java object into the desired entity type and writes the entity to the output stream. If an error occurs when writing the entity to the output stream the method should throw an IOException exception. If an error occurs that requires an HTTP error response, an WebApplicationException with the HTTP response should be thrown.

[Table 51.6, “Parameters used to read an entity”](#) describes the **writeTo()** method’s parameters.

Table 51.6. Parameters used to read an entity

Parameter	Type	Description
t	generic	Specifies the instance being written.
type	Class<T>	Specifies the Java class of the object being written.
genericType	Type	Specifies the Java type of object to be written, obtained either by reflection of a resource method return type or via inspection of the returned instance. The GenericEntity class, described in Section 48.4, “Returning entities with generic type information” , provides support for controlling this value.

Parameter	Type	Description
annotations	Annotation[]	Specifies the list of annotations on the method returning the entity.
mediaType	MediaType	Specifies the MIME type of the HTTP entity.
httpHeaders	MultivaluedMap<String, Object>	Specifies the HTTP response headers associated with the entity.
entityStream	OutputStream	Specifies the output stream into which the entity is written.

Before a MessageBodyWriter implementation can be used as an entity writer, it must be decorated with the **javax.ws.rs.ext.Provider** annotation. The **@Provider** annotation alerts the runtime that the supplied implementation provides additional functionality. The implementation must also be registered with the runtime as described in [the section called “Registering readers and writers”](#).

By default a custom entity provider handles all MIME types. You can limit the MIME types that a custom entity writer will handle using the **javax.ws.rs.Produces** annotation. The **@Produces** annotation specifies a comma separated list of MIME types that the custom entity provider generates. If an entity is not of a specified MIME type, the entity provider will not be selected as a possible writer.

[Example 51.4, “XML source entity writer”](#) shows an entity writer that takes Source objects and produces XML entities.

Example 51.4. XML source entity writer

```

import java.io.IOException;
import java.io.OutputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

import javax.ws.rs.Produces;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.ext.MessageBodyWriter;
import javax.ws.rs.ext.Provider;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;

import org.apache.cxf.jaxrs.ext.xml.XMLSource;

```

```

@Provider
@Produces({"application/xml", "application/*+xml", "text/xml"})
public class SourceProvider implements MessageBodyWriter<Source>
{

    public boolean isWriteable(Class<?> type,
                               Type genericType,
                               Annotation[] annotations,
                               MediaType mt)
    {
        return Source.class.isAssignableFrom(type);
    }

    public void writeTo(Source source,
                        Class<?> clazz,
                        Type genericType,
                        Annotation[] annotations,
                        MediaType mediatype,
                        MultivaluedMap<String, Object> httpHeaders,
                        OutputStream os)
        throws IOException
    {
        StreamResult result = new StreamResult(os);
        TransformerFactory tf = TransformerFactory.newInstance();
        try
        {
            Transformer t = tf.newTransformer();
            t.transform(source, result);
        }
        catch (TransformerException te)
        {
            te.printStackTrace();
            throw new WebApplicationException(te);
        }
    }

    public long getSize(Source source,
                        Class<?> type,
                        Type genericType,
                        Annotation[] annotations,
                        MediaType mt)
    {
        return -1;
    }
}

```

REGISTERING READERS AND WRITERS

Before a JAX-RS application can use any custom entity providers, the custom providers must be registered with the runtime. Providers are registered with the runtime using either the **jaxrs:providers** element in the application's configuration file or using the **JAXRSServerFactoryBean** class.

The **jaxrs:providers** element is a child of the **jaxrs:server** element and contains a list of **bean** elements. Each **bean** element defines one entity provider.

[Example 51.5, “Registering entity providers with the runtime”](#) show a JAX-RS server configured to use a set of custom entity providers.

Example 51.5. Registering entity providers with the runtime

```
<beans ...>
<jaxrs:server id="customerService" address="/">
...
<jaxrs:providers>
<bean id="isProvider" class="com.bar.providers.InputStreamProvider"/>
<bean id="longProvider" class="com.bar.providers.LongProvider"/>
</jaxrs:providers>
</jaxrs:server>
</beans>
```

The **JAXRSServerFactoryBean** class is a Apache CXF extension that provides access to the configuration APIs. It has a **setProvider()** method that allows you to add instantiated entity providers to an application. [Example 51.6, “Programmatically registering an entity provider”](#) shows code for registering an entity provider programmatically.

Example 51.6. Programmatically registering an entity provider

```
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
...
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
...
SourceProvider provider = new SourceProvider();
sf.setProvider(provider);
...
```

CHAPTER 52. GETTING AND USING CONTEXT INFORMATION

Abstract

Context information includes detailed information about a resource's URI, the HTTP headers, and other details that are not readily available using the other injection annotations. Apache CXF provides special class that amalgamates the all possible context information into a single object.

52.1. INTRODUCTION TO CONTEXTS

Context annotation

You specify that context information is to be injected into a field or a resource method parameter using the **javax.ws.rs.core.Context** annotation. Annotating a field or parameter of one of the context types will instruct the runtime to inject the appropriate context information into the annotated field or parameter.

Types of contexts

[Table 52.1, "Context types"](#) lists the types of context information that can be injected and the objects that support them.

Table 52.1. Context types

Object	Context information
UriInfo	The full request URI
HttpHeaders	The HTTP message headers
Request	Information that can be used to determine the best representation variant or to determine if a set of preconditions have been set
SecurityContext	Information about the security of the requester including the authentication scheme in use, if the request channel is secure, and the user principle

Where context information can be used

Context information is available to the following parts of a JAX-RS application:

- resource classes
- resource methods
- entity providers
- exception mappers

Scope

All context information injected using the **@Context** annotation is specific to the current request. This is true in all cases including entity providers and exception mappers.

Adding contexts

The JAX-RS framework allows developers to extend the types of information that can be injected using the context mechanism. You add custom contexts by implementing a Context<T> object and registering it with the runtime.

52.2. WORKING WITH THE FULL REQUEST URI

Abstract

The request URI contains a significant amount of information. Most of this information can be accessed using method parameters as described in [Section 47.2.2, “Injecting data from a request URI”](#), however using parameters forces certain constraints on how the URI is processed. Using parameters to access the segments of a URI also does not provide a resource access to the full request URI.

You can provide access to the complete request URI by injecting the URI context into a resource. The URI is provided as a UriInfo object. The UriInfo interface provides functions for decomposing the URI in a number of ways. It can also provide the URI as a UriBuilder object that allows you to construct URIs to return to clients.

:experimental:

52.2.1. Injecting the URI information

Overview

When a class field or method parameter that is a UriInfo object is decorated with the **@Context** annotation, the URI context for the current request is injected into the UriInfo object.

Example

[Injecting the URI context into a class field](#) shows a class with a field populated by injecting the URI context.

Injecting the URI context into a class field

```
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.Path;
...
@Path("/monstersforhire/")
public class MonsterService
{
    @Context
    UriInfo requestURI;
    ...
}
```

52.2.2. Working with the URI

Overview

One of the main advantages of using the URI context is that it provides access to the base URI of the service and the path segment of the URI for the selected resource. This information can be useful for a number of purposes such as making processing decisions based on the URI or calculating URIs to return as part of the response. For example if the base URI of the request contains a **.com** extension the service may decide to use US dollars and if the base URI contains a **.co.uk** extension is may decide to use British Pounds.

The UriInfo interface provides methods for accessing the parts of the URI:

- the base URI
- the resource path
- the full URI

Getting the Base URI

The *base URI* is the root URI on which the service is published. It does not contain any portion of the URI specified in any of the service's **@Path** annotations. For example if a service implementing the resource defined in [Example 47.5, "Disabling URI decoding"](#) were published to <http://fusesource.org> and a request was made on <http://fusesource.org/monstersforhire/nightstalker?12> the base URI would be <http://fusesource.org>.

[Table 52.2, "Methods for accessing a resource's base URI"](#) describes the methods that return the base URI.

Table 52.2. Methods for accessing a resource's base URI

Method	Description
URIgetBaseUri	Returns the service's base URI as a URI object.
UriBuildergetBaseUriBuilder	Returns the base URI as a javax.ws.rs.core.UriBuilder object. The UriBuilder class is useful for creating URLs for other resources implemented by the service.

Getting the path

The *path* portion of the request URI is the portion of the URI that was used to select the current resource. It does not include the base URI, but does include any URI template variable and matrix parameters included in the URI.

The value of the path depends on the resource selected. For example, the paths for the resources defined in [Getting a resource's path](#) would be:

- **rootPath** – `/monstersforhire/`
- **getterPath** – `/monstersforhire/nightstalker`
The **GET** request was made on `/monstersforhire/nightstalker`.
- **putterPath** – `/monstersforhire/911`
The **PUT** request was made on `/monstersforhire/911`.

Getting a resource's path

```

@Path("/monstersforhire/")
public class MonsterService
{
    @Context
    UriInfo rootUri;

    ...

    @GET
    public List<Monster> getMonsters(@Context UriInfo getUri)
    {
        String rootPath = rootUri.getPath();
        ...
    }

    @GET
    @Path("/{type}")
    public Monster getMonster(@PathParam("type") String type,
                             @Context UriInfo getUri)
    {
        String getterPath = getUri.getPath();
        ...
    }

    @PUT
    @Path("/{id}")
    public void addMonster(@Encoded @PathParam("type") String type,
                          @Context UriInfo putUri)
    {
        String putterPath = putUri.getPath();
        ...
    }
    ...
}

```

[Table 52.3, “Methods for accessing a resource’s path”](#) describes the methods that return the resource path.

Table 52.3. Methods for accessing a resource’s path

Method	Description
StringgetPath	Returns the resource’s path as a decoded URI.
StringgetPathbooleandecode	Returns the resource’s path. Specifying false disables URI decoding.

Method	Description
List<PathSegment>getPathSegments	Returns the decoded path as a list of javax.ws.rs.core.PathSegment objects. Each portion of the path, including matrix parameters, is placed into a unique entry in the list. For example the resource path box/round#tall would result in a list with three entries: box , round , and tall .
List<PathSegment>getPathSegments(boolean decode)	Returns the path as a list of javax.ws.rs.core.PathSegment objects. Each portion of the path, including matrix parameters, is placed into a unique entry in the list. Specifying false disables URI decoding. For example the resource path box#tall/round would result in a list with three entries: box , tall , and round .

Getting the full request URI

[Table 52.4, “Methods for accessing the full request URI”](#) describes the methods that return the full request URI. You have the option of returning the request URI or the absolute path of the resource. The difference is that the request URI includes the any query parameters appended to the URI and the absolute path does not include the query parameters.

Table 52.4. Methods for accessing the full request URI

Method	Description
URIgetRequestUri	Returns the complete request URI, including query parameters and matrix parameters, as a java.net.URI object.
UriBuildergetRequestUriBuilder	Returns the complete request URI, including query parameters and matrix parameters, as a javax.ws.rs.UriBuilder object. The UriBuilder class is useful for creating URIs for other resources implemented by the service.
URIgetAbsolutePath	Returns the complete request URI, including matrix parameters, as a java.net.URI object. The absolute path does not include query parameters.
UriBuildergetAbsolutePathBuilder	Returns the complete request URI, including matrix parameters, as a javax.ws.rs.UriBuilder object. The absolute path does not include query parameters.

For a request made using the URI <http://fusesource.org/monstersforhire/nightstalker?12>, the `getRequestUri()` methods would return <http://fusesource.org/monstersforhire/nightstalker?12>. The `getAbsolutePath()` method would return <http://fusesource.org/monstersforhire/nightstalker>.

52.2.3. Getting the value of URI template variables

Overview

As described in the section called “[Setting the path](#)”, resource paths can contain variable segments that are bound to values dynamically. Often these variable path segments are used as parameters to a resource method as described in the section called “[Getting data from the URI’s path](#)”. You can, however, also access them through the URI context.

Methods for getting the path parameters

The `UriInfo` interface provides two methods, shown in [Example 52.1, “Methods for returning path parameters from the URI context”](#), that return a list of the path parameters.

Example 52.1. Methods for returning path parameters from the URI context

```
MultivaluedMap<java.lang.String,  
java.lang.String>getPathParametersMultivaluedMap<java.lang.String,  
java.lang.String>getPathParametersbooleandecode
```

The `getPathParameters()` method that does not take any parameters automatically decodes the path parameters. If you want to disable URI decoding use `getPathParameters(false)`.

The values are stored in the map using their template identifiers as keys. For example if the URI template for the resource is `/{color}/box/{note}` the returned map will have two entries with the keys `color` and `note`.

Example

[Example 52.2, “Extracting path parameters from the URI context”](#) shows code for retrieving the path parameters using the URI context.

Example 52.2. Extracting path parameters from the URI context

```
import javax.ws.rs.Path;  
import javax.ws.rs.Get;  
import javax.ws.rs.core.Context;  
import javax.ws.rs.core.UriInfo;  
import javax.ws.rs.core.MultivaluedMap;  
  
@Path("/monstersforhire/")  
public class MonsterService  
  
    @GET  
    @Path("/{type}/{size}")  
    public Monster getMonster(@Context UriInfo uri)  
    {  
        MultivaluedMap paramMap = uri.getPathParameters();  
        String type = paramMap.getFirst("type");  
    }
```

```
    String size = paramMap.getFirst("size");
}
```

CHAPTER 53. ANNOTATION INHERITANCE

Abstract

JAX-RS annotations can be inherited by subclasses and classes implementing annotated interfaces. The inheritance mechanism allows for subclasses and implementation classes to override the annotations inherited from its parents.

OVERVIEW

Inheritance is one of the more powerful mechanisms in Java because it allows developers to create generic objects that can then be specialized to meet particular needs. JAX-RS keeps this power by allowing the annotations used in mapping classes to resources to be inherited from super classes.

JAX-RS's annotation inheritance also extends to support for interfaces. Implementation classes inherit the JAX-RS annotations used in the interface they implement.

The JAX-RS inheritance rules do provide a mechanism for overriding inherited annotations. However, it is not possible to completely remove JAX-RS annotations from a construct that inherits them from a super class or interface.

INHERITANCE RULES

Resource classes inherit any JAX-RS annotations from the interface(s) it implements. Resource classes also inherit any JAX-RS annotations from any super classes they extend. Annotations inherited from a super class take precedence over annotations inherited from an interface.

In the code sample shown in [Example 53.1, "Annotation inheritance"](#), the **Kaijin** class' **getMonster()** method inherits the **@Path**, **@GET**, and **@PathParam** annotations from the **Kaiju** interface.

Example 53.1. Annotation inheritance

```
public interface Kaiju
{
    @GET
    @Path("/{id}")
    public Monster getMonster(@PathParam("id") int id);
    ...
}

@Path("/kaijin")
public class Kaijin implements Kaiju
{
    public Monster getMonster(int id)
    {
        ...
    }
    ...
}
```

OVERRIDING INHERITED ANNOTATIONS

Overriding inherited annotations is as easy as providing new annotations. If the subclass, or implementation class, provides any of its own JAX-RS annotations for a method then all of the JAX-RS annotations for that method are ignored.

In the code sample shown in [Example 53.2, "Overriding annotation inheritance"](#), the **Kaijin** class' **getMonster()** method does not inherit any of the annotations from the Kaiju interface. The implementation class overrides the **@Produces** annotation which causes all of the annotations from the interface to be ignored.

Example 53.2. Overriding annotation inheritance

```
public interface Kaiju
{
    @GET
    @Path("/{id}")
    @Produces("text/xml");
    public Monster getMonster(@PathParam("id") int id);
    ...
}

@Path("/kaijin")
public class Kaijin implements Kaiju
{
    @GET
    @Path("/{id}")
    @Produces("application/octect-stream");
    public Monster getMonster(@PathParam("id") int id)
    {
        ...
    }
    ...
}
```

CHAPTER 54. EXTENDING JAX-RS ENDPOINTS WITH SWAGGER SUPPORT

Abstract

The CXF Swagger2Feature (**org.apache.cxf.jaxrs.swagger.Swagger2Feature**) allows you to generate Swagger 2.0 documents by extending published JAX-RS service endpoints with a simple configuration.

The Swagger2Feature is supported in both Spring Boot and Karaf implementations.

54.1. SWAGGER2FEATURE OPTIONS

You can use the following options in Swagger2Feature.

Table 54.1. Swagger2Feature operations

Name	Description	Default
basePath	The context root path+ (see also the usePathBasedConfig option)	null
contact	Your contact information+	" users@xf.apache.org "
description	A description+	"The Application"
filterClass	A security filter+	null
host	The host and port information+	null
ignoreRoutes	Excludes specific paths when scanning all resources (see the scanAllResources option)++	null
license	The license+	"Apache 2.0 License"
licenceUrl	The license URL+	http://www.apache.org/licenses/LICENSE-2.0.html
prettyPrint	When generating swagger.json , specifies to pretty-print the JSON document+	false
resourcePackage	A list of comma separated package names where resources must be scanned+	A list of service classes configured at the endpoint
runAsFilter	Runs the feature as a filter	false

Name	Description	Default
scan	Generates the swagger documentation+	true
scanAllResources	Scans all resources including non-annotated JAX-RS resources (see also the ignoreRoutes option)++	false
schemes	The protocol schemes+	null
swaggerUiConfig	Swagger UI configuration	null
termsOfServiceUrl	The terms of service URL+	null
title	The title+	"Sample REST Application"
usePathBasedConfig	Prevents Swagger from caching the value of the basePath option.	false
version	The version+	"1.0.0"

+ The option is defined in Swagger's BeanConfig

++ The option is defined in Swagger's ReaderConfig

54.2. KARAF IMPLEMENTATIONS

This section describes how to use the Swagger2Feature in which REST services are defined inside JAR files and deployed to a Fuse on Karaf container.

54.2.1. Quickstart example

You can download Red Hat Fuse **quickstarts** from the [Fuse Software Downloads](#) page.

The Quickstart zip file contains a `/cxf/rest/` directory for a quickstart that demonstrates how to create a RESTful (JAX-RS) web service using CXF and how to enable Swagger and annotate the JAX-RS endpoints.

54.2.2. Enabling Swagger

Enabling Swagger involves:

- Modifying the XML file that defines the CXF service by adding the CXF class (`org.apache.cxf.jaxrs.swagger.Swagger2Feature`) to the `<jaxrs:server>` definition.
For an example, see [Example 55.4 Example XML file](#).

- In the REST resource class:
 - Importing the Swagger API annotations for each annotation required by the service:

```
import io.swagger.annotations.*
```

where * = **Api**, **ApiOperation**, **ApiParam**, **ApiResponse**, **ApiResponses**, and so on.

For details, go to <https://github.com/swagger-api/swagger-core/wiki/Annotations>.

For an example, see [Example 55.5 Example Resource class](#).

- Adding Swagger annotations to the JAX-RS annotated endpoints (@**PATH**, @**PUT**, @**POST**, @**GET**, @**Produces**, @**Consumes**, @**DELETE**, @**PathParam**, and so on).

For an example, see [Example 55.5 Example Resource class](#).

Example 55.4 Example XML file

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
  xmlns:cxf="http://cxf.apache.org/blueprint/core"
  xsi:schemaLocation="
    http://www.osgi.org/xmlns/blueprint/v1.0.0
    https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
    http://cxf.apache.org/blueprint/jaxrs
    http://cxf.apache.org/schemas/blueprint/jaxrs.xsd
    http://cxf.apache.org/blueprint/core
    http://cxf.apache.org/schemas/blueprint/core.xsd">

  <jaxrs:server id="customerService" address="/crm">
    <jaxrs:serviceBeans>
      <ref component-id="customerSvc"/>
    </jaxrs:serviceBeans>
    <jaxrs:providers>
      <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider"/>
    </jaxrs:providers>
    <jaxrs:features>
      <bean class="org.apache.cxf.jaxrs.swagger.Swagger2Feature">
        <property name="title" value="Fuse:CXF:Quickstarts - Customer Service" />
        <property name="description" value="Sample REST-based Customer Service" />
        <property name="version" value="${project.version}" />
      </bean>
    </jaxrs:features>
  </jaxrs:server>

  <cxf:bus>
    <cxf:features>
      <cxf:logging />
    </cxf:features>
    <cxf:properties>
      <entry key="skip.default.json.provider.registration" value="true" />
    </cxf:properties>
  </cxf:bus>
```

```
<bean id="customerSvc" class="org.jboss.fuse.quickstarts.cxf.rest.CustomerService"/>
</blueprint>
```

Example 55.5 Example Resource class

```

.
.

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.Response;

import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import io.swagger.annotations.ApiParam;
import io.swagger.annotations.ApiResponse;
import io.swagger.annotations.ApiResponses;

.

.

.

@Path("/customerservice/")
@Api(value = "/customerservice", description = "Operations about customerservice")
public class CustomerService {

    private static final Logger LOG =
        LoggerFactory.getLogger(CustomerService.class);

    private MessageContext jaxrsContext;
    private long currentId          = 123;
    private Map<Long, Customer> customers = new HashMap<>();
    private Map<Long, Order> orders     = new HashMap<>();

    public CustomerService() {
        init();
    }

    @GET
    @Path("/customers/{id}/")
    @Produces("application/xml")
    @ApiOperation(value = "Find Customer by ID", notes = "More notes about this
method", response = Customer.class)
    @ApiResponses(value = {
        @ApiResponse(code = 500, message = "Invalid ID supplied"),
        @ApiResponse(code = 204, message = "Customer not found")
```

```

        })
public Customer getCustomer(@ApiParam(value = "ID of Customer to fetch",
    required = true) @PathParam("id") String id) {
    LOG.info("Invoking getCustomer, Customer id is: {}", id);
    long idNumber = Long.parseLong(id);
    return customers.get(idNumber);
}

@PUT
@Path("/customers/")
@Consumes({ "application/xml", "application/json" })
@ApiOperation(value = "Update an existing Customer")
@ApiResponses(value = {
    @ApiResponse(code = 500, message = "Invalid ID supplied"),
    @ApiResponse(code = 204, message = "Customer not found")
})
public Response updateCustomer(@ApiParam(value = "Customer object that needs
    to be updated", required = true) Customer customer) {
    LOG.info("Invoking updateCustomer, Customer name is: {}", customer.getName());
    Customer c = customers.get(customer.getId());
    Response r;
    if (c != null) {
        customers.put(customer.getId(), customer);
        r = Response.ok().build();
    } else {
        r = Response.notModified().build();
    }
    return r;
}

@POST
@Path("/customers/")
@Consumes({ "application/xml", "application/json" })
@ApiOperation(value = "Add a new Customer")
@ApiResponses(value = { @ApiResponse(code = 500, message = "Invalid ID
    supplied"), })
public Response addCustomer(@ApiParam(value = "Customer object that needs to
    be updated", required = true) Customer customer) {
    LOG.info("Invoking addCustomer, Customer name is: {}", customer.getName());
    customer.setId(++currentId);

    customers.put(customer.getId(), customer);
    if (jaxrsContext.getHttpHeaders().getMediaType().getSubtype().equals("json"))
    {
        return Response.ok().type("application/json").entity(customer).build();
    } else {
        return Response.ok().type("application/xml").entity(customer).build();
    }
}

@DELETE
@Path("/customers/{id}/")
@ApiOperation(value = "Delete Customer")
@ApiResponses(value = {

```

```

    @ApiResponse(code = 500, message = "Invalid ID supplied"),
    @ApiResponse(code = 204, message = "Customer not found")
)
public Response deleteCustomer(@ApiParam(value = "ID of Customer to delete",
    required = true) @PathParam("id") String id) {
    LOG.info("Invoking deleteCustomer, Customer id is: {}", id);
    long idNumber = Long.parseLong(id);
    Customer c = customers.get(idNumber);

    Response r;
    if (c != null) {
        r = Response.ok().build();
        customers.remove(idNumber);
    } else {
        r = Response.notModified().build();
    }

    return r;
}

.
.
.

}

```

54.3. SPRING BOOT IMPLEMENTATIONS

This section describes how to use the Swagger2Feature in Spring Boot.

54.3.1. Quickstart example

The Quickstart example (<https://github.com/fabric8-quickstarts/spring-boot-cxf-jaxrs>) demonstrates how you can use Apache CXF with Spring Boot. The Quickstart uses Spring Boot to configure an application that includes a CXF JAX-RS endpoint with Swagger enabled.

54.3.2. Enabling Swagger

Enabling Swagger involves:

- In the REST application:

- Importing Swagger2Feature:

```
import org.apache.cxf.jaxrs.swagger.Swagger2Feature;
```

- Adding Swagger2Feature to a CXF endpoint:

```
endpoint.setFeatures(Arrays.asList(new Swagger2Feature()));
```

For an example, see [Example 55.1 Example REST application](#).

- In the Java implementation file, importing the Swagger API annotations for each annotation required by the service:

```
import io.swagger.annotations.*
```

where * = **Api**, **ApiOperation**, **ApiParam**, **ApiResponse**, **ApiResponses**, and so on.

For details, see <https://github.com/swagger-api/swagger-core/wiki/Annotations>.

For an example, see [Example 55.2 Example Java implementation file](#).

- In the Java file, adding Swagger annotations to the JAX-RS annotated endpoints (@**PATH**, @**PUT**, @**POST**, @**GET**, @**Produces**, @**Consumes**, @**DELETE**, @**PathParam**, and so on).

For an example, see [Example 55.3 Example Java file](#).

Example 55.1 Example REST application

```
package io.fabric8.quickstarts.cxf.jaxrs;

import java.util.Arrays;

import org.apache.cxf.Bus;
import org.apache.cxf.endpoint.Server;
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
import org.apache.cxf.jaxrs.swagger.Swagger2Feature;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class SampleRestApplication {

    @Autowired
    private Bus bus;

    public static void main(String[] args) {
        SpringApplication.run(SampleRestApplication.class, args);
    }

    @Bean
    public Server rsServer() {
        // setup CXF-RS
        JAXRSServerFactoryBean endpoint = new JAXRSServerFactoryBean();
        endpoint.setBus(bus);
        endpoint.setServiceBeans(Arrays.<Object>asList(new HelloServiceImpl()));
        endpoint.setAddress("/");
        endpoint.setFeatures(Arrays.asList(new Swagger2Feature()));
        return endpoint.create();
    }
}
```

Example 55.2 Example Java implementation file

```
import io.swagger.annotations.Api;

@Api("/sayHello")
public class HelloServiceImpl implements HelloService {
```

```

public String welcome() {
    return "Welcome to the CXF RS Spring Boot application, append /{name} to call the hello
service";
}

public String sayHello(String a) {
    return "Hello " + a + ", Welcome to CXF RS Spring Boot World!!!";
}

}

```

Example 55.3 Example Java file

```

package io.fabric8.quickstarts.cxf.jaxrs;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.springframework.stereotype.Service;

@Path("/sayHello")
@Service
public interface HelloService {

    @GET
    @Path("")
    @Produces(MediaType.TEXT_PLAIN)
    String welcome();

    @GET
    @Path("/{a}")
    @Produces(MediaType.TEXT_PLAIN)
    String sayHello(@PathParam("a") String a);

}

```

54.4. ACCESSING SWAGGER DOCUMENTS

When Swagger is enabled by `Swagger2Feature`, the Swagger documents are available at the location URL constructed of the service endpoint location followed by `/swagger.json` or `/swagger.yaml`.

For example, for a JAX-RS endpoint that is published at <http://host:port/context/services> where **context** is a web application context and **/services** is a servlet URL, its Swagger documents are available at <http://host:port/context/services/swagger.json> and <http://host:port/context/services/swagger.yaml>.

If `Swagger2Feature` is active, the CXF Services page links to Swagger documents.

In the above example, you would go to <http://host:port/context/services/services> and then follow a Swagger link which returns a Swagger JSON document.

If CORS support is needed to access the definition from a Swagger UI on another host, you can add the **CrossOriginResourceSharingFilter** from **cxf-rt-rs-security-cors**.

54.5. ACCESSING SWAGGER THROUGH A REVERSE PROXY

If you want to access a Swagger JSON document or a Swagger UI through a reverse proxy, set the following options:

- Set the **CXFServlet use-x-forwarded-headers** init parameter to **true**.

- In Spring Boot, prefix the parameter name with **cxf.servlet.init**:

```
| cxf.servlet.init.use-x-forwarded-headers=true
```

- In Karaf, add the following line to the **installDir/etc/org.apache.cxf.osgi.cfg** configuration file:

```
| cxf.servlet.init.use-x-forwarded-headers=true
```

Note: If you do not already have an **org.apache.cxf.osgi.cfg** file in your **etc** directory, you can create one.

- If you specify a value for the Swagger2Feature **basePath** option and you want to prevent Swagger from caching the **basePath** value, set the Swagger2Feature **usePathBasedConfig** option to **TRUE**:

```
| <bean class="org.apache.cxf.jaxrs.swagger.Swagger2Feature">
|   <property name="usePathBasedConfig" value="TRUE" />
| </bean>
```

PART VII. DEVELOPING APACHE CXF INTERCEPTORS

This guide describes how to write Apache CXF interceptors that can perform pre and post processing on messages.

CHAPTER 55. INTERCEPTORS IN THE APACHE CXF RUNTIME

Abstract

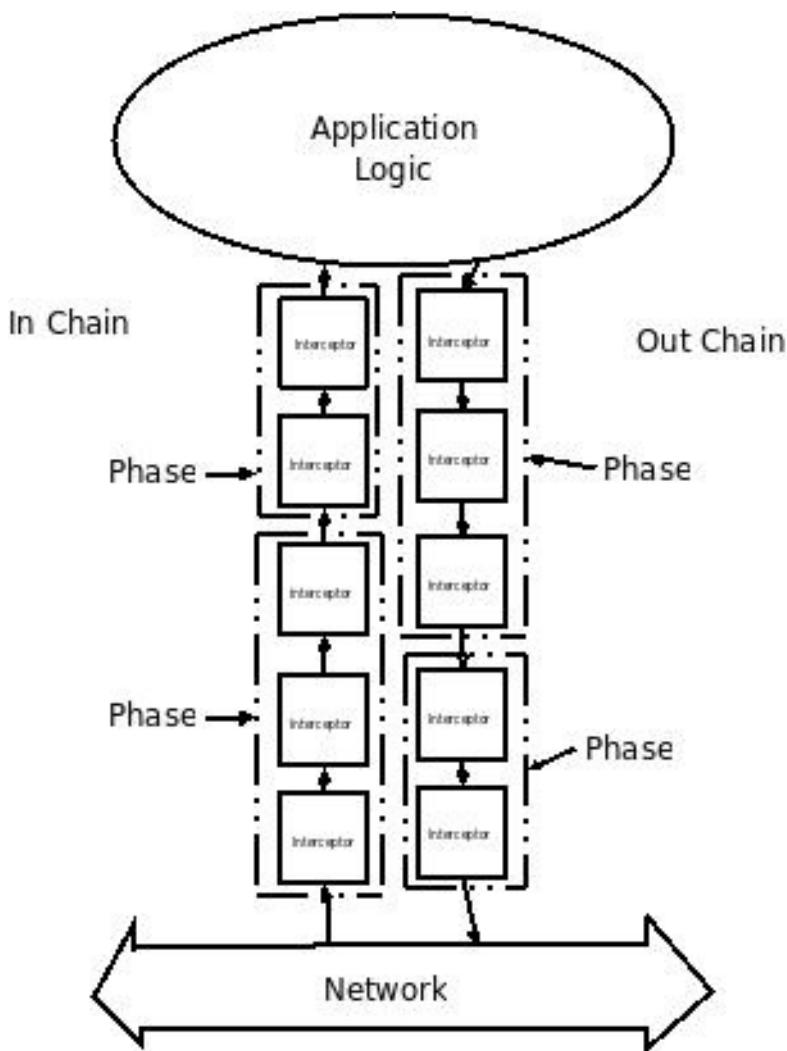
Most of the functionality in the Apache CXF runtime is implemented by interceptors. Every endpoint created by the Apache CXF runtime has three potential interceptor chains for processing messages. The interceptors in these chains are responsible for transforming messages between the raw data transported across the wire and the Java objects handled by the endpoint's implementation code. The interceptors are organized into phases to ensure that processing happens on the proper order.

OVERVIEW

A large part of what Apache CXF does entails processing messages. When a consumer makes a invocation on a remote service the runtime needs to marshal the data into a message the service can consume and place it on the wire. The service provider must unmarshal the message, execute its business logic, and marshal the response into the appropriate message format. The consumer must then unmarshal the response message, correlate it to the proper request, and pass it back to the consumer's application code. In addition to the basic marshaling and unmarshaling, the Apache CXF runtime may do a number of other things with the message data. For example, if WS-RM is activated, the runtime must process the message chunks and acknowledgement messages before marshaling and unmarshaling the message. If security is activated, the runtime must validate the message's credentials as part of the message processing sequence.

Figure 55.1, "Apache CXF interceptor chains" shows the basic path that a request message takes when it is received by a service provider.

Figure 55.1. Apache CXF interceptor chains



MESSAGE PROCESSING IN APACHE CXF

When a Apache CXF developed consumer invokes a remote service the following message processing sequence is started:

1. The Apache CXF runtime creates an outbound interceptor chain to process the request.
2. If the invocation starts a two-way message exchange, the runtime creates an inbound interceptor chain and a fault processing interceptor chain.
3. The request message is passed sequentially through the outbound interceptor chain. Each interceptor in the chain performs some processing on the message. For example, the Apache CXF supplied SOAP interceptors package the message in a SOAP envelope.
4. If any of the interceptors on the outbound chain create an error condition the chain is unwound and control is returned to the application level code.
An interceptor chain is unwound by calling the fault processing method on all of the previously invoked interceptors.
5. The request is dispatched to the appropriate service provider.
6. When the response is received, it is passed sequentially through the inbound interceptor chain.

**NOTE**

If the response is an error message, it is passed into the fault processing interceptor chain.

7. If any of the interceptors on the inbound chain create an error condition, the chain is unwound.
8. When the message reaches the end of the inbound interceptor chain, it is passed back to the application code.

When a Apache CXF developed service provider receives a request from a consumer, a similar process takes place:

1. The Apache CXF runtime creates an inbound interceptor chain to process the request message.
2. If the request is part of a two-way message exchange, the runtime also creates an outbound interceptor chain and a fault processing interceptor chain.
3. The request is passed sequentially through the inbound interceptor chain.
4. If any of the interceptors on the inbound chain create an error condition, the chain is unwound and a fault is dispatched to the consumer.
An interceptor chain is unwound by calling the fault processing method on all of the previously invoked interceptors.
5. When the request reaches the end of the inbound interceptor chain, it is passed to the service implementation.
6. When the response is ready it is passed sequentially through the outbound interceptor chain.

**NOTE**

If the response is an exception, it is passed through the fault processing interceptor chain.

7. If any of the interceptors on the outbound chain create an error condition, the chain is unwound and a fault message is dispatched.
8. Once the request reaches the end of the outbound chain, it is dispatched to the consumer.

INTERCEPTORS

All of the message processing in the Apache CXF runtime is done by *interceptors*. Interceptors are POJOs that have access to the message data before it is passed to the application layer. They can do a number of things including: transforming the message, stripping headers off of the message, or validating the message data. For example, an interceptor could read the security headers off of a message, validate the credentials against an external security service, and decide if message processing can continue.

The message data available to an interceptor is determined by several factors:

- the interceptor's chain
- the interceptor's phase

- the other interceptors that occur earlier in the chain

PHASES

Interceptors are organized into *phases*. A phase is a logical grouping of interceptors with common functionality. Each phase is responsible for a specific type of message processing. For example, interceptors that process the marshaled Java objects that are passed to the application layer would all occur in the same phase.

INTERCEPTOR CHAINS

Phases are aggregated into *interceptor chains*. An interceptor chain is a list of interceptor phases that are ordered based on whether messages are inbound or outbound.

Each endpoint created using Apache CXF has three interceptor chains:

- a chain for inbound messages
- a chain for outbound messages
- a chain for error messages

Interceptor chains are primarily constructed based on the choose of binding and transport used by the endpoint. Adding other runtime features, such as security or logging, also add interceptors to the chains. Developers can also add custom interceptors to a chain using configuration.

DEVELOPING INTERCEPTORS

Developing an interceptor, regardless of its functionality, always follows the same basic procedure:

1. [Chapter 56, The Interceptor APIs](#)

Apache CXF provides a number of abstract interceptors to make it easier to develop custom interceptors.

2. [Section 57.2, "Specifying an interceptor's phase"](#)

Interceptors require certain parts of a message to be available and require the data to be in a certain format. The contents of the message and the format of the data is partially determined by an interceptor's phase.

3. [Section 57.3, "Constraining an interceptors placement in a phase"](#)

In general, the ordering of interceptors within a phase is not important. However, in certain situations it may be important to ensure that an interceptor is executed before, or after, other interceptors in the same phase.

4. [Section 58.2, "Processing messages"](#)

5. [Section 58.3, "Unwinding after an error"](#)

If an error occurs in the active interceptor chain after the interceptor has executed, its fault processing logic is invoked.

6. [Chapter 59, Configuring Endpoints to Use Interceptors](#)

CHAPTER 56. THE INTERCEPTOR APIs

Abstract

Interceptors implement the `PhaseInterceptor` interface which extends the base `Interceptor` interface. This interface defines a number of methods used by the Apache CXF's runtime to control interceptor execution and are not appropriate for application developers to implement. To simplify interceptor development, Apache CXF provides a number of abstract interceptor implementations that can be extended.

INTERFACES

All of the interceptors in Apache CXF implement the base `Interceptor` interface shown in [Example 56.1, "Base interceptor interface"](#).

Example 56.1. Base interceptor interface

```
package org.apache.cxf.interceptor;

public interface Interceptor<T extends Message>
{
    void handleMessage(T message) throws Fault;
    void handleFault(T message);
}
```

The `Interceptor` interface defines the two methods that a developer needs to implement for a custom interceptor:

`handleMessage()`

The `handleMessage()` method does most of the work in an interceptor. It is called on each interceptor in a message chain and receives the contents of the message being processed. Developers implement the message processing logic of the interceptor in this method. For detailed information about implementing the `handleMessage()` method, see [Section 58.2, "Processing messages"](#).

`handleFault()`

The `handleFault()` method is called on an interceptor when normal message processing has been interrupted. The runtime calls the `handleFault()` method of each invoked interceptor in reverse order as it unwinds an interceptor chain. For detailed information about implementing the `handleFault()` method, see [Section 58.3, "Unwinding after an error"](#).

Most interceptors do not directly implement the `Interceptor` interface. Instead, they implement the `PhaseInterceptor` interface shown in [Example 56.2, "The phase interceptor interface"](#). The `PhaseInterceptor` interface adds four methods that allow an interceptor the participate in interceptor chains.

Example 56.2. The phase interceptor interface

```
package org.apache.cxf.phase;
```

```
...
public interface PhaseInterceptor<T extends Message> extends Interceptor<T>
{
    Set<String> getAfter();

    Set<String> getBefore();

    String getId();

    String getPhase();

}
```

ABSTRACT INTERCEPTOR CLASS

Instead of directly implementing the `PhaseInterceptor` interface, developers should extend the **AbstractPhaseInterceptor** class. This abstract class provides implementations for the phase management methods of the `PhaseInterceptor` interface. The **AbstractPhaseInterceptor** class also provides a default implementation of the **handleFault()** method.

Developers need to provide an implementation of the **handleMessage()** method. They can also provide a different implementation for the **handleFault()** method. The developer-provided implementations can manipulate the message data using the methods provided by the generic `org.apache.cxf.message.Message` interface.

For applications that work with SOAP messages, Apache CXF provides an **AbstractSoapInterceptor** class. Extending this class provides the **handleMessage()** method and the **handleFault()** method with access to the message data as an `org.apache.cxf.binding.soap.SoapMessage` object. **SoapMessage** objects have methods for retrieving the SOAP headers, the SOAP envelope, and other SOAP metadata from the message.

CHAPTER 57. DETERMINING WHEN THE INTERCEPTOR IS INVOKED

Abstract

Interceptors are organized into phases. The phase in which an interceptor runs determines what portions of the message data it can access. An interceptor can determine its location in relationship to the other interceptors in the same phase. The interceptor's phase and its location within the phase are set as part of the interceptor's constructor logic.

57.1. SPECIFYING THE INTERCEPTOR LOCATION

When developing a custom interceptor, the first thing to consider is where in the message processing chain the interceptor belongs. The developer can control an interceptor's position in the message processing chain in one of two ways:

- Specifying the interceptor's phase
- Specifying constraints on the location of the interceptor within the phase

Typically, the code specifying an interceptor's location is placed in the interceptor's constructor. This makes it possible for the runtime to instantiate the interceptor and put in the proper place in the interceptor chain without any explicit action in the application level code.

57.2. SPECIFYING AN INTERCEPTOR'S PHASE

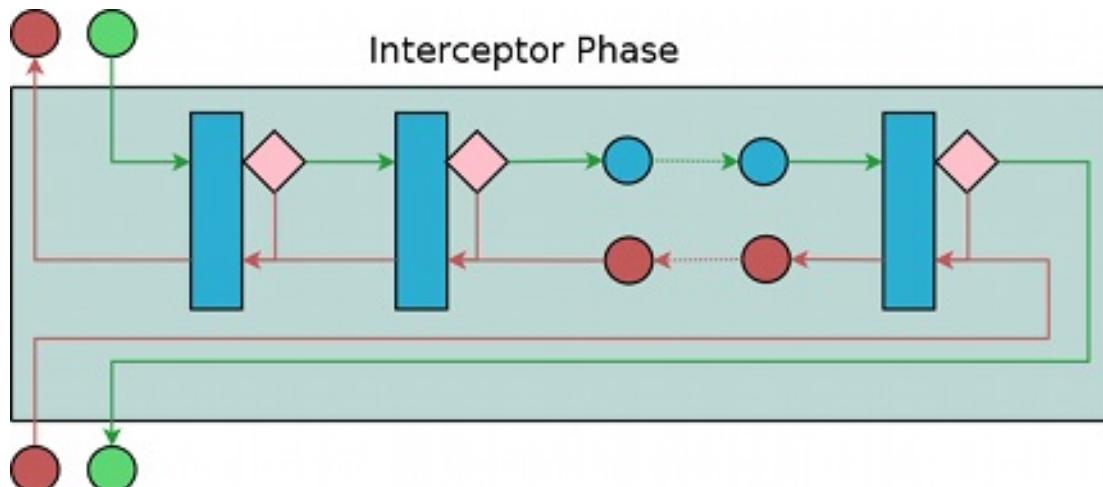
Overview

Interceptors are organized into phases. An interceptor's phase determines when in the message processing sequence it is called. Developers specify an interceptor's phase its constructor. Phases are specified using constant values provided by the framework.

Phase

Phases are a logical collection of interceptors. As shown in [Figure 57.1, "An interceptor phase"](#), the interceptors within a phase are called sequentially.

Figure 57.1. An interceptor phase



The phases are linked together in an ordered list to form an interceptor chain and provide defined logical steps in the message processing procedure. For example, a group of interceptors in the RECEIVE phase of an inbound interceptor chain processes transport level details using the raw message data picked up from the wire.

There is, however, no enforcement of what can be done in any of the phases. It is recommended that interceptors within a phase adhere to tasks that are in the spirit of the phase.

The complete list of phases defined by Apache CXF can be found in [Chapter 62, Apache CXF Message Processing Phases](#).

Specifying a phase

Apache CXF provides the **org.apache.cxf.Phase** class to use for specifying a phase. The class is a collection of constants. Each phase defined by Apache CXF has a corresponding constant in the **Phase** class. For example, the RECEIVE phase is specified by the value Phase.RECEIVE.

Setting the phase

An interceptor's phase is set in the interceptor's constructor. The **AbstractPhaseInterceptor** class defines three constructors for instantiating an interceptor:

- **public AbstractPhaseInterceptor(String phase)**—sets the phase of the interceptor to the specified phase and automatically sets the interceptor's id to the interceptor's class name. This constructor will satisfy most use cases.
- **public AbstractPhaseInterceptor(String id, String phase)**—sets the interceptor's id to the string passed in as the first parameter and the interceptor's phase to the second string.
- **public AbstractPhaseInterceptor(String phase, boolean uniqueld)**—specifies if the interceptor should use a unique, system generated id. If the **uniqueld** parameter is **true**, the interceptor's id will be calculated by the system. If the **uniqueld** parameter is **false** the interceptor's id is set to the interceptor's class name.

The recommended way to set a custom interceptor's phase is to pass the phase to the **AbstractPhaseInterceptor** constructor using the **super()** method as shown in [Example 57.1, "Setting an interceptor's phase"](#).

Example 57.1. Setting an interceptor's phase

```
import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;

public class StreamInterceptor extends AbstractPhaseInterceptor<Message>
{
    public StreamInterceptor()
    {
        super(Phase.PRE_STREAM);
    }
}
```

The **StreamInterceptor** interceptor shown in [Example 57.1, "Setting an interceptor's phase"](#) is placed into the PRE_STREAM phase.

57.3. CONSTRAINING AN INTERCEPTORS PLACEMENT IN A PHASE

Overview

Placing an interceptor into a phase may not provide fine enough control over its placement to ensure that the interceptor works properly. For example, if an interceptor needed to inspect the SOAP headers of a message using the SAAJ APIs, it would need to run after the interceptor that converts the message into a SAAJ object. There may also be cases where one interceptor consumes a part of the message needed by another interceptor. In these cases, a developer can supply a list of interceptors that must be executed before their interceptor. A developer can also supply a list of interceptors that must be executed after their interceptor.



IMPORTANT

The runtime can only honor these lists within the interceptor's phase. If a developer places an interceptor from an earlier phase in the list of interceptors that must execute after the current phase, the runtime will ignore the request.

Add to the chain before

One issue that arises when developing an interceptor is that the data required by the interceptor is not always present. This can occur when one interceptor in the chain consumes message data required by a later interceptor. Developers can control what a custom interceptor consumes and possibly fix the problem by modifying their interceptors. However, this is not always possible because a number of interceptors are used by Apache CXF and a developer cannot modify them.

An alternative solution is to ensure that a custom interceptor is placed before any interceptors that will consume the message data the custom interceptor requires. The easiest way to do that would be to place it in an earlier phase, but that is not always possible. For cases where an interceptor needs to be placed before one or more other interceptors the Apache CXF's **AbstractPhaseInterceptor** class provides two **addBefore()** methods.

As shown in [Example 57.2, "Methods for adding an interceptor before other interceptors"](#), one takes a single interceptor id and the other takes a collection of interceptor ids. You can make multiple calls to continue adding interceptors to the list.

Example 57.2. Methods for adding an interceptor before other interceptors

```
public addBeforeString()
public addBeforeCollection<String>i
```

As shown in [Example 57.3, "Specifying a list of interceptors that must run after the current interceptor"](#), a developer calls the **addBefore()** method in the constructor of a custom interceptor.

Example 57.3. Specifying a list of interceptors that must run after the current interceptor

```
public class MyPhasedOutInterceptor extends AbstractPhaseInterceptor<Message>
{
    public MyPhasedOutInterceptor() {
        super(Phase.PRE_LOGICAL);
```

```

        addBefore(HolderOutInterceptor.class.getName());
    }

    ...
}

```

Most interceptors use their class name for an interceptor id.

Add to the chain after

Another reason the data required by the interceptor is not present is that the data has not been placed in the message object. For example, an interceptor may want to work with the message data as a SOAP message, but it will not work if it is placed in the chain before the message is turned into a SOAP message. Developers can control what a custom interceptor consumes and possibly fix the problem by modifying their interceptors. However, this is not always possible because a number of interceptors are used by Apache CXF and a developer cannot modify them.

An alternative solution is to ensure that a custom interceptor is placed after the interceptor, or interceptors, that generate the message data the custom interceptor requires. The easiest way to do that would be to place it in a later phase, but that is not always possible. The **AbstractPhaseInterceptor** class provides two **addAfter()** methods for cases where an interceptor needs to be placed after one or more other interceptors.

As shown in [Example 57.4, “Methods for adding an interceptor after other interceptors”](#), one method takes a single interceptor id and the other takes a collection of interceptor ids. You can make multiple calls to continue adding interceptors to the list.

Example 57.4. Methods for adding an interceptor after other interceptors

```
public addAfterString() public addAfterCollection<String>i
```

As shown in [Example 57.5, “Specifying a list of interceptors that must run before the current interceptor”](#), a developer calls the **addAfter()** method in the constructor of a custom interceptor.

Example 57.5. Specifying a list of interceptors that must run before the current interceptor

```

public class MyPhasedOutInterceptor extends AbstractPhaseInterceptor<Message>
{
    public MyPhasedOutInterceptor() {
        super(Phase.PRE_LOGICAL);
        addAfter(StartingOutInterceptor.class.getName());
    }

    ...
}

```

Most interceptors use their class name for an interceptor id.

CHAPTER 58. IMPLEMENTING THE INTERCEPTORS PROCESSING LOGIC

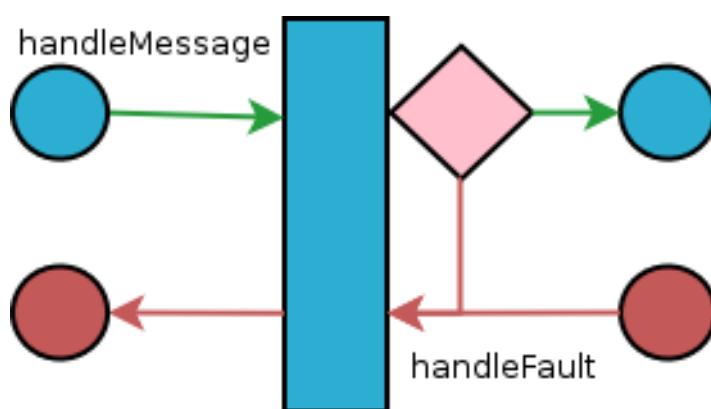
Abstract

Interceptors are straightforward to implement. The bulk of their processing logic is in the **handleMessage()** method. This method receives the message data and manipulates it as needed. Developers may also want to add some special logic to handle fault processing cases.

58.1. INTERCEPTOR FLOW

[Figure 58.1, "Flow through an interceptor"](#) shows the process flow through an interceptor.

Figure 58.1. Flow through an interceptor



In normal message processing, only the **handleMessage()** method is called. The **handleMessage()** method is where the interceptor's message processing logic is placed.

If an error occurs in the **handleMessage()** method of the interceptor, or any subsequent interceptor in the interceptor chain, the **handleFault()** method is called. The **handleFault()** method is useful for cleaning up after an interceptor in the event of an error. It can also be used to alter the fault message.

58.2. PROCESSING MESSAGES

Overview

In normal message processing, an interceptor's **handleMessage()** method is invoked. It receives that message data as a Message object. Along with the actual contents of the message, the Message object may contain a number of properties related to the message or the message processing state. The exact contents of the Message object depends on the interceptors preceding the current interceptor in the chain.

Getting the message contents

The Message interface provides two methods that can be used in extracting the message contents:

- `public<T> T getContent(java.lang.Class<T> format)` The **getContent()** method returns the content of the message in an object of the specified class. If the contents are not available as an instance of the specified class, null is returned. The list of available content types is determined by the interceptor's location on the interceptor chain and the direction of the interceptor chain.

- **publicCollection<Attachment>getAttachments** The **getAttachments()** method returns a Java **Collection** object containing any binary attachments associated with the message. The attachments are stored in org.apache.cxf.message.Attachment objects. Attachment objects provide methods for managing the binary data.



IMPORTANT

Attachments are only available after the attachment processing interceptors have executed.

Determining the message's direction

The direction of a message can be determined by querying the message exchange. The message exchange stores the inbound message and the outbound message in separate properties.^[3]

The message exchange associated with a message is retrieved using the message's **getExchange()** method. As shown in [Example 58.1, "Getting the message exchange"](#), **getExchange()** does not take any parameters and returns the message exchange as a org.apache.cxf.message.Exchange object.

Example 58.1. Getting the message exchange

ExchangegetExchange

The Exchange object has four methods, shown in [Example 58.2, "Getting messages from a message exchange"](#), for getting the messages associated with an exchange. Each method will either return the message as a org.apache.cxf.Message object or it will return null if the message does not exist.

Example 58.2. Getting messages from a message exchange

MessagegetInMessageMessagegetInFaultMessageMessagegetOutMessageMessagegetOutFaultMessage

[Example 58.3, "Checking the direction of a message chain"](#) shows code for determining if the current message is outbound. The method gets the message exchange and checks to see if the current message is the same as the exchange's outbound message. It also checks the current message against the exchanges outbound fault message to error messages on the outbound fault interceptor chain.

Example 58.3. Checking the direction of a message chain

```
public static boolean isOutbound()
{
    Exchange exchange = message.getExchange();
    return message != null
        && exchange != null
        && (message == exchange.getOutMessage()
            || message == exchange.getOutFaultMessage());
}
```

Example

[Example 58.4, “Example message processing method”](#) shows code for an interceptor that processes zip compressed messages. It checks the direction of the message and then performs the appropriate actions.

Example 58.4. Example message processing method

```
import java.io.IOException;
import java.io.InputStream;
import java.util.zip.GZIPInputStream;

import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;

public class StreamInterceptor extends AbstractPhaseInterceptor<Message>
{
    ...

    public void handleMessage(Message message)
    {

        boolean isOutbound = false;
        isOutbound = message == message.getExchange().getOutMessage()
            || message == message.getExchange().getOutFaultMessage();

        if (!isOutbound)
        {
            try
            {
                InputStream is = message.getContent(InputStream.class);
                GZIPInputStream zipInput = new GZIPInputStream(is);
                message.setContent(InputStream.class, zipInput);
            }
            catch (IOException ioe)
            {
                ioe.printStackTrace();
            }
        }
        else
        {
            // zip the outbound message
        }
    }
    ...
}
```

58.3. UNWINDING AFTER AN ERROR

Overview

When an error occurs during the execution of an interceptor chain, the runtime stops traversing the interceptor chain and unwinds the chain by calling the **handleFault()** method of any interceptors in the chain that have already been executed.

The **handleFault()** method can be used to clean up any resources used by an interceptor during normal message processing. It can also be used to rollback any actions that should only stand if message processing completes successfully. In cases where the fault message will be passed on to an outbound fault processing interceptor chain, the **handleFault()** method can also be used to add information to the fault message.

Getting the message payload

The **handleFault()** method receives the same Message object as the **handleMessage()** method used in normal message processing. Getting the message contents from the Message object is described in [the section called "Getting the message contents"](#).

Example

[Example 58.5, "Handling an unwinding interceptor chain"](#) shows code used to ensure that the original XML stream is placed back into the message when the interceptor chain is unwound.

Example 58.5. Handling an unwinding interceptor chain

```
@Override  
public void handleFault(SoapMessage message)  
{  
    super.handleFault(message);  
    XMLStreamWriter writer = (XMLStreamWriter)message.get(ORIGINAL_XML_WRITER);  
    if (writer != null)  
    {  
        message.setContent(XMLStreamWriter.class, writer);  
    }  
}
```

[3] It also stores inbound and outbound faults separately.

CHAPTER 59. CONFIGURING ENDPOINTS TO USE INTERCEPTORS

Abstract

Interceptors are added to an endpoint when it is included in a message exchange. The endpoint's interceptor chains are constructed from the interceptor chains of a number of components in the Apache CXF runtime. Interceptors are specified in either the endpoint's configuration or the configuration of one of the runtime components. Interceptors can be added using either the configuration file or the interceptor API.

59.1. DECIDING WHERE TO ATTACH INTERCEPTORS

Overview

There are a number of runtime objects that host interceptor chains. These include:

- the endpoint object
- the service object
- the proxy object
- the factory object used to create the endpoint or the proxy
- the binding
- the central **Bus** object

A developer can attach their own interceptors to any of these objects. The most common objects to attach interceptors are the bus and the individual endpoints. Choosing the correct object requires understanding how these runtime objects are combined to make an endpoint. As per the design, each cxf related bundle has its own cxf bus. Hence, if the interceptors are configured in the bus and the service at the same Blueprint context is imported or created into another bundle, the interceptor won't be processed. Instead you can configure the interceptors directly into the JAXWS client or endpoint in the imported service.

Endpoints and proxies

Attaching interceptors to either the endpoint or the proxy is the most fine grained way to place an interceptor. Any interceptors attached directly to an endpoint or a proxy only effect the specific endpoint or proxy. This is a good place to attach interceptors that are specific to a particular incarnation of a service. For example, if a developer wants to expose one instance of a service that converts units from metric to imperial they could attach the interceptors directly to one endpoint.

Factories

Using the Spring configuration to attach interceptors to the factories used to create an endpoint or a proxy has the same effect as attaching the interceptors directly to the endpoint or proxy. However, when interceptors are attached to a factory programmatically the interceptors attached to the factory are propagated to every endpoint or proxy created by the factory.

Bindings

Attaching interceptors to the binding allows the developer to specify a set of interceptors that are applied to all endpoints that use the binding. For example, if a developer wants to force all endpoints that use the raw XML binding to include a special ID element, they could attach the interceptor responsible for adding the element to the XML binding.

Buses

The most general place to attach interceptors is the bus. When interceptors are attached to the bus, the interceptors are propagated to all of the endpoints managed by that bus. Attaching interceptors to the bus is useful in applications that create multiple endpoints that share a similar set of interceptors.

Combining attachment points

Because an endpoint's final set of interceptor chains is an amalgamation of the interceptor chains contributed by the listed objects, several of the listed objects can be combined in a single endpoint's configuration. For example, if an application spawned multiple endpoints that all required an interceptor that checked for a validation token, that interceptor would be attached to the application's bus. If one of those endpoints also required an interceptor that converted Euros into dollars, the conversion interceptor would be attached directly to the specific endpoint.

59.2. ADDING INTERCEPTORS USING CONFIGURATION

Overview

The easiest way to attach interceptors to an endpoint is using the configuration file. Each interceptor to be attached to an endpoint is configured using a standard Spring bean. The interceptor's bean can then be added to the proper interceptor chain using Apache CXF configuration elements.

Each runtime component that has an associated interceptor chain is configurable using specialized Spring elements. Each of the component's elements have a standard set of children for specifying their interceptor chains. There is one child for each interceptor chain associated with the component. The children list the beans for the interceptors to be added to the chain.

Configuration elements

[Table 59.1, “Interceptor chain configuration elements”](#) describes the four configuration elements for attaching interceptors to a runtime component.

Table 59.1. Interceptor chain configuration elements

Element	Description
inInterceptors	Contains a list of beans configuring interceptors to add to an endpoint's inbound interceptor chain.
outInterceptors	Contains a list of beans configuring interceptors to add to an endpoint's outbound interceptor chain.
inFaultInterceptors	Contains a list of beans configuring interceptors to add to an endpoint's inbound fault processing interceptor chain.

Element	Description
outFaultInterceptors	Contains a list of beans configuring interceptors to add to an endpoint's outbound fault processing interceptor chain.

All of the interceptor chain configuration elements take a **list** child element. The **list** element has one child for each of the interceptors being attached to the chain. Interceptors can be specified using either a **bean** element directly configuring the interceptor or a **ref** element that refers to a **bean** element that configures the interceptor.

Examples

[Example 59.1, "Attaching interceptors to the bus"](#) shows configuration for attaching interceptors to a bus' inbound interceptor chain.

Example 59.1. Attaching interceptors to the bus

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cxf="http://cxf.apache.org/core"
       xmlns:http="http://cxf.apache.org/transports/http/configuration"
       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://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">
    ...
    <bean id="GZIPStream" class="demo.stream.interceptor.StreamInterceptor"/>

    <cxf:bus>
        *<cxf:inInterceptors>
            <list>
                <ref bean="GZIPStream"/>
            </list>
        </cxf:inInterceptors>*
    </cxf:bus>
</beans>
```

[Example 59.2, "Attaching interceptors to a JAX-WS service provider"](#) shows configuration for attaching an interceptor to a JAX-WS service's outbound interceptor chain.

Example 59.2. Attaching interceptors to a JAX-WS service provider

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
```

```

xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

<jaxws:endpoint ...>
*<jaxws:outInterceptors>
<list>
<bean id="GZIPStream" class="demo.stream.interceptor.StreamInterceptor" />
</list>
</jaxws:outInterceptors>*
</jaxws:endpoint>
</beans>

```

More information

For more information about configuring endpoints using the Spring configuration see [Part IV, "Configuring Web Service Endpoints"](#).

59.3. ADDING INTERCEPTORS PROGRAMMATICALLY

59.3.1. Approaches to Adding Interceptors

Interceptors can be attached to endpoints programmatically using either one of two approaches:

- the InterceptorProvider API
- Java annotations

Using the InterceptorProvider API allows the developer to attach interceptors to any of the runtime components that have interceptor chains, but it requires working with the underlying Apache CXF classes. The Java annotations can only be added to service interfaces or service implementations, but they allow developers to stay within the JAX-WS API or the JAX-RS API.

59.3.2. Using the interceptor provider API

Overview

Interceptors can be registered with any component that implements the InterceptorProvider interface shown in [The interceptor provider interface](#).

The interceptor provider interface

```

package org.apache.cxf.interceptor;

import java.util.List;

public interface InterceptorProvider
{
    List<Interceptor<? extends Message>> getInInterceptors();

    List<Interceptor<? extends Message>> getOutInterceptors();
}

```

```

    List<Interceptor<? extends Message>> getInFaultInterceptors();
    List<Interceptor<? extends Message>> getOutFaultInterceptors();
}

```

The four methods in the interface allow you to retrieve each of an endpoint's interceptor chains as a Java **List** object. Using the methods offered by the Java **List** object, developers can add and remove interceptors to any of the chains.

Procedure

To use the InterceptorProvider API to attach an interceptor to a runtime component's interceptor chain, you must:

1. Get access to the runtime component with the chain to which the interceptor is being attached. Developers must use Apache CXF specific APIs to access the runtime components from standard Java application code. The runtime components are usually accessible by casting the JAX-WS or JAX-RS artifacts into the underlying Apache CXF objects.
2. Create an instance of the interceptor.
3. Use the proper get method to retrieve the desired interceptor chain.
4. Use the **List** object's **add()** method to attach the interceptor to the interceptor chain. This step is usually combined with retrieving the interceptor chain.

Attaching an interceptor to a consumer

[Attaching an interceptor to a consumer programmatically](#) shows code for attaching an interceptor to the inbound interceptor chain of a JAX-WS consumer.

Attaching an interceptor to a consumer programmatically

```

package com.fusesource.demo;

import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import org.apache.cxf.endpoint.Client;

public class Client
{
    public static void main(String args[])
    {
        QName serviceName = new QName("http://demo.eric.org", "stockQuoteReporter");
        Service s = Service.create(serviceName);

        QName portName = new QName("http://demo.eric.org", "stockQuoteReporterPort");
        s.addPort(portName, "http://schemas.xmlsoap.org/soap/", "http://localhost:9000/EricStockQuote");

        quoteReporter proxy = s.getPort(portName, quoteReporter.class);

        Client cxfClient = (Client) proxy;
    }
}

```

```

ValidateInterceptor validInterceptor = new ValidateInterceptor();
cxfClient.getInInterceptor().add(validInterceptor);

...
}
}

```

The code in [Attaching an interceptor to a consumer programmatically](#) does the following:

- Creates a JAX-WS **Service** object for the consumer.
- Adds a port to the **Service** object that provides the consumer's target address.
- Creates the proxy used to invoke methods on the service provider.
- Casts the proxy to the **org.apache.cxf.endpoint.Client** type.
- Creates an instance of the interceptor.
- Attaches the interceptor to the inbound interceptor chain.

Attaching an interceptor to a service provider

[Attaching an interceptor to a service provider programmatically](#) shows code for attaching an interceptor to a service provider's outbound interceptor chain.

Attaching an interceptor to a service provider programmatically

```

package com.fusesource.demo;
import java.util.*;

import org.apache.cxf.endpoint.Server;
import org.apache.cxf.frontend.ServerFactoryBean;
import org.apache.cxf.frontend.EndpointImpl;

public class stockQuoteReporter implements quoteReporter
{
    ...
    public stockQuoteReporter()
    {
        ServerFactoryBean sfb = new ServerFactoryBean();
        Server server = sfb.create();
        EndpointImpl endpt = server.getEndpoint();

        AuthTokenInterceptor authInterceptor = new AuthTokenInterceptor();

        endpt.getOutInterceptor().add(authInterceptor);
    }
}

```

The code in [Attaching an interceptor to a service provider programmatically](#) does the following:

- Creates a **ServerFactoryBean** object that will provide access to the underlying Apache CXF objects.
- Gets the **Server** object that Apache CXF uses to represent the endpoint.

Gets the Apache CXF **EndpointImpl** object for the service provider.

Creates an instance of the interceptor.

Attaches the interceptor to the endpoint's outbound interceptor chain.

Attaching an interceptor to a bus

[Attaching an interceptor to a bus](#) shows code for attaching an interceptor to a bus' inbound interceptor chain.

Attaching an interceptor to a bus

```
import org.apache.cxf.BusFactory;
org.apache.cxf.Bus;

...
Bus bus = BusFactory.getDefaultBus();

WatchInterceptor watchInterceptor = new WatchInterceptor();

bus.getInInterceptor().add(watchInterceptor);

...
```

The code in [Attaching an interceptor to a bus](#) does the following:

Gets the default bus for the runtime instance.

Creates an instance of the interceptor.

Attaches the interceptor to the inbound interceptor chain.

The **WatchInterceptor** will be attached to the inbound interceptor chain of all endpoints created by the runtime instance.

59.3.3. Using Java annotations

Overview

Apache CXF provides four Java annotations that allow a developer to specify the interceptor chains used by an endpoint. Unlike the other means of attaching interceptors to endpoints, the annotations are attached to application-level artifacts. The artifact that is used determines the scope of the annotation's effect.

Where to place the annotations

The annotations can be placed on the following artifacts:

- the service endpoint interface(SEI) defining the endpoint
If the annotations are placed on an SEI, all of the service providers that implement the interface and all of the consumers that use the SEI to create proxies will be affected.
- a service implementation class

If the annotations are placed on an implementation class, all of the service providers using the implementation class will be affected.

The annotations

The annotations are all in the `org.apache.cxf.interceptor` package and are described in [Table 59.2, “Interceptor chain annotations”](#).

Table 59.2. Interceptor chain annotations

Annotation	Description
InInterceptors	Specifies the interceptors for the inbound interceptor chain.
OutInterceptors	Specifies the interceptors for the outbound interceptor chain.
InFaultInterceptors	Specifies the interceptors for the inbound fault interceptor chain.
OutFaultInterceptors	Specifies the interceptors for the outbound fault interceptor chain.

Listing the interceptors

The list of interceptors is specified as a list of fully qualified class names using the syntax shown in [Syntax for listing interceptors in a chain annotation](#) .

Syntax for listing interceptors in a chain annotation

```
interceptors={"interceptor1", "interceptor2", ..., "interceptorN"}
```

Example

[Attaching interceptors to a service implementation](#) shows annotations that attach two interceptors to the inbound interceptor chain of endpoints that use the logic provided by `SayHiImpl`.

Attaching interceptors to a service implementation

```
import org.apache.cxf.interceptor.InInterceptors;

@InInterceptors(interceptors={"com.sayhi.interceptors.FirstLast",
"com.sayhi.interceptors.LogName"})
public class SayHiImpl implements SayHi
{
    ...
}
```

CHAPTER 60. MANIPULATING INTERCEPTOR CHAINS ON THE FLY

Abstract

Interceptors can reconfigure an endpoint's interceptor chain as part of its message processing logic. It can add new interceptors, remove interceptors, reorder interceptors, and even suspend the interceptor chain. Any on-the-fly manipulation is invocation-specific, so the original chain is used each time an endpoint is involved in a message exchange.

OVERVIEW

Interceptor chains only live as long as the message exchange that sparked their creation. Each message contains a reference to the interceptor chain responsible for processing it. Developers can use this reference to alter the message's interceptor chain. Because the chain is per-exchange, any changes made to a message's interceptor chain will not effect other message exchanges.

CHAIN LIFE-CYCLE

Interceptor chains and the interceptors in the chain are instantiated on a per-invocation basis. When an endpoint is invoked to participate in a message exchange, the required interceptor chains are instantiated along with instances of its interceptors. When the message exchange that caused the creation of the interceptor chain is completed, the chain and its interceptor instances are destroyed.

This means that any changes you make to the interceptor chain or to the fields of an interceptor do not persist across message exchanges. So, if an interceptor places another interceptor in the active chain only the active chain is effected. Any future message exchanges will be created from a pristine state as determined by the endpoint's configuration. It also means that a developer cannot set flags in an interceptor that will alter future message processing.

If an interceptor needs to pass information along to future instances, it can set a property in the message context. The context does persist across message exchanges.

GETTING THE INTERCEPTOR CHAIN

The first step in changing a message's interceptor chain is getting the interceptor chain. This is done using the `Message.getInterceptorChain()` method shown in [Example 60.1, "Method for getting an interceptor chain"](#). The interceptor chain is returned as a `org.apache.cxf.interceptor.InterceptorChain` object.

Example 60.1. Method for getting an interceptor chain

`InterceptorChain getInterceptorChain`

ADDING INTERCEPTORS

The `InterceptorChain` object has two methods, shown in [Example 60.2, "Methods for adding interceptors to an interceptor chain"](#), for adding interceptors to an interceptor chain. One allows you to add a single interceptor and the other allows you to add multiple interceptors.

Example 60.2. Methods for adding interceptors to an interceptor chain

addInterceptor<? extends Message>iaddCollection<Interceptor<? extends Message>>i

[Example 60.3, “Adding an interceptor to an interceptor chain on-the-fly”](#) shows code for adding a single interceptor to a message’s interceptor chain.

Example 60.3. Adding an interceptor to an interceptor chain on-the-fly

```
void handleMessage(Message message)
{
    ...
    AddledInterceptor addled = new AddledInterceptor();
    InterceptorChain chain = message.getInterceptorChain();
    chain.add(addled);
    ...
}
```

The code in [Example 60.3, “Adding an interceptor to an interceptor chain on-the-fly”](#) does the following:

Instantiates a copy of the interceptor to be added to the chain.

**IMPORTANT**

The interceptor being added to the chain should be in either the same phase as the current interceptor or a latter phase than the current interceptor.

Gets the interceptor chain for the current message.

Adds the new interceptor to the chain.

REMOVING INTERCEPTORS

The InterceptorChain object has one method, shown in [Example 60.4, “Methods for removing interceptors from an interceptor chain”](#), for removing an interceptor from an interceptor chain.

Example 60.4. Methods for removing interceptors from an interceptor chain**removeInterceptor<? extends Message>i**

[Example 60.5, “Removing an interceptor from an interceptor chain on-the-fly”](#) shows code for removing an interceptor from a message’s interceptor chain.

Example 60.5. Removing an interceptor from an interceptor chain on-the-fly

```
void handleMessage(Message message)
{
    ...
    Iterator<Interceptor<? extends Message>> iterator =
        message.getInterceptorChain().iterator();
    Interceptor<?> removeInterceptor = null;
    for (; iterator.hasNext(); ) {
```

```
Interceptor<?> interceptor = iterator.next();
if (interceptor.getClass().getName().equals("InterceptorClassName")) {
    removeInterceptor = interceptor;
    break;
}

if (removeInterceptor != null) {
    log.debug("Removing interceptor {}",removeInterceptor.getClass().getName());
    message.getInterceptorChain().remove(removeInterceptor);
}
...
}
```

Where **InterceptorClassName** is the class name of the interceptor you want to remove from the chain.

CHAPTER 61. JAX-RS 2.0 FILTERS AND INTERCEPTORS

Abstract

JAX-RS 2.0 defines standard APIs and semantics for installing filters and interceptors in the processing pipeline for REST invocations. Filters and interceptors are typically used to provide such capabilities as logging, authentication, authorization, message compression, message encryption, and so on.

61.1. INTRODUCTION TO JAX-RS FILTERS AND INTERCEPTORS

Overview

This section provides an overview of the processing pipeline for JAX-RS filters and interceptors, highlighting the extension points where it is possible to install a filter chain or an interceptor chain.

Filters

A JAX-RS 2.0 *filter* is a type of plug-in that gives a developer access to all of the JAX-RS messages passing through a CXF client or server. A filter is suitable for processing the metadata associated with a message: HTTP headers, query parameters, media type, and other metadata. Filters have the capability to abort a message invocation (useful for security plug-ins, for example).

If you like, you can install multiple filters at each extension point, in which case the filters are executed in a chain (the order of execution is undefined, however, unless you specify a **priority** value for each installed filter).

Interceptors

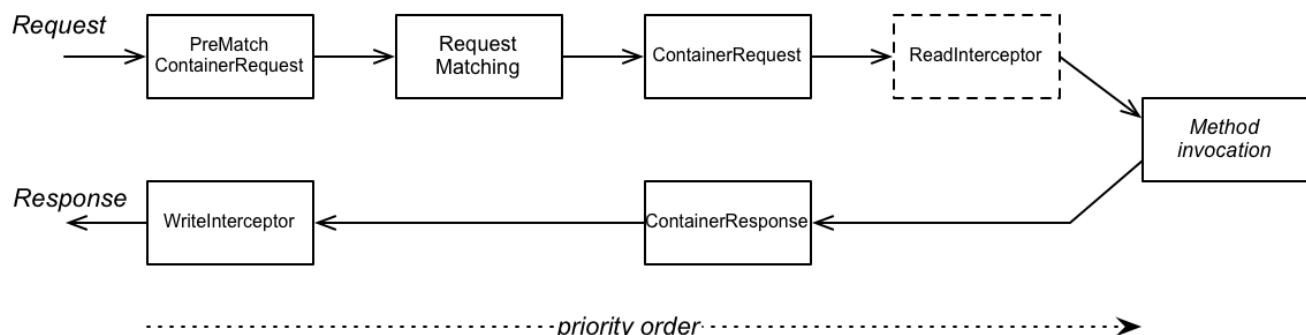
A JAX-RS 2.0 *interceptor* is a type of plug-in that gives a developer access to a message body as it is being read or written. Interceptors are wrapped around either the **MessageBodyReader.readFrom** method invocation (for reader interceptors) or the **MessageBodyWriter.writeTo** method invocation (for writer interceptors).

If you like, you can install multiple interceptors at each extension point, in which case the interceptors are executed in a chain (the order of execution is undefined, however, unless you specify a **priority** value for each installed interceptor).

Server processing pipeline

[Figure 61.1, "Server-Side Filter and Interceptor Extension Points"](#) shows an outline of the processing pipeline for JAX-RS filters and interceptors installed on the server side.

Figure 61.1. Server-Side Filter and Interceptor Extension Points



Server extension points

In the server processing pipeline, you can add a filter (or interceptor) at any of the following extension points:

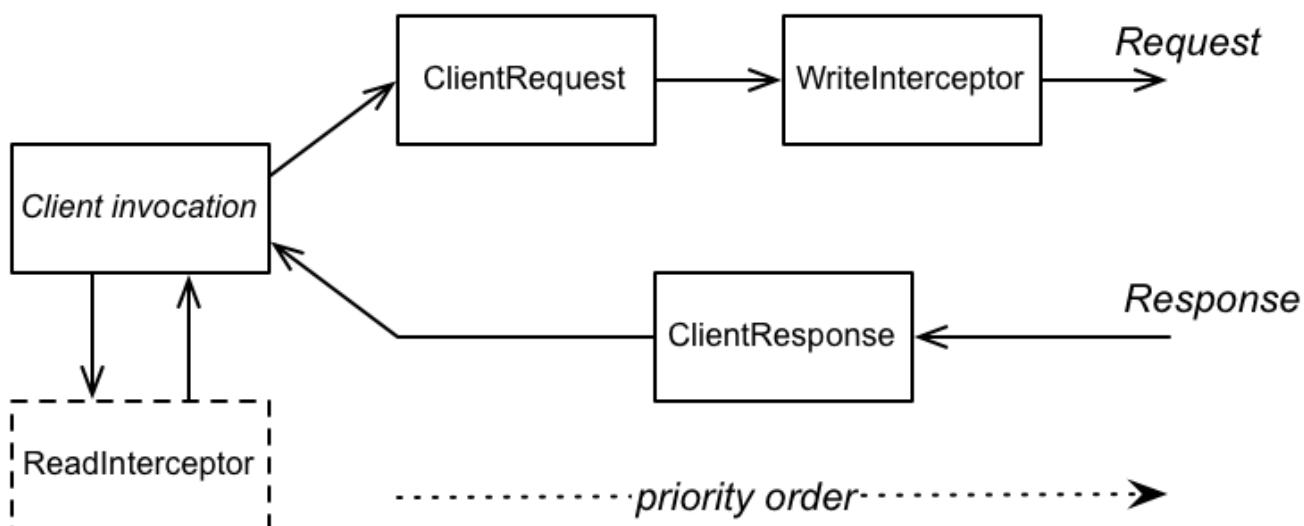
1. **PreMatchContainerRequest** filter
2. **ContainerRequest** filter
3. **ReadInterceptor**
4. **ContainerResponse** filter
5. **WriteInterceptor**

Note that the **PreMatchContainerRequest** extension point is reached **before** resource matching has occurred, so some of the context metadata will not be available at this point.

Client processing pipeline

[Figure 61.2, "Client-Side Filter and Interceptor Extension Points"](#) shows an outline of the processing pipeline for JAX-RS filters and interceptors installed on the client side.

Figure 61.2. Client-Side Filter and Interceptor Extension Points



Client extension points

In the client processing pipeline, you can add a filter (or interceptor) at any of the following extension points:

1. **ClientRequest** filter
2. **WriteInterceptor**
3. **ClientResponse** filter
4. **ReadInterceptor**

Filter and interceptor order

If you install multiple filters or interceptors at the same extension point, the execution order of the filters depends on the priority assigned to them (using the `@Priority` annotation in the Java source). A priority is represented as an integer value. In general, a filter with a **higher** priority number is placed closer to the resource method invocation on the server side; while a filter with a **lower** priority number is placed closer to the client invocation. In other words, the filters and interceptors acting on a **request** message are executed in **ascending order** of priority number; while the filters and interceptors acting on a **response** message are executed in **descending order** of priority number.

Filter classes

The following Java interfaces can be implemented in order to create custom REST message filters:

- [javax.ws.rs.container.ContainerRequestFilter](#)
- [javax.ws.rs.container.ContainerResponseFilter](#)
- [javax.ws.rs.client.ClientRequestFilter](#)
- [javax.ws.rs.client.ClientResponseFilter](#)

Interceptor classes

The following Java interfaces can be implemented in order to create custom REST message interceptors:

- [javax.ws.rs.ext.ReaderInterceptor](#)
- [javax.ws.rs.ext.WriterInterceptor](#)

61.2. CONTAINER REQUEST FILTER

Overview

This section explains how to implement and register a *container request filter*, which is used to intercept an incoming request message on the server (container) side. Container request filters are often used to process headers on the server side and can be used for any kind of generic request processing (that is, processing that is independent of the particular resource method called).

Moreover, the container request filter is something of a special case, because it can be installed at two distinct extension points: **PreMatchContainerRequest** (before the resource matching step); and **ContainerRequest** (after the resource matching step).

ContainerRequestFilter interface

The [javax.ws.rs.container.ContainerRequestFilter](#) interface is defined as follows:

```
// Java
...
package javax.ws.rs.container;
import java.io.IOException;
```

```
public interface ContainerRequestFilter {
    public void filter(ContainerRequestContext requestContext) throws IOException;
}
```

By implementing the **ContainerRequestFilter** interface, you can create a filter for either of the following extension points on the server side:

- **PreMatchContainerRequest**
- **ContainerRequest**

ContainerRequestContext interface

The **filter** method of **ContainerRequestFilter** receives a single argument of type [javax.ws.rs.container.ContainerRequestContext](#), which can be used to access the incoming request message and its related metadata. The **ContainerRequestContext** interface is defined as follows:

```
// Java
...
package javax.ws.rs.container;

import java.io.OutputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import java.net.URI;
import java.util.Date;
import java.util.Locale;
import java.util.Map;
import java.util.Set;

import javax.ws.rs.core.EntityTag;
import javax.ws.rs.core.Link;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.NewCookie;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.MessageBodyWriter;

public interface ContainerResponseContext {

    public int getStatus();

    public void setStatus(int code);

    public Response.StatusType getStatusInfo();

    public void setStatusInfo(Response.StatusType statusInfo);

    public MultivaluedMap<String, Object> getHeaders();

    public abstract MultivaluedMap<String, String> getStringHeaders();

    public String getHeaderString(String name);

    public Set<String> getAllowedMethods();
```

```
public Date getDate();

public Locale getLanguage();

public int getLength();

public MediaType getMediaType();

public Map<String, NewCookie> getCookies();

public EntityTag getEntityTag();

public Date getLastModified();

public URI getLocation();

public Set<Link> getLinks();

boolean hasLink(String relation);

public Link getLink(String relation);

public Link.Builder getLinkBuilder(String relation);

public boolean hasEntity();

public Object getEntity();

public Class<?> getEntityClass();

public Type getEntityType();

public void setEntity(final Object entity);

public void setEntity(
    final Object entity,
    final Annotation[] annotations,
    final MediaType mediaType);

public Annotation[] getEntityAnnotations();

public OutputStream getEntityStream();

public void setEntityStream(OutputStream outputStream);

}
```

Sample implementation for PreMatchContainerRequest filter

To implement a container request filter for the **PreMatchContainerRequest** extension point (that is, where the filter is executed prior to resource matching), define a class that implements the **ContainerRequestFilter** interface, making sure to annotate the class with the **@PreMatching** annotation (to select the **PreMatchContainerRequest** extension point).

For example, the following code shows an example of a simple container request filter that gets installed in the **PreMatchContainerRequest** extension point, with a priority of 20:

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.PreMatching;
import javax.annotation.Priority;
import javax.ws.rs.ext.Provider;

@PreMatching
@Priority(value = 20)
@Provider
public class SamplePreMatchContainerRequestFilter implements
    ContainerRequestFilter {

    public SamplePreMatchContainerRequestFilter() {
        System.out.println("SamplePreMatchContainerRequestFilter starting up");
    }

    @Override
    public void filter(ContainerRequestContext requestContext) {
        System.out.println("SamplePreMatchContainerRequestFilter.filter() invoked");
    }
}
```

Sample implementation for ContainerRequest filter

To implement a container request filter for the **ContainerRequest** extension point (that is, where the filter is executed **after** resource matching), define a class that implements the **ContainerRequestFilter** interface, **without** the **@PreMatching** annotation.

For example, the following code shows an example of a simple container request filter that gets installed in the **ContainerRequest** extension point, with a priority of 30:

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.ext.Provider;
import javax.annotation.Priority;

@Provider
@Priority(value = 30)
public class SampleContainerRequestFilter implements ContainerRequestFilter {

    public SampleContainerRequestFilter() {
        System.out.println("SampleContainerRequestFilter starting up");
    }

    @Override
    public void filter(ContainerRequestContext requestContext) {
        System.out.println("SampleContainerRequestFilter.filter() invoked");
    }
}
```

Injecting ResourcelInfo

At the **ContainerRequest** extension point (that is, **after** resource matching has occurred), it is possible to access the matched resource class and resource method by injecting the **ResourcelInfo** class. For example, the following code shows how to inject the **ResourcelInfo** class as a field of the **ContainerRequestFilter** class:

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.ResourcelInfo;
import javax.ws.rs.ext.Provider;
import javax.annotation.Priority;
import javax.ws.rs.core.Context;

@Provider
@Priority(value = 30)
public class SampleContainerRequestFilter implements ContainerRequestFilter {

    @Context
    private ResourcelInfo resinfo;

    public SampleContainerRequestFilter() {
        ...
    }

    @Override
    public void filter(ContainerRequestContext requestContext) {
        String resourceClass = resinfo.getResourceClass().getName();
        String methodName = resinfo.getResourceMethod().getName();
        System.out.println("REST invocation bound to resource class: " + resourceClass);
        System.out.println("REST invocation bound to resource method: " + methodName);
    }
}
```

Aborting the invocation

It is possible to abort a server-side invocation by creating a suitable implementation of a container request filter. Typically, this is useful for implementing security features on the server side: for example, to implement an authentication feature or an authorization feature. If an incoming request fails to authenticate successfully, you could abort the invocation from within the container request filter.

For example, the following pre-matching feature attempts to extract a username and password from the URI's query parameters and calls an authenticate method to check the username and password credentials. If the authentication fails, the invocation is aborted by calling **abortWith** on the **ContainerRequestContext** object, passing the error response that is to be returned to the client.

```
// Java
package org.jboss.fuse.example;

import javax.annotation.Priority;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
```

```

import javax.ws.rs.container.PreMatching;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.ResponseBuilder;
import javax.ws.rs.core.Response.Status;
import javax.ws.rs.ext.Provider;

@PreMatching
@Priority(value = 20)
@Provider
public class SampleAuthenticationRequestFilter implements
ContainerRequestFilter {

public SampleAuthenticationRequestFilter() {
    System.out.println("SampleAuthenticationRequestFilter starting up");
}

@Override
public void filter(ContainerRequestContext requestContext) {
    ResponseBuilder responseBuilder = null;
    Response response = null;

    String userName = requestContext.getUriInfo().getQueryParameters().getFirst("UserName");
    String password = requestContext.getUriInfo().getQueryParameters().getFirst("Password");
    if (authenticate(userName, password) == false) {
        responseBuilder = Response.serverError();
        response = responseBuilder.status(Status.BAD_REQUEST).build();
        requestContext.abortWith(response);
    }
}

public boolean authenticate(String userName, String password) {
    // Perform authentication of 'user'
    ...
}
}

```

Binding the server request filter

To **bind** a server request filter (that is, to install it into the Apache CXF runtime), perform the following steps:

1. Add the **@Provider** annotation to the container request filter class, as shown in the following code fragment:

```

// Java
package org.jboss.fuse.example;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.ext.Provider;
import javax.annotation.Priority;

@Provider
@Priority(value = 30)

```

```

public class SampleContainerRequestFilter implements ContainerRequestFilter {
    ...
}

```

When the container request filter implementation is loaded into the Apache CXF runtime, the REST implementation automatically scans the loaded classes to search for the classes marked with the **@Provider** annotation (the *scanning phase*).

2. When defining a JAX-RS server endpoint in XML (for example, see [Section 18.1, “Configuring JAX-RS Server Endpoints”](#)), add the server request filter to the list of providers in the **jaxrs:providers** element.

```

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
    xmlns:cxf="http://cxf.apache.org/blueprint/core"
    ...
>
...
<jaxrs:server id="customerService" address="/customers">
    ...
    <jaxrs:providers>
        <ref bean="filterProvider" />
    </jaxrs:providers>
    <bean id="filterProvider"
        class="org.jboss.fuse.example.SampleContainerRequestFilter"/>
</jaxrs:server>
</blueprint>

```



NOTE

This step is a non-standard requirement of Apache CXF. Strictly speaking, according to the JAX-RS standard, the **@Provider** annotation should be all that is required to bind the filter. But in practice, the standard approach is somewhat inflexible and can lead to clashing providers when many libraries are included in a large project.

61.3. CONTAINER RESPONSE FILTER

Overview

This section explains how to implement and register a *container response filter*, which is used to intercept an outgoing response message on the server side. Container response filters can be used to populate headers automatically in a response message and, in general, can be used for any kind of generic response processing.

ContainerResponseFilter interface

The **javax.ws.rs.container.ContainerResponseFilter** interface is defined as follows:

```
// Java
```

```

...
package javax.ws.rs.container;

import java.io.IOException;

public interface ContainerResponseFilter {
    public void filter(ContainerRequestContext requestContext, ContainerResponseContext
responseContext)
        throws IOException;
}

```

By implementing the **ContainerResponseFilter**, you can create a filter for the **ContainerResponse** extension point on the server side, which filters the response message **after** the invocation has executed.



NOTE

The container response filter gives you access both to the request message (through the **requestContext** argument) and the response message (through the **responseObject** message), but only the response can be modified at this stage.

ContainerResponseContext interface

The **filter** method of **ContainerResponseFilter** receives two arguments: an argument of type **javax.ws.rs.container.ContainerRequestContext** (see [the section called "ContainerRequestContext interface"](#)); and an argument of type **javax.ws.rs.container.ContainerResponseContext**, which can be used to access the outgoing response message and its related metadata.

The **ContainerResponseContext** interface is defined as follows:

```

// Java
...
package javax.ws.rs.container;

import java.io.OutputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import java.net.URI;
import java.util.Date;
import java.util.Locale;
import java.util.Map;
import java.util.Set;

import javax.ws.rs.core.EntityTag;
import javax.ws.rs.core.Link;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.NewCookie;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.MessageBodyWriter;

public interface ContainerResponseContext {
    public int getStatus();
}

```

```
public void setStatus(int code);

public Response.StatusType getStatusInfo();

public void setStatusInfo(Response.StatusType statusInfo);

public MultivaluedMap<String, Object> getHeaders();

public abstract MultivaluedMap<String, String> getStringHeaders();

public String getHeaderString(String name);

public Set<String> getAllowedMethods();

public Date getDate();

public Locale getLanguage();

public int getLength();

public MediaType getMediaType();

public Map<String, NewCookie> getCookies();

public EntityTag getEntityTag();

public Date getLastModified();

public URI getLocation();

public Set<Link> getLinks();

boolean hasLink(String relation);

public Link getLink(String relation);

public Link.Builder getLinkBuilder(String relation);

public boolean hasEntity();

public Object getEntity();

public Class<?> getEntityClass();

public Type getEntityType();

public void setEntity(final Object entity);

public void setEntity(
    final Object entity,
    final Annotation[] annotations,
    final MediaType mediaType);

public Annotation[] getEntityAnnotations();

public OutputStream getEntityStream();
```

```

    public void setEntityStream(OutputStream outputStream);
}

```

Sample implementation

To implement a container response filter for the **ContainerResponse** extension point (that is, where the filter is executed after the invocation has been executed on the server side), define a class that implements the **ContainerResponseFilter** interface.

For example, the following code shows an example of a simple container response filter that gets installed in the **ContainerResponse** extension point, with a priority of 10:

```

// Java
package org.jboss.fuse.example;

import javax.annotation.Priority;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.ext.Provider;

@Provider
@Priority(value = 10)
public class SampleContainerResponseFilter implements ContainerResponseFilter {

    public SampleContainerResponseFilter() {
        System.out.println("SampleContainerResponseFilter starting up");
    }

    @Override
    public void filter(
        ContainerRequestContext requestContext,
        ContainerResponseContext responseContext
    ) {
        // This filter replaces the response message body with a fixed string
        if (responseContext.hasEntity()) {
            responseContext.setEntity("New message body!");
        }
    }
}

```

Binding the server response filter

To **bind** a server response filter (that is, to install it into the Apache CXF runtime), perform the following steps:

1. Add the **@Provider** annotation to the container response filter class, as shown in the following code fragment:

```

// Java
package org.jboss.fuse.example;

```

```

import javax.annotation.Priority;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.ext.Provider;

@Provider
@Priority(value = 10)
public class SampleContainerResponseFilter implements ContainerResponseFilter {
    ...
}

```

When the container response filter implementation is loaded into the Apache CXF runtime, the REST implementation automatically scans the loaded classes to search for the classes marked with the **@Provider** annotation (the *scanning phase*).

2. When defining a JAX-RS server endpoint in XML (for example, see [Section 18.1, "Configuring JAX-RS Server Endpoints"](#)), add the server response filter to the list of providers in the **jaxrs:providers** element.

```

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
    xmlns:cxf="http://cxf.apache.org/blueprint/core"
    ...
>
...
<jaxrs:server id="customerService" address="/customers">
...
<jaxrs:providers>
    <ref bean="filterProvider" />
</jaxrs:providers>
<bean id="filterProvider"
    class="org.jboss.fuse.example.SampleContainerResponseFilter"/>
</jaxrs:server>
</blueprint>

```



NOTE

This step is a non-standard requirement of Apache CXF. Strictly speaking, according to the JAX-RS standard, the **@Provider** annotation should be all that is required to bind the filter. But in practice, the standard approach is somewhat inflexible and can lead to clashing providers when many libraries are included in a large project.

61.4. CLIENT REQUEST FILTER

Overview

This section explains how to implement and register a *client request filter*, which is used to intercept an outgoing request message on the client side. Client request filters are often used to process headers and can be used for any kind of generic request processing.

ClientRequestFilter interface

The **javax.ws.rs.client.ClientRequestFilter** interface is defined as follows:

```
// Java
package javax.ws.rs.client;

...
import javax.ws.rs.client.ClientRequestFilter;
import javax.ws.rs.client.ClientRequestContext;

...
public interface ClientRequestFilter {
    void filter(ClientRequestContext requestContext) throws IOException;
}
```

By implementing the **ClientRequestFilter**, you can create a filter for the **ClientRequest** extension point on the client side, which filters the request message before sending the message to the server.

ClientRequestContext interface

The **filter** method of **ClientRequestFilter** receives a single argument of type **javax.ws.rs.client.ClientRequestContext**, which can be used to access the outgoing request message and its related metadata. The **ClientRequestContext** interface is defined as follows:

```
// Java
...
package javax.ws.rs.client;

import java.io.OutputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import java.net.URI;
import java.util.Collection;
import java.util.Date;
import java.util.List;
import java.util.Locale;
import java.util.Map;

import javax.ws.rs.core.Configuration;
import javax.ws.rs.core.Cookie;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.MessageBodyWriter;

public interface ClientRequestContext {

    public Object getProperty(String name);

    public Collection<String> getPropertyNames();

    public void setProperty(String name, Object object);

    public void removeProperty(String name);

    public URI getUri();
```

```
public void setUri(URI uri);

public String getMethod();

public void setMethod(String method);

public MultivaluedMap<String, Object> getHeaders();

public abstract MultivaluedMap<String, String> getStringHeaders();

public String getHeaderString(String name);

public Date getDate();

public Locale getLanguage();

public MediaType getMediaType();

public List<MediaType> getAcceptableMediaTypes();

public List<Locale> getAcceptableLanguages();

public Map<String, Cookie> getCookies();

public boolean hasEntity();

public Object getEntity();

public Class<?> getEntityClass();

public Type getEntityType();

public void setEntity(final Object entity);

public void setEntity(
    final Object entity,
    final Annotation[] annotations,
    final MediaType mediaType);

public Annotation[] getEntityAnnotations();

public OutputStream getEntityStream();

public void setEntityStream(OutputStream outputStream);

public Client getClient();

public Configuration getConfiguration();

public void abortWith(Response response);
}
```

Sample implementation

To implement a client request filter for the **ClientRequest** extension point (that is, where the filter is executed prior to sending the request message), define a class that implements the **ClientRequestFilter** interface.

For example, the following code shows an example of a simple client request filter that gets installed in the **ClientRequest** extension point, with a priority of 20:

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.client.ClientRequestContext;
import javax.ws.rs.client.ClientRequestFilter;
import javax.annotation.Priority;

@Priority(value = 20)
public class SampleClientRequestFilter implements ClientRequestFilter {

    public SampleClientRequestFilter() {
        System.out.println("SampleClientRequestFilter starting up");
    }

    @Override
    public void filter(ClientRequestContext requestContext) {
        System.out.println("ClientRequestFilter.filter() invoked");
    }
}
```

Aborting the invocation

It is possible to abort a client-side invocation by implementing a suitable client request filter. For example, you might implement a client-side filter to check whether a request is correctly formatted and, if necessary, abort the request.

The following test code **always** aborts the request, returning the **BAD_REQUEST** HTTP status to the client calling code:

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.client.ClientRequestContext;
import javax.ws.rs.client.ClientRequestFilter;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;
import javax.annotation.Priority;

@Priority(value = 10)
public class TestAbortClientRequestFilter implements ClientRequestFilter {

    public TestAbortClientRequestFilter() {
        System.out.println("TestAbortClientRequestFilter starting up");
    }

    @Override
    public void filter(ClientRequestContext requestContext) {
        // Test filter: aborts with BAD_REQUEST status
    }
}
```

```

    requestContext.abortWith(Response.status(Status.BAD_REQUEST).build());
}
}

```

Registering the client request filter

Using the JAX-RS 2.0 client API, you can register a client request filter directly on a **javax.ws.rs.client.Client** object or on a **javax.ws.rs.client.WebTarget** object. Effectively, this means that the client request filter can optionally be applied to different scopes, so that only certain URI paths are affected by the filter.

For example, the following code shows how to register the **SampleClientRequestFilter** filter so that it applies to all invocations made using the **client** object; and how to register the **TestAbortClientRequestFilter** filter, so that it applies only to sub-paths of `rest/TestAbortClientRequest`.

```

// Java
...
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
client.register(new SampleClientRequestFilter());
WebTarget target = client
    .target("http://localhost:8001/rest/TestAbortClientRequest");
target.register(new TestAbortClientRequestFilter());

```

61.5. CLIENT RESPONSE FILTER

Overview

This section explains how to implement and register a *client response filter*, which is used to intercept an incoming response message on the client side. Client response filters can be used for any kind of generic response processing on the client side.

ClientResponseFilter interface

The **javax.ws.rs.client.ClientResponseFilter** interface is defined as follows:

```

// Java
package javax.ws.rs.client;
...
import java.io.IOException;

public interface ClientResponseFilter {
    void filter(ClientRequestContext requestContext, ClientResponseContext responseContext)
        throws IOException;
}

```

By implementing the **ClientResponseFilter**, you can create a filter for the **ClientResponse** extension point on the client side, which filters the response message after it is received from the server.

ClientResponseContext interface

The **filter** method of **ClientResponseFilter** receives two arguments: an argument of type **javax.ws.rs.client.ClientRequestContext** (see [the section called “ClientRequestContext interface”](#)); and an argument of type **javax.ws.rs.client.ClientResponseContext**, which can be used to access the outgoing response message and its related metadata.

The **ClientResponseContext** interface is defined as follows:

```
// Java
...
package javax.ws.rs.client;

import java.io.InputStream;
import java.net.URI;
import java.util.Date;
import java.util.Locale;
import java.util.Map;
import java.util.Set;

import javax.ws.rs.core.EntityTag;
import javax.ws.rs.core.Link;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.NewCookie;
import javax.ws.rs.core.Response;

public interface ClientResponseContext {
```

 public int getStatus();

 public void setStatus(int code);

 public Response.StatusType getStatusInfo();

 public void setStatusInfo(Response.StatusType statusInfo);

 public MultivaluedMap<String, String> getHeaders();

 public String getHeaderString(String name);

 public Set<String> getAllowedMethods();

 public Date getDate();

 public Locale getLanguage();

 public int getLength();

 public MediaType getMediaType();

 public Map<String, NewCookie> getCookies();

```

public EntityTag getEntityTag();

public Date getLastModified();

public URI getLocation();

public Set<Link> getLinks();

boolean hasLink(String relation);

public Link getLink(String relation);

public Link.Builder getLinkBuilder(String relation);

public boolean hasEntity();

public InputStream getEntityStream();

public void setEntityStream(InputStream input);
}

```

Sample implementation

To implement a client response filter for the **ClientResponse** extension point (that is, where the filter is executed after receiving a response message from the server), define a class that implements the **ClientResponseFilter** interface.

For example, the following code shows an example of a simple client response filter that gets installed in the **ClientResponse** extension point, with a priority of 20:

```

// Java
package org.jboss.fuse.example;

import javax.ws.rs.client.ClientRequestContext;
import javax.ws.rs.client.ClientResponseContext;
import javax.ws.rs.client.ClientResponseFilter;
import javax.annotation.Priority;

@Priority(value = 20)
public class SampleClientResponseFilter implements ClientResponseFilter {

    public SampleClientResponseFilter() {
        System.out.println("SampleClientResponseFilter starting up");
    }

    @Override
    public void filter(
        ClientRequestContext requestContext,
        ClientResponseContext responseContext
    ) {
        // Add an extra header on the response
        responseContext.getHeaders().putSingle("MyCustomHeader", "my custom data");
    }
}

```

Registering the client response filter

Using the JAX-RS 2.0 client API, you can register a client response filter directly on a **javax.ws.rs.client.Client** object or on a **javax.ws.rs.client.WebTarget** object. Effectively, this means that the client request filter can optionally be applied to different scopes, so that only certain URI paths are affected by the filter.

For example, the following code shows how to register the **SampleClientResponseFilter** filter so that it applies to all invocations made using the **client** object:

```
// Java
...
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
client.register(new SampleClientResponseFilter());
```

61.6. ENTITY READER INTERCEPTOR

Overview

This section explains how to implement and register an *entity reader interceptor*, which enables you to intercept the input stream when reading a message body either on the client side or on the server side. This is typically useful for generic transformations of the request body, such as encryption and decryption, or compressing and decompressing.

ReaderInterceptor interface

The **javax.ws.rs.ext.ReaderInterceptor** interface is defined as follows:

```
// Java
...
package javax.ws.rs.ext;

public interface ReaderInterceptor {
    public Object aroundReadFrom(ReaderInterceptorContext context)
        throws java.io.IOException, javax.ws.rs.WebApplicationException;
}
```

By implementing the **ReaderInterceptor** interface, you can intercept the message body (**Entity** object) as it is being read either on the server side or the client side. You can use an entity reader interceptor in either of the following contexts:

- **Server side**—if bound as a server-side interceptor, the entity reader interceptor intercepts the request message body when it is accessed by the application code (in the matched resource). Depending on the semantics of the REST request, the message body might not be accessed by the matched resource, in which case the reader interceptor is not called.
- **Client side**—if bound as a client-side interceptor, the entity reader interceptor intercepts the response message body when it is accessed by the client code. If the client code does not

explicitly access the response message (for example, by calling the **Response.getEntity** method), the reader interceptor is not called.

ReaderInterceptorContext interface

The **aroundReadFrom** method of **ReaderInterceptor** receives one argument of type **javax.ws.rs.ext.ReaderInterceptorContext**, which can be used to access both the message body (**Entity** object) and message metadata.

The **ReaderInterceptorContext** interface is defined as follows:

```
// Java
...
package javax.ws.rs.ext;

import java.io.IOException;
import java.io.InputStream;

import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MultivaluedMap;

public interface ReaderInterceptorContext extends InterceptorContext {

    public Object proceed() throws IOException, WebApplicationException;

    public InputStream getInputStream();

    public void setInputStream(InputStream is);

    public MultivaluedMap<String, String> getHeaders();
}
```

InterceptorContext interface

The **ReaderInterceptorContext** interface also supports the methods inherited from the base **InterceptorContext** interface.

The **InterceptorContext** interface is defined as follows:

```
// Java
...
package javax.ws.rs.ext;

import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import java.util.Collection;

import javax.ws.rs.core.MediaType;

public interface InterceptorContext {

    public Object getProperty(String name);

    public Collection<String> getPropertyNames();
```

```

public void setProperty(String name, Object object);

public void removeProperty(String name);

public Annotation[] getAnnotations();

public void setAnnotations(Annotation[] annotations);

Class<?> getType();

public void setType(Class<?> type);

Type getGenericType();

public void setGenericType(Type genericType);

public MediaType getMediaType();

public void setMediaType(MediaType mediaType);
}

```

Sample implementation on the client side

To implement an entity reader interceptor for the client side, define a class that implements the **ReaderInterceptor** interface.

For example, the following code shows an example of an entity reader interceptor for the client side (with a priority of 10), which replaces all instances of **COMPANY_NAME** by **Red Hat** in the message body of the incoming response:

```

// Java
package org.jboss.fuse.example;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;

import javax.annotation.Priority;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.ReaderInterceptor;
import javax.ws.rs.ext.ReaderInterceptorContext;

@Priority(value = 10)
public class SampleClientReaderInterceptor implements ReaderInterceptor {

    @Override
    public Object aroundReadFrom(ReaderInterceptorContext interceptorContext)
            throws IOException, WebApplicationException
    {
        InputStream inputStream = interceptorContext.getInputStream();
        byte[] bytes = new byte[inputStream.available()];
        inputStream.read(bytes);
        String responseContent = new String(bytes);
        responseContent = responseContent.replaceAll("COMPANY_NAME", "Red Hat");
        interceptorContext.setInputStream(new ByteArrayInputStream(responseContent.getBytes()));
    }
}

```

```

        return interceptorContext.proceed();
    }
}

```

Sample implementation on the server side

To implement an entity reader interceptor for the server side, define a class that implements the **ReaderInterceptor** interface and annotate it with the **@Provider** annotation.

For example, the following code shows an example of an entity reader interceptor for the server side (with a priority of 10), which replaces all instances of **COMPANY_NAME** by **Red Hat** in the message body of the incoming request:

```

// Java
package org.jboss.fuse.example;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;

import javax.annotation.Priority;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.Provider;
import javax.ws.rs.ext.ReaderInterceptor;
import javax.ws.rs.ext.ReaderInterceptorContext;

@Priority(value = 10)
@Provider
public class SampleServerReaderInterceptor implements ReaderInterceptor {

    @Override
    public Object aroundReadFrom(ReaderInterceptorContext interceptorContext)
            throws IOException, WebApplicationException {
        InputStream inputStream = interceptorContext.getInputStream();
        byte[] bytes = new byte[inputStream.available()];
        inputStream.read(bytes);
        String requestContent = new String(bytes);
        requestContent = requestContent.replaceAll("COMPANY_NAME", "Red Hat");
        interceptorContext.setInputStream(new ByteArrayInputStream(requestContent.getBytes()));

        return interceptorContext.proceed();
    }
}

```

Binding a reader interceptor on the client side

Using the JAX-RS 2.0 client API, you can register an entity reader interceptor directly on a **javax.ws.rs.client.Client** object or on a **javax.ws.rs.client.WebTarget** object. Effectively, this means that the reader interceptor can optionally be applied to different scopes, so that only certain URL paths are affected by the interceptor.

For example, the following code shows how to register the **SampleClientReaderInterceptor** interceptor so that it applies to all invocations made using the **client** object:

```
// Java
...
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
client.register(SampleClientReaderInterceptor.class);
```

For more details about registering interceptors with a JAX-RS 2.0 client, see [Section 49.5, “Configuring the Client Endpoint”](#).

Binding a reader interceptor on the server side

To **bind** a reader interceptor on the server side (that is, to install it into the Apache CXF runtime), perform the following steps:

1. Add the **@Provider** annotation to the reader interceptor class, as shown in the following code fragment:

```
// Java
package org.jboss.fuse.example;
...
import javax.annotation.Priority;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.Provider;
import javax.ws.rs.ext.ReaderInterceptor;
import javax.ws.rs.ext.ReaderInterceptorContext;

@Priority(value = 10)
@Provider
public class SampleServerReaderInterceptor implements ReaderInterceptor {
...
}
```

When the reader interceptor implementation is loaded into the Apache CXF runtime, the REST implementation automatically scans the loaded classes to search for the classes marked with the **@Provider** annotation (the *scanning phase*).

2. When defining a JAX-RS server endpoint in XML (for example, see [Section 18.1, “Configuring JAX-RS Server Endpoints”](#)), add the reader interceptor to the list of providers in the **jaxrs:providers** element.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
  xmlns:cxf="http://cxf.apache.org/blueprint/core"
  ...
>
...
<jaxrs:server id="customerService" address="/customers">
...
<jaxrs:providers>
```

```

<ref bean="interceptorProvider" />
</jaxrs:providers>
<bean id="interceptorProvider"
class="org.jboss.fuse.example.SampleServerReaderInterceptor"/>

</jaxrs:server>

</blueprint>

```



NOTE

This step is a non-standard requirement of Apache CXF. Strictly speaking, according to the JAX-RS standard, the **@Provider** annotation should be all that is required to bind the interceptor. But in practice, the standard approach is somewhat inflexible and can lead to clashing providers when many libraries are included in a large project.

61.7. ENTITY WRITER INTERCEPTOR

Overview

This section explains how to implement and register an *entity writer interceptor*, which enables you to intercept the output stream when writing a message body either on the client side or on the server side. This is typically useful for generic transformations of the request body, such as encryption and decryption, or compressing and decompressing.

WriterInterceptor interface

The **javax.ws.rs.ext.WriterInterceptor** interface is defined as follows:

```

// Java
...
package javax.ws.rs.ext;

public interface WriterInterceptor {
    void aroundWriteTo(WriterInterceptorContext context)
        throws java.io.IOException, javax.ws.rs.WebApplicationException;
}

```

By implementing the **WriterInterceptor** interface, you can intercept the message body (**Entity** object) as it is being written either on the server side or the client side. You can use an entity writer interceptor in either of the following contexts:

- **Server side**—if bound as a server-side interceptor, the entity writer interceptor intercepts the response message body just before it is marshalled and sent back to the client.
- **Client side**—if bound as a client-side interceptor, the entity writer interceptor intercepts the request message body just before it is marshalled and sent out to the server.

WriterInterceptorContext interface

The **aroundWriteTo** method of **WriterInterceptor** receives one argument of type **javax.ws.rs.ext.WriterInterceptorContext**, which can be used to access both the message body (**Entity** object) and message metadata.

The **WriterInterceptorContext** interface is defined as follows:

```
// Java
...
package javax.ws.rs.ext;

import java.io.IOException;
import java.io.OutputStream;

import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MultivaluedMap;

public interface WriterInterceptorContext extends InterceptorContext {

    void proceed() throws IOException, WebApplicationException;

    Object getEntity();

    void setEntity(Object entity);

    OutputStream getOutputStream();

    public void setOutputStream(OutputStream os);

    MultivaluedMap<String, Object> getHeaders();
}
```

InterceptorContext interface

The **WriterInterceptorContext** interface also supports the methods inherited from the base **InterceptorContext** interface. For the definition of **InterceptorContext**, see [the section called “InterceptorContext interface”](#).

Sample implementation on the client side

To implement an entity writer interceptor for the client side, define a class that implements the **WriterInterceptor** interface.

For example, the following code shows an example of an entity writer interceptor for the client side (with a priority of 10), which appends an extra line of text to the message body of the outgoing request:

```
// Java
package org.jboss.fuse.example;

import java.io.IOException;
import java.io.OutputStream;

import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.WriterInterceptor;
import javax.ws.rs.ext.WriterInterceptorContext;
import javax.annotation.Priority;
```

```

@Priority(value = 10)
public class SampleClientWriterInterceptor implements WriterInterceptor {

    @Override
    public void aroundWriteTo(WriterInterceptorContext interceptorContext)
        throws IOException, WebApplicationException {
        OutputStream outputStream = interceptorContext.getOutputStream();
        String appendedContent = "\nInterceptors always get the last word in.";
        outputStream.write(appendedContent.getBytes());
        interceptorContext.setOutputStream(outputStream);

        interceptorContext.proceed();
    }
}

```

Sample implementation on the server side

To implement an entity writer interceptor for the server side, define a class that implements the **WriterInterceptor** interface and annotate it with the **@Provider** annotation.

For example, the following code shows an example of an entity writer interceptor for the server side (with a priority of 10), which appends an extra line of text to the message body of the outgoing request:

```

// Java
package org.jboss.fuse.example;

import java.io.IOException;
import java.io.OutputStream;

import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.Provider;
import javax.ws.rs.ext.WriterInterceptor;
import javax.ws.rs.ext.WriterInterceptorContext;
import javax.annotation.Priority;

@Priority(value = 10)
@Provider
public class SampleServerWriterInterceptor implements WriterInterceptor {

    @Override
    public void aroundWriteTo(WriterInterceptorContext interceptorContext)
        throws IOException, WebApplicationException {
        OutputStream outputStream = interceptorContext.getOutputStream();
        String appendedContent = "\nInterceptors always get the last word in.";
        outputStream.write(appendedContent.getBytes());
        interceptorContext.setOutputStream(outputStream);

        interceptorContext.proceed();
    }
}

```

Binding a writer interceptor on the client side

Using the JAX-RS 2.0 client API, you can register an entity writer interceptor directly on a **javax.ws.rs.client.Client** object or on a **javax.ws.rs.client.WebTarget** object. Effectively, this means that the writer interceptor can optionally be applied to different scopes, so that only certain URI paths are affected by the interceptor.

For example, the following code shows how to register the **SampleClientReaderInterceptor** interceptor so that it applies to all invocations made using the **client** object:

```
// Java
...
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
client.register(SampleClientReaderInterceptor.class);
```

For more details about registering interceptors with a JAX-RS 2.0 client, see [Section 49.5, “Configuring the Client Endpoint”](#).

Binding a writer interceptor on the server side

To **bind** a writer interceptor on the server side (that is, to install it into the Apache CXF runtime), perform the following steps:

1. Add the **@Provider** annotation to the writer interceptor class, as shown in the following code fragment:

```
// Java
package org.jboss.fuse.example;
...
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.Provider;
import javax.ws.rs.ext.WriterInterceptor;
import javax.ws.rs.ext.WriterInterceptorContext;
import javax.annotation.Priority;

@Priority(value = 10)
@Provider
public class SampleServerWriterInterceptor implements WriterInterceptor {
    ...
}
```

When the writer interceptor implementation is loaded into the Apache CXF runtime, the REST implementation automatically scans the loaded classes to search for the classes marked with the **@Provider** annotation (the *scanning phase*).

2. When defining a JAX-RS server endpoint in XML (for example, see [Section 18.1, “Configuring JAX-RS Server Endpoints”](#)), add the writer interceptor to the list of providers in the **jaxrs:providers** element.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
xmlns:cxf="http://cxf.apache.org/blueprint/core"
...
>
...
<jaxrs:server id="customerService" address="/customers">
...
<jaxrs:providers>
<ref bean="interceptorProvider" />
</jaxrs:providers>
<bean id="interceptorProvider"
class="org.jboss.fuse.example.SampleServerWriterInterceptor"/>

</jaxrs:server>

</blueprint>

```



NOTE

This step is a non-standard requirement of Apache CXF. Strictly speaking, according to the JAX-RS standard, the **@Provider** annotation should be all that is required to bind the interceptor. But in practice, the standard approach is somewhat inflexible and can lead to clashing providers when many libraries are included in a large project.

61.8. DYNAMIC BINDING

Overview

The standard approach to binding container filters and container interceptors to resources is to annotate the filters and interceptors with the **@Provider** annotation. This ensures that the binding is **global**: that is, the filters and interceptors are bound to **every** resource class and resource method on the server side.

Dynamic binding is an alternative approach to binding on the server side, which enables you to pick and choose which resource methods your interceptors and filters are applied to. To enable dynamic binding for your filters and interceptors, you must implement a custom **DynamicFeature** interface, as described here.

DynamicFeature interface

The **DynamicFeature** interface is defined in the **javax.ws.rs.container** package, as follows:

```

// Java
package javax.ws.rs.container;

import javax.ws.rs.core.FeatureContext;
import javax.ws.rs.ext.ReaderInterceptor;
import javax.ws.rs.ext.WriterInterceptor;

public interface DynamicFeature {
    public void configure(ResourceInfo resourceInfo, FeatureContext context);
}

```

Implementing a dynamic feature

You implement a dynamic feature, as follows:

1. Implement one or more container filters or container interceptors, as described previously. But do **not** annotate them with the **@Provider** annotation (otherwise, they would be bound globally, making the dynamic feature effectively irrelevant).
2. Create your own dynamic feature by implementing the **DynamicFeature** class, overriding the **configure** method.
3. In the **configure** method, you can use the **ResourceInfo** argument to discover which resource class and which resource method this feature is being called for. You can use this information as the basis for deciding whether or not to register some of the filters or interceptors.
4. If you decide to register a filter or an interceptor with the current resource method, you can do so by invoking one of the **context.register** methods.
5. Remember to annotate your dynamic feature class with the **@Provider** annotation, to ensure that it gets picked up during the scanning phase of deployment.

Example dynamic feature

The following example shows you how to define a dynamic feature that registers the **LoggingFilter** filter for any method of the **MyResource** class (or subclass) that is annotated with **@GET**:

```
// Java
...
import javax.ws.rs.container.DynamicFeature;
import javax.ws.rs.container.ResourceInfo;
import javax.ws.rs.core.FeatureContext;
import javax.ws.rs.ext.Provider;

@Provider
public class DynamicLoggingFilterFeature implements DynamicFeature {
    @Override
    void configure(ResourceInfo resourceInfo, FeatureContext context) {
        if (MyResource.class.isAssignableFrom(resourceInfo.getResourceClass())
            && resourceInfo.getResourceMethod().isAnnotationPresent(GET.class)) {
            context.register(new LoggingFilter());
        }
    }
}
```

Dynamic binding process

The JAX-RS standard requires that the **DynamicFeature.configure** method is called exactly once for each resource method. This means that every resource method could potentially have filters or interceptors installed by the dynamic feature, but it is up to the dynamic feature to decide whether to register the filters or interceptors in each case. In other words, the granularity of binding supported by the dynamic feature is at the level of individual resource methods.

FeatureContext interface

The **FeatureContext** interface (which enables you to register filters and interceptors in the **configure** method) is defined as a sub-interface of **Configurable<>**, as follows:

```
// Java
package javax.ws.rs.core;

public interface FeatureContext extends Configurable<FeatureContext> {
}
```

The **Configurable<>** interface defines a variety of methods for registering filters and interceptors on a single resource method, as follows:

```
// Java
...
package javax.ws.rs.core;

import java.util.Map;

public interface Configurable<C extends Configurable> {
    public Configuration getConfiguration();
    public C property(String name, Object value);
    public C register(Class<?> componentClass);
    public C register(Class<?> componentClass, int priority);
    public C register(Class<?> componentClass, Class<?>... contracts);
    public C register(Class<?> componentClass, Map<Class<?>, Integer> contracts);
    public C register(Object component);
    public C register(Object component, int priority);
    public C register(Object component, Class<?>... contracts);
    public C register(Object component, Map<Class<?>, Integer> contracts);
}
```

CHAPTER 62. APACHE CXF MESSAGE PROCESSING PHASES

INBOUND PHASES

[Table 62.1, “Inbound message processing phases”](#) lists the phases available in inbound interceptor chains.

Table 62.1. Inbound message processing phases

Phase	Description
RECEIVE	Performs transport specific processing, such as determining MIME boundaries for binary attachments.
PRE_STREAM	Processes the raw data stream received by the transport.
USER_STREAM	
POST_STREAM	
READ	Determines if a request is a SOAP or XML message and builds adds the proper interceptors. SOAP message headers are also processed in this phase.
PRE_PROTOCOL	Performs protocol level processing. This includes processing of WS-* headers and processing of the SOAP message properties.
USER_PROTOCOL	
POST_PROTOCOL	
UNMARSHAL	Unmarshals the message data into the objects used by the application level code.
PRE_LOGICAL	Processes the unmarshalled message data.
USER_LOGICAL	
POST_LOGICAL	
PRE_INVOKE	
INVOKE	Passes the message to the application code. On the server side, the service implementation is invoked in this phase. On the client side, the response is handed back to the application.

Phase	Description
POST_INVOKE	Invokes the outbound interceptor chain.

OUTBOUND PHASES

[Table 62.2, “Inbound message processing phases”](#) lists the phases available in inbound interceptor chains.

Table 62.2. Inbound message processing phases

Phase	Description
SETUP	Performs any set up that is required by later phases in the chain.
PRE_LOGICAL	Performs processing on the unmarshalled data passed from the application level.
USER_LOGICAL	
POST_LOGICAL	
PREPARE_SEND	Opens the connection for writing the message on the wire.
PRE_STREAM	Performs processing required to prepare the message for entry into a data stream.
PRE_PROTOCOL	Begins processing protocol specific information.
WRITE	Writes the protocol message.
PRE_MARSHAL	Marshals the message.
MARSHAL	
POST_MARSHAL	
USER_PROTOCOL	Process the protocol message.
POST_PROTOCOL	
USER_STREAM	Process the byte-level message.
POST_STREAM	

Phase	Description
SEND	Sends the message and closes the transport stream.



IMPORTANT

Outbound interceptor chains have a mirror set of ending phases whose names are appended with **_ENDING**. The ending phases are used interceptors that require some terminal action to occur before data is written on the wire.

CHAPTER 63. APACHE CXF PROVIDED INTERCEPTORS

63.1. CORE APACHE CXF INTERCEPTORS

Inbound

[Table 63.1, “Core inbound interceptors”](#) lists the core inbound interceptors that are added to all Apache CXF endpoints.

Table 63.1. Core inbound interceptors

Class	Phase	Description
ServiceInvokerInterceptor	INVOK E	Invokes the proper method on the service.

Outbound

The Apache CXF does not add any core interceptors to the outbound interceptor chain by default. The contents of an endpoint’s outbound interceptor chain depend on the features in use.

63.2. FRONT-ENDS

JAX-WS

[Table 63.2, “Inbound JAX-WS interceptors”](#) lists the interceptors added to a JAX-WS endpoint’s inbound message chain.

Table 63.2. Inbound JAX-WS interceptors

Class	Phase	Description
HolderInInterceptor	PRE_INVOK E	Creates holder objects for any out or in/out parameters in the message.
WrapperClassInInterceptor	POST_LOGICAL	Unwraps the parts of a wrapped doc/literal message into the appropriate array of objects.
LogicalHandlerInInterceptor	PRE_PROTOCOL	Passes message processing to the JAX-WS logical handlers used by the endpoint. When the JAX-WS handlers complete, the message is passed along to the next interceptor on the inbound chain.

Class	Phase	Description
SOAPHandlerInterceptor	PRE_PROTOCOL	Passes message processing to the JAX-WS SOAP handlers used by the endpoint. When the SOAP handlers finish with the message, the message is passed along to the next interceptor in the chain.

Table 63.3, “Outbound JAX-WS interceptors” lists the interceptors added to a JAX-WS endpoint’s outbound message chain.

Table 63.3. Outbound JAX-WS interceptors

Class	Phase	Description
HolderOutInterceptor	PRE_LOGICAL	Removes the values of any out and in/out parameters from their holder objects and adds the values to the message’s parameter list.
WebFaultOutInterceptor	PRE_PROTOCOL	Processes outbound fault messages.
WrapperClassOutInterceptor	PRE_LOGICAL	Makes sure that wrapped doc/literal messages and rpc/literal messages are properly wrapped before being added to the message.
LogicalHandlerOutInterceptor	PRE_MARSHAL	Passes message processing to the JAX-WS logical handlers used by the endpoint. When the JAX-WS handlers complete, the message is passed along to the next interceptor on the outbound chain.
SOAPHandlerInterceptor	PRE_PROTOCOL	Passes message processing to the JAX-WS SOAP handlers used by the endpoint. When the SOAP handlers finish processing the message, it is passed along to the next interceptor in the chain.
MessageSenderInterceptor	PREPARE_SEND	Calls back to the Destination object to have it setup the output streams, headers, etc. to prepare the outgoing transport.

JAX-RS

[Table 63.4, “Inbound JAX-RS interceptors”](#) lists the interceptors added to a JAX-RS endpoint’s inbound message chain.

Table 63.4. Inbound JAX-RS interceptors

Class	Phase	Description
JAXRSInInterceptor	PRE_STREAM	Selects the root resource class, invokes any configured JAX-RS request filters, and determines the method to invoke on the root resource.



IMPORTANT

The inbound chain for a JAX-RS endpoint skips straight to the **ServiceInvokerInInterceptor** interceptor. No other interceptors will be invoked after the **JAXRSInInterceptor**.

[Table 63.5, “Outbound JAX-RS interceptors”](#) lists the interceptors added to a JAX-RS endpoint’s outbound message chain.

Table 63.5. Outbound JAX-RS interceptors

Class	Phase	Description
JAXRSOutInterceptor	MARSHAL	Marshals the response into the proper format for transmission.

63.3. MESSAGE BINDINGS

SOAP

[Table 63.6, “Inbound SOAP interceptors”](#) lists the interceptors added to a endpoint’s inbound message chain when using the SOAP Binding.

Table 63.6. Inbound SOAP interceptors

Class	Phase	Description
CheckFaultInterceptor	POST_PROTOCOL	Checks if the message is a fault message. If the message is a fault message, normal processing is aborted and fault processing is started.
MustUnderstandInterceptor	PRE_PROTOCOL	Processes the must understand headers.

Class	Phase	Description
RPCInInterceptor	UNMARSHAL	Unmarshals rpc/literal messages. If the message is bare, the message is passed to a BareInInterceptor object to deserialize the message parts.
ReadsHeadersInterceptor	READ	Parses the SOAP headers and stores them in the message object.
SapActionInInterceptor	READ	Parses the SOAP action header and attempts to find a unique operation for the action.
SapHeaderInterceptor	UNMARSHAL	Binds the SOAP headers that map to operation parameters to the appropriate objects.
AttachmentInInterceptor	RECEIVE	Parses the mime headers for mime boundaries, finds the root part and resets the input stream to it, and stores the other parts in a collection of Attachment objects.
DocLiteralInInterceptor	UNMARSHAL	Examines the first element in the SOAP body to determine the appropriate operation and calls the data binding to read in the data.
StaxInInterceptor	POST_STREAM	Creates an XMLStreamReader object from the message.
URI MappingInInterceptor	UNMARSHAL	Handles the processing of HTTP GET methods.
SwAInInterceptor	PRE_INVOKE	Creates the required MIME handlers for binary SOAP attachments and adds the data to the parameter list.

Table 63.7, “Outbound SOAP interceptors” lists the interceptors added to a endpoint’s outbound message chain when using the SOAP Binding.

Table 63.7. Outbound SOAP interceptors

Class	Phase	Description
RPCOutInterceptor	MARSHAL	Marshals rpc style messages for transmission.
SapHeaderOutFilterIntercep tor	PRE_LOGICAL	Removes all SOAP headers that are marked as inbound only.
SapPreProtocolOutIntercep tor	POST_LOGICAL	Sets up the SOAP version and the SOAP action header.
AttachmentOutInterceptor	PRE_STREAM	Sets up the attachment marshalers and the mime stuff required to process any attachments that might be in the message.
BareOutInterceptor	MARSHAL	Writes the message parts.
StaxOutInterceptor	PRE_STREAM	Creates an XMLStreamWriter object from the message.
WrappedOutInterceptor	MARSHAL	Wraps the outbound message parameters.
SapOutInterceptor	WRITE	Writes the soap:envelope element and the elements for the header blocks in the message. Also writes an empty soap:body element for the remaining interceptors to populate.
SwAOutInterceptor	PRE_LOGICAL	Removes any binary data that will be packaged as a SOAP attachment and stores it for later processing.

XML

Table 63.8, “Inbound XML interceptors” lists the interceptors added to a endpoint’s inbound message chain when using the XML Binding.

Table 63.8. Inbound XML interceptors

Class	Phase	Description
-------	-------	-------------

Class	Phase	Description
AttachmentInInterceptor	RECEIVE	Parses the mime headers for mime boundaries, finds the root part and resets the input stream to it, and then stores the other parts in a collection of Attachment objects.
DocLiteralInInterceptor	UNMARSHAL	Examines the first element in the message body to determine the appropriate operation and then calls the data binding to read in the data.
StaxInInterceptor	POST_STREAM	Creates an XMLStreamReader object from the message.
URIMappingInterceptor	UNMARSHAL	Handles the processing of HTTP GET methods.
XMLMessageInInterceptor	UNMARSHAL	Unmarshals the XML message.

Table 63.9, “Outbound XML interceptors” lists the interceptors added to a endpoint’s outbound message chain when using the XML Binding.

Table 63.9. Outbound XML interceptors

Class	Phase	Description
StaxOutInterceptor	PRE_STREAM	Creates an XMLStreamWriter objects from the message.
WrappedOutInterceptor	MARSHAL	Wraps the outbound message parameters.
XMLMessageOutInterceptor	MARSHAL	Marshals the message for transmission.

CORBA

Table 63.10, “Inbound CORBA interceptors” lists the interceptors added to a endpoint’s inbound message chain when using the CORBA Binding.

Table 63.10. Inbound CORBA interceptors

Class	Phase	Description
CorbaStreamInInterceptor	PRE_STREAM	Deserializes the CORBA message.
BareInInterceptor	UNMARSHAL	Deserializes the message parts.

[Table 63.11, "Outbound CORBA interceptors"](#) lists the interceptors added to a endpoint's outbound message chain when using the CORBA Binding.

Table 63.11. Outbound CORBA interceptors

Class	Phase	Description
CorbaStreamOutInterceptor	PRE_STREAM	Serializes the message.
BareOutInterceptor	MARSHAL	Writes the message parts.
CorbaStreamOutEndingInterceptor	USER_STREAM	Creates a streamable object for the message and stores it in the message context.

63.4. OTHER FEATURES

Logging

[Table 63.12, "Inbound logging interceptors"](#) lists the interceptors added to a endpoint's inbound message chain to support logging.

Table 63.12. Inbound logging interceptors

Class	Phase	Description
LoggingInInterceptor	RECEIVE	Writes the raw message data to the logging system.

[Table 63.13, "Outbound logging interceptors"](#) lists the interceptors added to a endpoint's outbound message chain to support logging.

Table 63.13. Outbound logging interceptors

Class	Phase	Description
LoggingOutInterceptor	PRE_STREAM	Writes the outbound message to the logging system.

For more information about logging see [Chapter 19, Apache CXF Logging](#).

WS-Addressing

[Table 63.14, "Inbound WS-Addressing interceptors"](#) lists the interceptors added to a endpoint's inbound message chain when using WS-Addressing.

Table 63.14. Inbound WS-Addressing interceptors

Class	Phase	Description
MAPCodec	PRE_PROTOCOL	Decodes the message addressing properties.

[Table 63.15, "Outbound WS-Addressing interceptors"](#) lists the interceptors added to a endpoint's outbound message chain when using WS-Addressing.

Table 63.15. Outbound WS-Addressing interceptors

Class	Phase	Description
MAPAggregator	PRE_LOGICAL	Aggregates the message addressing properties for a message.
MAPCodec	PRE_PROTOCOL	Encodes the message addressing properties.

For more information about WS-Addressing see [Chapter 20, Deploying WS-Addressing](#).

WS-RM



IMPORTANT

WS-RM relies on WS-Addressing so all of the WS-Addressing interceptors will also be added to the interceptor chains.

[Table 63.16, "Inbound WS-RM interceptors"](#) lists the interceptors added to a endpoint's inbound message chain when using WS-RM.

Table 63.16. Inbound WS-RM interceptors

Class	Phase	Description
RMIInterceptor	PRE_LOGICAL	Handles the aggregation of message parts and acknowledgement messages.
RMSoapInterceptor	PRE_PROTOCOL	Encodes and decodes the WS-RM properties from messages.

[Table 63.17, "Outbound WS-RM interceptors"](#) lists the interceptors added to a endpoint's outbound message chain when using WS-RM.

Table 63.17. Outbound WS-RM interceptors

Class	Phase	Description
RMOutInterceptor	PRE_LOGICAL	Handles the chunking of messages and the transmission of the chunks. Also handles the processing of acknowledgements and resend requests.
RMSoapInterceptor	PRE_PROTOCOL	Encodes and decodes the WS-RM properties from messages.

For more information about WS-RM see [Chapter 21, Enabling Reliable Messaging](#).

CHAPTER 64. INTERCEPTOR PROVIDERS

OVERVIEW

Interceptor providers are objects in the Apache CXF runtime that have interceptor chains attached to them. They all implement the org.apache.cxf.interceptor.InterceptorProvider interface. Developers can attach their own interceptors to any interceptor provider.

LIST OF PROVIDERS

The following objects are interceptor providers:

- **AddressingPolicyInterceptorProvider**
- **ClientFactoryBean**
- **ClientImpl**
- **ClientProxyFactoryBean**
- **CorbaBinding**
- **CXFBusImpl**
- **org.apache.cxf.jaxws.EndpointImpl**
- **org.apache.cxf.endpoint.EndpointImpl**
- **ExtensionManagerBus**
- **JAXRSClientFactoryBean**
- **JAXRSServerFactoryBean**
- **JAXRSServiceImpl**
- **JaxWsClientEndpointImpl**
- **JaxWsClientFactoryBean**
- **JaxWsEndpointImpl**
- **JaxWsProxyFactoryBean**
- **JaxWsServerFactoryBean**
- **JaxwsServiceBuilder**
- **MTOMPolicyInterceptorProvider**
- **NoOpPolicyInterceptorProvider**
- **ObjectBinding**
- **RMPolicyInterceptorProvider**

- **ServerFactoryBean**
- **ServiceImpl**
- **SimpleServiceBuilder**
- **SoapBinding**
- **WrappedEndpoint**
- **WrappedService**
- **XMLBinding**

PART VIII. APACHE CXF FEATURES

This guide describes how to enable various advanced features of Apache CXF.

CHAPTER 65. BEAN VALIDATION

Abstract

Bean validation is a Java standard that enables you to define runtime constraints by adding Java annotations to service classes or interfaces. Apache CXF uses interceptors to integrate this feature with Web service method invocations.

65.1. INTRODUCTION

Overview

Bean Validation 1.1 ([JSR-349](#))—which is an evolution of the original Bean Validation 1.0 (JSR-303) standard—enables you to declare constraints that can be checked at run time, using Java annotations. You can use annotations to define constraints on the following parts of the Java code:

- Fields in a bean class.
- Method and constructor parameters.
- Method return values.

Example of annotated class

The following example shows a Java class annotated with some standard bean validation constraints:

```
// Java
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Max;
import javax.validation.Valid;
...
public class Person {
    @NotNull private String firstName;
    @NotNull private String lastName;
    @Valid @NotNull private Person boss;

    public @NotNull String saveItem( @Valid @NotNull Person person, @Max( 23 ) BigDecimal age )
    {
        // ...
    }
}
```

Bean validation or schema validation?

In some respects, bean validation and schema validation are quite similar. Configuring an endpoint with an XML schema is a well established way to validate messages at run time on a Web services endpoint. An XML schema can check many of the same constraints as bean validation on incoming and outgoing messages. Nevertheless, bean validation can sometimes be a useful alternative for one or more of the following reasons:

- Bean validation enables you to define constraints independently of the XML schema (which is useful, for example, in the case of code-first service development).

- If your current XML schema is too lax, you can use bean validation to define stricter constraints.
- Bean validation lets you define custom constraints, which might be impossible to define using XML schema language.

Dependencies

The Bean Validation 1.1 (JSR-349) standard defines just the API, not the implementation. Dependencies must therefore be provided in two parts:

- **Core dependencies**—provide the bean validation 1.1 API, Java unified expression language API and implementation.
- **Hibernate Validator dependencies**—provides the implementation of bean validation 1.1.

Core dependencies

To use bean validation, you must add the following core dependencies to your project's Maven **pom.xml** file:

```
<dependency>
  <groupId>javax.validation</groupId>
  <artifactId>validation-api</artifactId>
  <version>1.1.0.Final</version>
</dependency>
<dependency>
  <groupId>javax.el</groupId>
  <artifactId>javax.el-api</artifactId>
  <!-- use 3.0-b02 version for Java 6 -->
  <version>3.0.0</version>
</dependency>
<dependency>
  <groupId>org.glassfish</groupId>
  <artifactId>javax.el</artifactId>
  <!-- use 3.0-b01 version for Java 6 -->
  <version>3.0.0</version>
</dependency>
```



NOTE

The **javax.el/javax.el-api** and **org.glassfish/javax.el** dependencies provide the API and implementation of Java's unified expression language. This expression language is used internally by bean validation, but is not important at the application programming level.

Hibernate Validator dependencies

To use the **Hibernate Validator** implementation of bean validation, you must add the following additional dependencies to your project's Maven **pom.xml** file:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>5.0.3.Final</version>
</dependency>
```

Resolving the validation provider in an OSGi environment

The default mechanism for resolving a validation provider involves scanning the classpath to find the provider resource. In the case of an OSGi (Apache Karaf) environment, however, this mechanism does **not** work, because the validation provider (for example, the Hibernate validator) is packaged in a separate bundle and is thus not automatically available in your application classpath. In the context of OSGi, the Hibernate validator needs to be wired to your application bundle, and OSGi needs a bit of help to do this successfully.

Configuring the validation provider explicitly in OSGi

In the context of OSGi, you need to configure the validation provider explicitly, instead of relying on automatic discovery. For example, if you are using the common validation feature (see [the section called "Bean validation feature"](#)) to enable bean validation, you must configure it with a validation provider, as follows:

```
<bean id="commonValidationFeature" class="org.apache.cxf.validation.BeanValidationFeature">
    <property name="provider" ref="beanValidationProvider"/>
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

Where the **HibernateValidationProviderResolver** is a custom class that wraps the Hibernate validation provider.

Example `HibernateValidationProviderResolver` class

The following code example shows how to define a custom **HibernateValidationProviderResolver**, which resolves the Hibernate validator:

```
// Java
package org.example;

import static java.util.Collections.singletonList;
import org.hibernate.validator.HibernateValidator;
import javax.validation.ValidationProviderResolver;
import java.util.List;

/**
 * OSGi-friendly implementation of {@code javax.validation.ValidationProviderResolver} returning
 * {@code org.hibernate.validator.HibernateValidator} instance.
 */
public class HibernateValidationProviderResolver implements ValidationProviderResolver {

    @Override
    public List<ValidationProviderResolver> getValidationProviders() {
        return singletonList(new HibernateValidator());
    }
}
```

When you build the preceding class in a Maven build system, which is configured to use the Maven bundle plug-in, your application will be wired to the Hibernate validator bundle at deploy time (assuming you have already deployed the Hibernate validator bundle to the OSGi container).

65.2. DEVELOPING SERVICES WITH BEAN VALIDATION

65.2.1. Annotating a Service Bean

Overview

The first step in developing a service with bean validation is to apply the relevant validation annotations to the Java classes or interfaces that represent your services. The validation annotations enable you to apply constraints to method parameters, return values, and class fields, which are then checked at run time, every time the service is invoked.

Validating simple input parameters

To validate the parameters of a service method—where the parameters are simple Java types—you can apply any of the constraint annotations from the bean validation API (**javax.validation.constraints** package). For example, the following code example tests both parameters for nullness (**@NotNull** annotation), whether the **id** string matches the `\d+` regular expression (**@Pattern** annotation), and whether the length of the **name** string lies in the range 1 to 50:

```
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;
...
@POST
@Path("/books")
public Response addBook(
    @NotNull @Pattern(regexp = "\d+") @FormParam("id") String id,
    @NotNull @Size(min = 1, max = 50) @FormParam("name") String name) {
    // do some work
    return Response.created().build();
}
```

Validating complex input parameters

To validate complex input parameters (object instances), apply the **@Valid** annotation to the parameter, as shown in the following example:

```
import javax.validation.Valid;
...
@POST
@Path("/books")
public Response addBook( @Valid Book book ) {
    // do some work
    return Response.created().build();
}
```

The **@Valid** annotation does not specify any constraints by itself. When you annotate the **Book** parameter with **@Valid**, you are effectively telling the validation engine to look inside the definition of the **Book** class (recursively) to look for validation constraints. In this example, the **Book** class is defined

with validation constraints on its **id** and **name** fields, as follows:

```
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;
...
public class Book {
    @NotNull @Pattern(regexp = "\d+") private String id;
    @NotNull @Size(min = 1, max = 50) private String name;
    ...
}
```

Validating return values (non-Response)

To apply validation to regular method return values (non-Response), add the annotations in front of the method signature. For example, to test the return value for nullness (**@NotNull** annotation) and to test validation constraints recursively (**@Valid** annotation), annotate the **getBook** method as follows:

```
import javax.validation.constraints.NotNull;
import javax.validation.Valid;
...
@GET
@Path("/books/{bookId}")
@Override
@NotNull @Valid
public Book getBook(@PathParam("bookId") String id) {
    return new Book( id );
}
```

Validating return values (Response)

To apply validation to a method that returns a **javax.ws.rs.core.Response** object, you can use the same annotations as in the non-Response case. For example:

```
import javax.validation.constraints.NotNull;
import javax.validation.Valid;
import javax.ws.rs.core.Response;
...
@GET
@Path("/books/{bookId}")
@Valid @NotNull
public Response getBookResponse(@PathParam("bookId") String id) {
    return Response.ok( new Book( id ) ).build();
}
```

65.2.2. Standard Annotations

Bean validation constraints

[Table 65.1, “Standard Annotations for Bean Validation”](#) shows the standard annotations defined in the Bean Validation specification, which can be used to define constraints on fields and on method return values and parameters (none of the standard annotations can be applied at the class level).

Table 65.1. Standard Annotations for Bean Validation

Annotation	Applicable to	Description
@AssertFalse	Boolean, boolean	Checks that the annotated element is false .
@AssertTrue	Boolean, boolean	Checks that the annotated element is true .
@DecimalMax(value=, inclusive=)	BigDecimal, BigInteger, CharSequence, byte, short, int, long and primitive type wrappers	When inclusive=false , checks that the annotated value is less than the specified maximum. Otherwise, checks that the value is less than or equal to the specified maximum. The value parameter specifies the maximum in BigDecimal string format.
@DecimalMin(value=, inclusive=)	BigDecimal, BigInteger, CharSequence, byte, short, int, long and primitive type wrappers	When inclusive=false , checks that the annotated value is greater than the specified minimum. Otherwise, checks that the value is greater than or equal to the specified minimum. The value parameter specifies the minimum in BigDecimal string format.
@Digits(integer=, fraction=)	BigDecimal, BigInteger, CharSequence, byte, short, int, long and primitive type wrappers	Checks whether the annotated value is a number having up to integer digits and fraction fractional digits.
@Future	java.util.Date, java.util.Calendar	Checks whether the annotated date is in the future.
@Max(value=)	BigDecimal, BigInteger, CharSequence, byte, short, int, long and primitive type wrappers	Checks whether the annotated value is less than or equal to the specified maximum.
@Min(value=)	BigDecimal, BigInteger, CharSequence, byte, short, int, long and primitive type wrappers	Checks whether the annotated value is greater than or equal to the specified minimum.
@NotNull	Any type	Checks that the annotated value is not null .
@Null	Any type	Checks that the annotated value is null .

Annotation	Applicable to	Description
@Past	java.util.Date, java.util.Calendar	Checks whether the annotated date is in the past.
@Pattern(regex=, flag=)	CharSequence	Checks whether the annotated string matches the regular expression regex considering the given flag match.
@Size(min=, max=)	CharSequence, Collection, Map and arrays	Checks whether the size of the annotated collection, map, or array lies between min and max (inclusive).
@Valid	Any non-primitive type	Performs validation recursively on the annotated object. If the object is a collection or an array, the elements are validated recursively. If the object is a map, the value elements are validated recursively.

65.2.3. Custom Annotations

Defining custom constraints in Hibernate

It is possible to define your own custom constraints annotations with the bean validation API. For details of how to do this in the Hibernate validator implementation, see the [Creating custom constraints](#) chapter of the [Hibernate Validator Reference Guide](#).

65.3. CONFIGURING BEAN VALIDATION

65.3.1. JAX-WS Configuration

Overview

This section describes how to enable bean validation on a JAX-WS service endpoint, which is defined either in Blueprint XML or in Spring XML. The interceptors used to perform bean validation are common to both JAX-WS endpoints and JAX-RS 1.1 endpoints (JAX-RS 2.0 endpoints use different interceptor classes, however).

Namespaces

In the XML examples shown in this section, you must remember to map the **jaxws** namespace prefix to the appropriate namespace, either for Blueprint or Spring, as shown in the following table:

XML Language	Namespace
Blueprint	http://cxf.apache.org/blueprint/jaxws
Spring	http://cxf.apache.org/jaxws

Bean validation feature

The simplest way to enable bean validation on a JAX-WS endpoint is to add the *bean validation feature* to the endpoint. The bean validation feature is implemented by the following class:

org.apache.cxf.validation.BeanValidationFeature

By adding an instance of this feature class to the JAX-WS endpoint (either through the Java API or through the **jaxws:features** child element of **jaxws:endpoint** in XML), you can enable bean validation on the endpoint. This feature installs two interceptors: an **In** interceptor that validates incoming message data; and an **Out** interceptor that validates return values (where the interceptors are created with default configuration parameters).

Sample JAX-WS configuration with bean validation feature

The following XML example shows how to enable bean validation functionality in a JAX-WS endpoint, by adding the **commonValidationFeature** bean to the endpoint as a JAX-WS feature:

```
<jaxws:endpoint xmlns:s="http://bookworld.com"
    serviceName="s:BookWorld"
    endpointName="s:BookWorldPort"
    implementor="#bookWorldValidation"
    address="/bwsoap">
    <jaxws:features>
        <ref bean="commonValidationFeature" />
    </jaxws:features>
</jaxws:endpoint>

<bean id="bookWorldValidation"
    class="org.apache.cxf.systest.jaxrs.validation.spring.BookWorldImpl"/>

<bean id="commonValidationFeature" class="org.apache.cxf.validation.BeanValidationFeature">
    <property name="provider" ref="beanValidationProvider"/>
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

For a sample implementation of the **HibernateValidationProviderResolver** class, see the section called “[Example HibernateValidationProviderResolver class](#)”. It is only necessary to configure the **beanValidationProvider** in the context of an OSGi environment (Apache Karaf).

**NOTE**

Remember to map the **jaxws** prefix to the appropriate XML namespace for either Blueprint or Spring, depending on the context.

Common bean validation 1.1 interceptors

If you want to have more fine-grained control over the configuration of the bean validation, you can install the interceptors individually, instead of using the bean validation feature. In place of the bean validation feature, you can configure one or both of the following interceptors:

org.apache.cxf.validation.BeanValidationInInterceptor

When installed in a JAX-WS (or JAX-RS 1.1) endpoint, validates resource method parameters against validation constraints. If validation fails, raises the **javax.validation.ConstraintViolationException** exception. To install this interceptor, add it to the endpoint through the **jaxws:inInterceptors** child element in XML (or the **jaxrs:inInterceptors** child element in XML).

org.apache.cxf.validation.BeanValidationOutInterceptor

When installed in a JAX-WS (or JAX-RS 1.1) endpoint, validates response values against validation constraints. If validation fails, raises the **javax.validation.ConstraintViolationException** exception. To install this interceptor, add it to the endpoint through the **jaxws:outInterceptors** child element in XML (or the **jaxrs:outInterceptors** child element in XML).

Sample JAX-WS configuration with bean validation interceptors

The following XML example shows how to enable bean validation functionality in a JAX-WS endpoint, by explicitly adding the relevant **In** interceptor bean and **Out** interceptor bean to the endpoint:

```
<jaxws:endpoint xmlns:s="http://bookworld.com"
    serviceName="s:BookWorld"
    endpointName="s:BookWorldPort"
    implementor="#bookWorldValidation"
    address="/bwsoap">
<jaxws:inInterceptors>
    <ref bean="validationInInterceptor" />
</jaxws:inInterceptors>

<jaxws:outInterceptors>
    <ref bean="validationOutInterceptor" />
</jaxws:outInterceptors>
</jaxws:endpoint>

<bean id="bookWorldValidation"
    class="org.apache.cxf.systest.jaxrs.validation.spring.BookWorldImpl"/>

<bean id="validationInInterceptor" class="org.apache.cxf.validation.BeanValidationInInterceptor">
    <property name="provider" ref="beanValidationProvider"/>
</bean>
<bean id="validationOutInterceptor" class="org.apache.cxf.validation.BeanValidationOutInterceptor">
    <property name="provider" ref="beanValidationProvider"/>
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
```

```
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

For a sample implementation of the **HibernateValidationProviderResolver** class, see the section called “[Example HibernateValidationProviderResolver class](#)”. It is only necessary to configure the **beanValidationProvider** in the context of an OSGi environment (Apache Karaf).

Configuring a BeanValidationProvider

The **org.apache.cxf.validation.BeanValidationProvider** is a simple wrapper class that wraps the bean validation implementation (**validation provider**). By overriding the default **BeanValidationProvider** class, you can customize the implementation of bean validation. The **BeanValidationProvider** bean enables you to override one or more of the following provider classes:

[javax.validation.ParameterNameProvider](#)

Provides names for method and constructor parameters. Note that this class is needed, because the Java reflection API does **not** give you access to the names of method parameters or constructor parameters.

[javax.validation.spi.ValidationProvider<T>](#)

Provides an implementation of bean validation for the specified type, **T**. By implementing your own **ValidationProvider** class, you can define custom validation rules for your own classes. This mechanism effectively enables you to extend the bean validation framework.

[javax.validation.ValidationProviderResolver](#)

Implements a mechanism for discovering **ValidationProvider** classes and returns a list of the discovered classes. The default resolver looks for a **META-INF/services/javax.validation.spi.ValidationProvider** file on the classpath, which should contain a list of **ValidationProvider** classes.

[javax.validation.ValidatorFactory](#)

A factory that returns **javax.validation.Validator** instances.

[org.apache.cxf.validation.ValidationConfiguration](#)

A CXF wrapper class that enables you override more classes from the validation provider layer.

To customize the **BeanValidationProvider**, pass a custom **BeanValidationProvider** instance to the constructor of the validation **In** interceptor and to the constructor of the validation **Out** interceptor. For example:

```
<bean id="validationProvider" class="org.apache.cxf.validation.BeanValidationProvider" />

<bean id="validationInInterceptor" class="org.apache.cxf.validation.BeanValidationInInterceptor">
    <property name="provider" ref="validationProvider" />
</bean>

<bean id="validationOutInterceptor" class="org.apache.cxf.validation.BeanValidationOutInterceptor">
    <property name="provider" ref="validationProvider" />
</bean>
```

65.3.2. JAX-RS Configuration

Overview

This section describes how to enable bean validation on a JAX-RS service endpoint, which is defined either in Blueprint XML or in Spring XML. The interceptors used to perform bean validation are common to both JAX-WS endpoints and JAX-RS 1.1 endpoints (JAX-RS 2.0 endpoints use different interceptor classes, however).

Namespaces

In the XML examples shown in this section, you must remember to map the **jaxws** namespace prefix to the appropriate namespace, either for Blueprint or Spring, as shown in the following table:

XML Language	Namespace
Blueprint	http://cxf.apache.org/blueprint/jaxws
Spring	http://cxf.apache.org/jaxws

Bean validation feature

The simplest way to enable bean validation on a JAX-RS endpoint is to add the *bean validation feature* to the endpoint. The bean validation feature is implemented by the following class:

org.apache.cxf.validation.BeanValidationFeature

By adding an instance of this feature class to the JAX-RS endpoint (either through the Java API or through the **jaxrs:features** child element of **jaxrs:server** in XML), you can enable bean validation on the endpoint. This feature installs two interceptors: an **In** interceptor that validates incoming message data; and an **Out** interceptor that validates return values (where the interceptors are created with default configuration parameters).

Validation exception mapper

A JAX-RS endpoint also requires you to configure a *validation exception mapper*, which is responsible for mapping validation exceptions to HTTP error responses. The following class implements validation exception mapping for JAX-RS:

org.apache.cxf.jaxrs.validation.ValidationExceptionMapper

Implements validation exception mapping in accordance with the JAX-RS 2.0 specification: any input parameter validation violations are mapped to HTTP status code **400 Bad Request**; and any return value validation violation (or internal validation violation) is mapped to HTTP status code **500 Internal Server Error**.

Sample JAX-RS configuration

The following XML example shows how to enable bean validation functionality in a JAX-RS endpoint, by adding the **commonValidationFeature** bean as a JAX-RS feature and by adding the **exceptionMapper** bean as a JAX-RS provider:

```
<jaxrs:server address="/bwrest">
    <jaxrs:serviceBeans>
        <ref bean="bookWorldValidation"/>
    </jaxrs:serviceBeans>
    <jaxrs:providers>
        <ref bean="exceptionMapper"/>
    </jaxrs:providers>
</jaxrs:server>
```

```

</jaxrs:providers>
<jaxrs:features>
    <ref bean="commonValidationFeature" />
</jaxrs:features>
</jaxrs:server>

<bean id="bookWorldValidation"
class="org.apache.cxf.systest.jaxrs.validation.spring.BookWorldImpl"/>
<beanid="exceptionMapper"class="org.apache.cxf.jaxrs.validation.ValidationExceptionMapper"/>

<bean id="commonValidationFeature" class="org.apache.cxf.validation.BeanValidationFeature">
    <property name="provider" ref="beanValidationProvider"/>
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>

```

For a sample implementation of the **HibernateValidationProviderResolver** class, see [the section called “Example HibernateValidationProviderResolver class”](#). It is only necessary to configure the **beanValidationProvider** in the context of an OSGi environment (Apache Karaf).



NOTE

Remember to map the **jaxrs** prefix to the appropriate XML namespace for either Blueprint or Spring, depending on the context.

Common bean validation 1.1 interceptors

Instead of using the bean validation feature, you can optionally install bean validation interceptors to get more fine-grained control over the validation implementation. JAX-RS uses the same interceptors as JAX-WS for this purpose—see [the section called “Common bean validation 1.1 interceptors”](#)

Sample JAX-RS configuration with bean validation interceptors

The following XML example shows how to enable bean validation functionality in a JAX-RS endpoint, by explicitly adding the relevant **In** interceptor bean and **Out** interceptor bean to the server endpoint:

```

<jaxrs:server address="/">
    <jaxrs:inInterceptors>
        <ref bean="validationInInterceptor" />
    </jaxrs:inInterceptors>

    <jaxrs:outInterceptors>
        <ref bean="validationOutInterceptor" />
    </jaxrs:outInterceptors>

    <jaxrs:serviceBeans>
        ...
    </jaxrs:serviceBeans>

    <jaxrs:providers>
        <ref bean="exceptionMapper"/>
    </jaxrs:providers>

```

```

</jaxrs:providers>
</jaxrs:server>

<bean id="exceptionMapper" class="org.apache.cxf.jaxrs.validation.ValidationExceptionMapper"/>

<bean id="validationInInterceptor" class="org.apache.cxf.validation.BeanValidationInInterceptor">
    <property name="provider" ref="beanValidationProvider" />
</bean>

<bean id="validationOutInterceptor" class="org.apache.cxf.validation.BeanValidationOutInterceptor">
    <property name="provider" ref="beanValidationProvider" />
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>

```

For a sample implementation of the **HibernateValidationProviderResolver** class, see [the section called "Example HibernateValidationProviderResolver class"](#). It is only necessary to configure the **beanValidationProvider** in the context of an OSGi environment (Apache Karaf).

Configuring a BeanValidationProvider

You can inject a custom **BeanValidationProvider** instance into the validation interceptors, as described in [the section called "Configuring a BeanValidationProvider"](#).

65.3.3. JAX-RS 2.0 Configuration

Overview

Unlike JAX-RS 1.1 (which shares common validation interceptors with JAX-WS), the JAX-RS 2.0 configuration relies on dedicated validation interceptor classes that are specific to JAX-RS 2.0.

Bean validation feature

For JAX-RS 2.0, there is a dedicated bean validation feature, which is implemented by the following class:

org.apache.cxf.validation.JAXRSBeanValidationFeature

By adding an instance of this feature class to the JAX-RS endpoint (either through the Java API or through the **jaxrs:features** child element of **jaxrs:server** in XML), you can enable bean validation on a JAX-RS 2.0 server endpoint. This feature installs two interceptors: an **In** interceptor that validates incoming message data; and an **Out** interceptor that validates return values (where the interceptors are created with default configuration parameters).

Validation exception mapper

JAX-RS 2.0 uses the same validation exception mapper class as JAX-RS 1.x:

org.apache.cxf.jaxrs.validation.ValidationExceptionMapper

Implements validation exception mapping in accordance with the JAX-RS 2.0 specification: any input parameter validation violations are mapped to HTTP status code **400 Bad Request**; and any return

value validation violation (or internal validation violation) is mapped to HTTP status code **500 Internal Server Error**.

Bean validation invoker

If you configure the JAX-RS service with a non-default lifecycle policy (for example, using Spring lifecycle management), you should also register a **org.apache.cxf.jaxrs.validation.JAXRSBeanValidationInvoker** instance—using the **jaxrs:invoker** element in the endpoint configuration—with the service endpoint, to ensure that bean validation is invoked correctly.

For more details about JAX-RS service lifecycle management, see [the section called “Lifecycle management in Spring XML”](#).

Sample JAX-RS 2.0 configuration with bean validation feature

The following XML example shows how to enable bean validation functionality in a JAX-RS 2.0 endpoint, by adding the **jaxrsValidationFeature** bean as a JAX-RS feature and by adding the **exceptionMapper** bean as a JAX-RS provider:

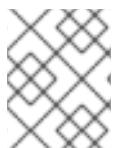
```
<jaxrs:server address="/">
  <jaxrs:serviceBeans>
    ...
  </jaxrs:serviceBeans>
  <jaxrs:providers>
    <ref bean="exceptionMapper"/>
  </jaxrs:providers>
  <jaxrs:features>
    <ref bean="jaxrsValidationFeature" />
  </jaxrs:features>
</jaxrs:server>

<bean id="exceptionMapper" class="org.apache.cxf.jaxrs.validation.ValidationExceptionMapper"/>
<bean id="jaxrsValidationFeature" class="org.apache.cxf.validation.JAXRSBeanValidationFeature">
  <property name="provider" ref="beanValidationProvider"/>
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
  <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

For a sample implementation of the **HibernateValidationProviderResolver** class, see [the section called “Example HibernateValidationProviderResolver class”](#). It is only necessary to configure the **beanValidationProvider** in the context of an OSGi environment (Apache Karaf).



NOTE

Remember to map the **jaxrs** prefix to the appropriate XML namespace for either Blueprint or Spring, depending on the context.

Common bean validation 1.1 interceptors

If you want to have more fine-grained control over the configuration of the bean validation, you can install the JAX-RS interceptors individually, instead of using the bean validation feature. Configure one or both of the following JAX-RS interceptors:

org.apache.cxf.validation.JAXRSBeanValidationInInterceptor

When installed in a JAX-RS 2.0 server endpoint, validates resource method parameters against validation constraints. If validation fails, raises the **javax.validation.ConstraintViolationException** exception. To install this interceptor, add it to the endpoint through the **jaxrs:inInterceptors** child element in XML.

org.apache.cxf.validation.JAXRSBeanValidationOutInterceptor

When installed in a JAX-RS 2.0 endpoint, validates response values against validation constraints. If validation fails, raises the **javax.validation.ConstraintViolationException** exception. To install this interceptor, add it to the endpoint through the **jaxrs:inInterceptors** child element in XML.

Sample JAX-RS 2.0 configuration with bean validation interceptors

The following XML example shows how to enable bean validation functionality in a JAX-RS 2.0 endpoint, by explicitly adding the relevant **In** interceptor bean and **Out** interceptor bean to the server endpoint:

```
<jaxrs:server address="/">
  <jaxrs:inInterceptors>
    <ref bean="validationInInterceptor" />
  </jaxrs:inInterceptors>

  <jaxrs:outInterceptors>
    <ref bean="validationOutInterceptor" />
  </jaxrs:outInterceptors>

  <jaxrs:serviceBeans>
...
  </jaxrs:serviceBeans>

  <jaxrs:providers>
    <ref bean="exceptionMapper"/>
  </jaxrs:providers>
</jaxrs:server>

<bean id="exceptionMapper" class="org.apache.cxf.jaxrs.validation.ValidationExceptionMapper"/>

<bean id="validationInInterceptor"
class="org.apache.cxf.jaxrs.validation.JAXRSBeanValidationInInterceptor">
  <property name="provider" ref="beanValidationProvider" />
</bean>

<bean id="validationOutInterceptor"
class="org.apache.cxf.jaxrs.validation.JAXRSBeanValidationOutInterceptor">
  <property name="provider" ref="beanValidationProvider" />
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
  <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

For a sample implementation of the **HibernateValidationProviderResolver** class, see [the section called "Example HibernateValidationProviderResolver class"](#). It is only necessary to configure the **beanValidationProvider** in the context of an OSGi environment (Apache Karaf).

Configuring a BeanValidationProvider

You can inject a custom **BeanValidationProvider** instance into the validation interceptors, as described in [the section called "Configuring a BeanValidationProvider"](#).

Configuring a JAXRSPParameterNameProvider

The **org.apache.cxf.jaxrs.validation.JAXRSPParameterNameProvider** class is an implementation of the **javax.validation.ParameterNameProvider** interface, which can be used to provide the names for method and constructor parameters in the context of JAX-RS 2.0 endpoints.