Enterprise Services Development

Common Service Runtime Developer’s Guide

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Revision** | **Description** | **Author** |
| 12/22/2010 | 1.1 | Initial Draft | Anu Maddipatla |
| 12/29/2010 | 1.2 | Updated sections for Transaction Handlers and getDocument vs. getObject | Frank Mayfield |
| 12/29/2010 | 1.3 | Updated custom version extractor | John McKinnon |
| 12/30/2010 | 1.4 | Updated the Security Section | Brian Glas |
| 12/30/2010 | 1.5 | Added Logging | Todd M |
| 12/30/2010 | 1.6 | Updated the Programmatic Versioning | Pradeepa Arvabhumi |
| 01/07/2011 | 1.7 | Updated validation and helper classes | Rani Chode |
| 01/07/2011 | 1.8 | Updated XSLT Versioning | Pradeepa Arvabhumi |
| 01/07/2011 | 1.9 | Merged changes that got overwritten | Anu Maddipatla |
| 01/12/2011 | 2.0 | Added project layout | Anu Maddipatla |
| 1/21/2011 | 2.1 | Added Workflow Counters | Brian Glas |
| 1/27/2011 | 2.2 | Added supported operations | Pallavi Kothakonda |
| 1/28/2011 | 2.3 | Revised Tracer & TransID section | Todd Melancon |
| 1/28/2011 | 2.4 | Updated the TOC and added content relating to definitionsconfig file in the versioning section. | Pradeepa Arvabhumi |
| 6/23/2011 | 2.5 | Overall cleanup and addition of Sensitive Data masking in DEBUG log | Frank Mayfield |
| 11/11/2011 | 3.0 | Overall cleanup and addition of Sections 1, 5, 15 & 20. | Ali Bahu |
| 6/05/2012 | 3.1 | Added references for Samples and Demos, Judge/Jury security setup. | John McKinnon  Ali Bahu |
| 9/25/2012 | 3.2 | Added sections for Chained Converter, Converter Plugin, Fault Barrier | Frank Mayfield |
| 1/24/2013 | 4.0 | Changed Spring version reference | John McKinnon |
| 6/11/2013 | 4.1 | Updates for Governor enhancements, CSR Service Layer Appendix | Frank Mayfield |
| 12/3/2013 | 5.0.0 | Documented send.metrics parameter | John McKinnon |
| 12/5/2014 | 5.1.1 | Documented custom error handling | John McKinnon |
| 07/27/2016 | 7.0.0 | AppCertRotation Common Endpoint and Java 1.7 upgrade from Framework 7.0.0 version | Raja Konkala |
| 8/19/2016 | 7.1.0 | REST Implementation | Derek Cochran |

Reference Documents

|  |
| --- |
| **Document** |
| [Development Framework Developers Guide](http://itg.prod.fedex.com/sf/go/doc325481?nav=1) |
| [Maven for Development Framework](http://itg.prod.fedex.com/sf/go/doc1242185?nav=1) |

Table of Contents

[1. CSR Overview 7](#_Toc459375867)

[2. What is a Web Service? 8](#_Toc459375868)

[2.1 Key Components of a Web Service 9](#_Toc459375869)

[2.2 Understanding Interactions between Components 10](#_Toc459375870)

[3. CSR Technology 12](#_Toc459375871)

[4. Standardization of Common Features in CSR 13](#_Toc459375872)

[5. Assumptions 13](#_Toc459375873)

[6. Application Setup for CSR 14](#_Toc459375874)

[6.1 Eclipse Project layout 15](#_Toc459375875)

[6.2 Define a contract (schema) 16](#_Toc459375876)

[6.3 Create objects from Schema using the jaxb compiler. 17](#_Toc459375877)

[6.4 Create POJO(s) 18](#_Toc459375878)

[7. Programmatic Versioning 30](#_Toc459375879)

[7.1 Overview of Programmatic Versioning 30](#_Toc459375880)

[7.2 Code Implementation and Spring wiring for Programmatic Versioning 31](#_Toc459375881)

[7.3 Build and deployment: 36](#_Toc459375882)

[8. XSLT Versioning 37](#_Toc459375883)

[8.1 XSLT for copying 37](#_Toc459375884)

[8.2 XSLT Request Conversion 38](#_Toc459375885)

[8.3 XSLT Response Conversion 39](#_Toc459375886)

[8.4 Configuration of operations.xml 40](#_Toc459375887)

[8.5 Build and deployment: 42](#_Toc459375888)

[8.6 Chained Converters 43](#_Toc459375889)

[8.7 Converter Plugin 44](#_Toc459375890)

[8.8 Chained Converter/Converter Plugin Setup 45](#_Toc459375891)

[9. Transaction Handlers 47](#_Toc459375892)

[9.1 Overview of Transaction Handlers 47](#_Toc459375893)

[9.2 Code Implementation/Spring Wiring for Transaction Handlers 48](#_Toc459375894)

[9.3 Using a Transaction Handler to Prevent Logging of Sensitive Data 49](#_Toc459375895)

[10. Document versus Object 52](#_Toc459375896)

[11. Custom Version Extractors 52](#_Toc459375897)

[12. Validation 55](#_Toc459375898)

[12.1 Default validation 56](#_Toc459375899)

[12.2 Programmatic validation: 57](#_Toc459375900)

[13. Security with CSR 60](#_Toc459375901)

[13.1 Security.xml 60](#_Toc459375902)

[13.2 The Judge 61](#_Toc459375903)

[13.3 Service Id (or Service Name) 62](#_Toc459375904)

[13.4 Service Action 62](#_Toc459375905)

[13.5 Configuration for an ESC Based Authorization Policy 62](#_Toc459375906)

[13.6 Configuration for a File Based Authorization Policy 64](#_Toc459375907)

[13.7 Judge/Jury Security setup in ESC 65](#_Toc459375908)

[14. Workflow Counters 67](#_Toc459375909)

[14.1 Operations.xml 67](#_Toc459375910)

[14.2 Workflow.properties 68](#_Toc459375911)

[14.3 SOAP Fault Exception 68](#_Toc459375912)

[14.4 Fault message sent to the client for Governor 69](#_Toc459375913)

[14.5 Fault message sent to the client for Workflow 69](#_Toc459375914)

[15. Governor and Workflow Enhancements 70](#_Toc459375915)

[15.1 Workflow Enhancements 70](#_Toc459375916)

[15.2 Governor Enhancements 71](#_Toc459375917)

[15.3 Governor Override via Plugin 72](#_Toc459375918)

[15.4 Comparison between Workflow and Governor 74](#_Toc459375919)

[16. Supported Operations 75](#_Toc459375920)

[17. Client Side Helper Classes 76](#_Toc459375921)

[17.1 JaxWS Implementation 76](#_Toc459375922)

[17.2 Spring REST Implementation 80](#_Toc459375923)

[18. Logger, TransID and Tracer Flag 82](#_Toc459375924)

[19. Overriding Request/Response Suffixes 86](#_Toc459375925)

[20. Error handling 87](#_Toc459375926)

[20.1 To produce a Soap Fault that includes a custom error message: 87](#_Toc459375927)

[20.2 Custom exception mapper 88](#_Toc459375928)

[21. Wily Monitoring Enablement 90](#_Toc459375929)

[21.1 Prerequisites 90](#_Toc459375930)

[21.2 Usage Details 90](#_Toc459375931)

[21.3 Technical Details 90](#_Toc459375932)

[22. CSR Fault Barrier 91](#_Toc459375933)

[23. Defining Log Level for CSR Client Exceptions (From Framework Release 5.0.0) 94](#_Toc459375934)

[24. Framework metrics 94](#_Toc459375935)

[25. REST Feature 95](#_Toc459375936)

[25.1 URL Structure 95](#_Toc459375937)

[25.2 Transactions 95](#_Toc459375938)

[25.3 Security 96](#_Toc459375939)

[25.4 Implementation 96](#_Toc459375940)

[25.5 WADL Support 97](#_Toc459375941)

[25.6 Enhance WADL Support 98](#_Toc459375942)

[25.7 XSD Support 101](#_Toc459375943)

[*25.8* Exception Handling 101](#_Toc459375944)

[25.9 SOAP UI Support 101](#_Toc459375945)

[25.10 Extendible 102](#_Toc459375946)

[26. Tomcat 7 Deployment 102](#_Toc459375947)

[Appendix A: FedEx Web service Prototype 103](#_Toc459375948)

[Designing the Server and Clients 105](#_Toc459375949)

[Transmission Primitives 105](#_Toc459375950)

[System Design 106](#_Toc459375951)

[Appendix B: CSR Service Layer 108](#_Toc459375952)

[Detailed Design: Spring Beans 108](#_Toc459375953)

[Appendix C: Glossary 111](#_Toc459375954)

# CSR Overview

The Common Service Runtime, a component of Development Framework, provides a Spring-based Web service environment that is reusable, extensible and based on industry standards. Common Service Runtime provides Web service components that are tested and certified so that developers can focus on solving business related problems. The benefits of development teams using CSR to create Web services include:

1. Increased consistency across services within the FedEx Enterprise.
2. Streamline the development of request-response style services.
3. Provide the ability isolate business logic from underlying technology.
4. Provide a common feature set for all CSR enabled services, such as security and versioning.
5. Ultimately, increase developer productivity.

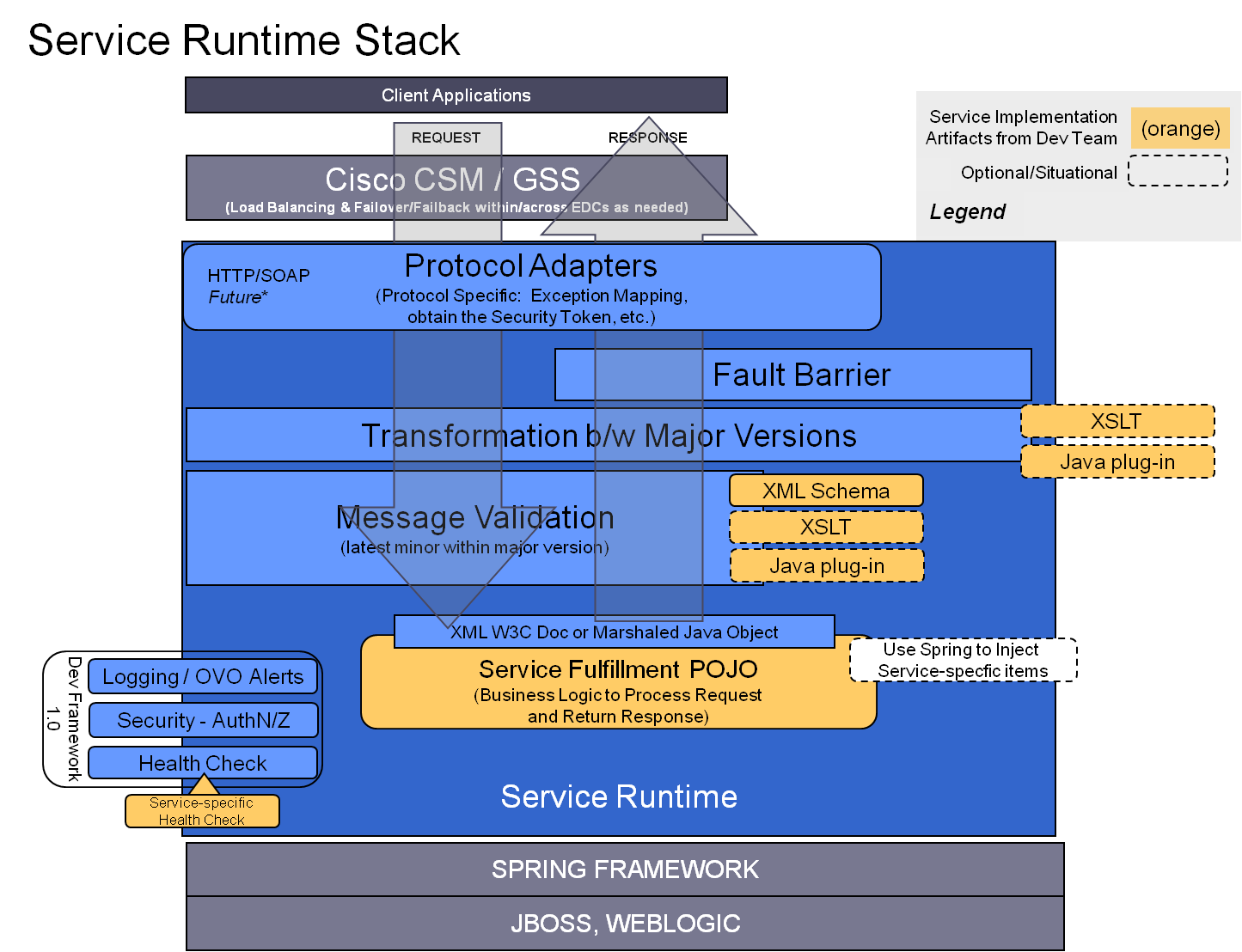


Figure . CSR Stack

# What is a Web Service?

A Web service is a software application that can be accessed remotely using different XML-based languages. Normally, a Web service is identified by a URL, just like any other Web site. What makes Web services different from ordinary Web sites is the type of interaction that they can provide. A Web service is similar in that it is accessed via a URL. The difference lies in the content of what is sent in the request from the client to the service. Web service clients send an XML document, formatted in a special way in accordance with the rules of the SOAP specification. A SOAP message can contain a call to a method along with any parameters that might be needed. In addition, the message can contain a number of header items that further specify the intent of the client. These header items might designate what Web services will get this method call after the current service finishes its work, or they might contain security information. In any case, the complexity of the SOAP message far exceeds the complexity that is possible using only a browser.

It is important to note that interoperability is one of the goals of Web services and some of the main advantages of Web services are:

* Web services are platform independent and it doesn’t matter what kind of computer sends the SOAP message or on what operating system it is running.
* Location of the client is not required.
* Client can be written in any programming language to communicate with the Web service.

## Key Components of a Web Service

Below are the core components of a Web service and everything that works together to make it a happen:

* **SOAP**

SOAP is a specification that defines an XML grammar for both sending messages and responding to messages that you receive from other parties. The goal of SOAP is to describe a message format that is not bound to any hardware or software architecture, but one that carries a message from any platform to any other platform in an unambiguous fashion. The SOAP standard contains two parts: the header that carries processing instructions and the body that contains the payload. The payload contains the information that you want to send. The two types of SOAP messages are documents and Remote Procedure Calls (RPCs). The payload of a document message is any XML document that you want to move from one computer to another. An RPC is a method call that is intended to be executed on the Web service’s computer. The RPC message performs the same function as an ordinary method call in an ordinary programming language. The difference is that this call can take place over the Internet.

* **Extensible Markup Language (XML)**

Extensible Markup Language (XML) is the language that all the Web services Languages are built on. XML is a tool for constructing self-describing documents. In fact, XML is more of a meta-language than a language and it is used to create grammars. These grammars are described in XML schemas that specify the tags that are allowed and the relationships between the elements defined by these tags. SOAP, WSDL, and UDDI are all XML-based grammars.

* **Hypertext Transport Protocol (HTTP)**

Hypertext Transport Protocol (HTTP) is a standard that was developed to facilitate the transfer of requests from a browser to a Web server. Web services takes advantage of the existence of this mature protocol to move SOAP messages and WSDL documents from one computer to another. SOAP can also use FTP, SMTP, JMS, etc. for the same purpose.

* **Web services Description Language (WSDL)**

Web services Description Language (WSDL) is a specification that tells us how to describe a piece of software in terms of the method calls that it responds to. These methods are described in an abstract way that is independent of what programming language the actual service is written in or what computer and operating system it runs on, communication between a PC and a mainframe computer for example. The WSDL also contains a concrete section in which the various details of how to actually make a connection (HTTP, FTP, JMS, or SMTP) to the service are stored.

## Understanding Interactions between Components

Sometimes it is hard to visualize the interaction between parts of a complex system like the Web service. For this reason, we are going to walk through a scenario to show how software written according to each of these standards interacts with the others in a typical transaction. To understand the interactions between all the different components in a Web service let us consider another example. Our hypothetical FedEx Web service begins its life as a COBOL program that accepts a file of addresses. This program was written in 1975 when FedEx was still in its infancy. This program basically compares the addresses against the official address database, corrects the address (if needed) and adds the last four digits of the ZIP Code (if missing).

Let us say that we are using Apache Axis as our SOAP engine for our Web service. We also have a special piece of Java software that accepts SOAP remote procedure calls as input and makes calls to the legacy Java system. In Axis jargon, this is a special type of handler called a dispatcher. First thing we need to do is to write the WSDL by hand to gain experience. This is not too hard because the service is very simple. We describe the method calls that our Web service will accept by following a combination of the WSDL and schema specifications. Next step is to publish information about the Service Provider to the Service Registry of our choice, the Microsoft public registry (which is written to the UDDI specification). There we enter in data about the Web site. Part of the information that we place on the site is the URL of the WSDL for the Web service. A potential customer performs a find operation on the Service Registry of his choice, the IBM public registry. This registry replicates the entries made on the Microsoft registry every night, so our Web service is listed in both registries. This replication is done according to the UDDI specification.

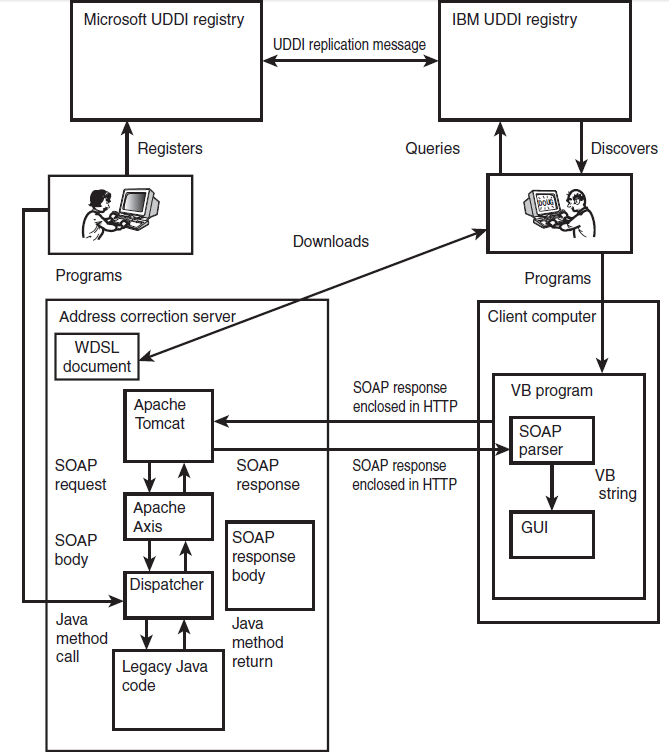
The potential customer finds the entry for our service and uses the listed URL to download a copy of the WSDL. Using the WSDL, he writes a Visual Basic program to serve as the Service Requestor to access the service. When the client-side programming is complete, he tests it by requesting that it perform a bind operation on the Web service. After the bind operation is successful, the client passes in a version of his home address that purposely contains a misspelling and a missing ZIP Code and waits for a response.

The client software packages his request into a SOAP envelope and sends it to the URL for the site using the HTTP. The JBoss Web server receives the HTTP message and strips off the HTTP headers. It passes the SOAP message to the Axis SOAP engine. The SOAP engine removes the header portion and processes any directives that might appear in the header.

The Axis SOAP engine calls the dispatcher program. The dispatcher calls the methods in the legacy system and passes the incorrect address to it as a parameter. The legacy address correction system fixes the address and adds the correct ZIP Code. It passes back the corrected address to the dispatcher. The SOAP engine creates the SOAP response message, adds any SOAP headers that are needed, and returns it to the JBoss Web server. JBoss adds the HTTP-specific data to the response and sends it back to the client.

The client program converts the SOAP message into a Visual Basic data type and returns it to the Visual Basic program. This program stores the response in a text box and displays it on the screen. The Visual Basic programmer looks at the corrected address as proof that everything worked properly. Figure given below shows this process graphically.

In essence, every Web service transaction follows this same general approach with the exception of the find, which is either done the first time or omitted altogether. Sometimes the message is forwarded on to the next Web service in a chain, but even then, the basic transactions are just being chained together.



# CSR Technology

CSR uses industry standards such as HTTP, SOAP, WSDL, and XML (including XSDs, XSLTs, and XPath) and is built on Java 1.6 and Spring 3.1 as discussed in chapter 1. In particular, XML provides a standard way to create a service contract, which defines the requests that the service will support, and the responses that will be returned for each request. Versioning strategies were analyzed and adopted from eBay and other companies. Lessons and building blocks were taken from work previously done on Common Data Service. Ideas and abstractions were incorporated from the existing FAST framework.

# Standardization of Common Features in CSR

CSR provides common features out of the box. Security is wrapped in a common way for both clients and services. Versioning strategies are well defined and helper interfaces and classes are provided. Boilerplate protocol specific setup has already been provided without the developer having to ‘figure out’ how something works. A standard fault bearing mechanism has also been provided. These common features only have to be tested once (both for features and for performance) and then can be used everywhere. This frees up developers to focus and spend more time on business requirements and other technical details.

# Assumptions

The basic requirements for creating a CSR-enabled application are listed below.

1. Java 1.6 (Java 1.7 is required from Framework 7.0.0 Version)
2. Weblogic 12.1.x or Tomcat 7.x installed locally for unit testing.
3. Application should be setup in Symphony/EAI and should have a valid Application ID.
4. A security certificate for the APP ID must be obtained via FedEx InfoSec’s certzilla website.
5. Development Framework 2.1.0 or later must be downloaded from FEDEXTRAS and installed locally.
6. Auto Server Build (ASB) will be used to build and deploy VM DEV/TEST/PROD environment.
7. Developer should have FedEx approved Eclipse from OSSM. Developer should be familiar with FedEx Eclipse IDE, ANT, and XML/XSLT/XPath.
8. Developer should be familiar with SOAP test tools (such as SOAP UI).
9. Developer must completely review the Development Framework User’s Guide and should familiarize with the operation of all the major components i.e. Security, Logging, Health Check, fp.properties (fingerprint file).
10. Developer should be able to create and deploy a Framework enabled project successfully.

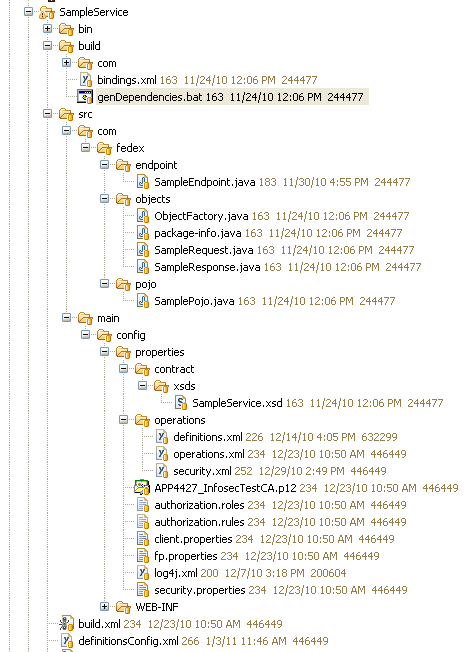
# Application Setup for CSR

The SampleService project, which is a CSR enabled Web service, can be downloaded from FedExtras. It is located in the “SamplesAndDemos” downloadable bundle. This bundle contains both a deployable WAR and a ZIP archive that includes source code, config files, etc. This section will walk through the development of a sample Web service in CSR, using SampleService as a reference.

One of the first things the developer will note is the project layout of the service. This layout is standard for all CSR enabled services, and allows the project to build and deploy easily. The CSR directory layout is shown below. Note that there are a few key files that are common to all CSR/Development Framework projects:

1. **main/config/properties/security.properties –** used by the CSR security implementation. This can be used “as-is” out of the box. If security performance needs to be optimized, there are some timer settings that can be adjusted.
2. **main/config/properties/log4j.xml –** configures the logging appenders for the FedExLogger, and is based on log4j API.
3. **main/config/properties fp.properties –** this is the fingerprint file for the application, and is required for any Framework-enabled project.
4. **main/config/properties client.properties –** contains information specific to the security clients PK12 certificate, including name and passphrase, and is used by the security API upon bootstrap.
5. **main/config/properties APP4227\_InfosecTestCA.p12 –** this is the certificate that is ordered from InfoSec certzilla web site, and is downloaded and bundled with the app.
6. **main/config/properties /operations/security.xml –** this contains the configuration for the security component of CSR – such as whether file-based or ESC-based policy will be used.
7. **main/config/properties/operations/definitions.xml –** this file defines all the beans with the WSDL details.
8. **main/config/properties/operations/operations.xml –** this contains all the bean information regarding operations supported by the service. It contains the bean that represents our service's schema. It is also in the definitions.xml file.
9. **main/config/properties/contract/SampleService.xsd –** this is the “contract” for other apps that wish to consume your service, and is part of the “contract-first” philosophy of CSR Web services.
10. **definitionsConfig.xml –** used by the BuildWsdl ANT task that is provided as part of the CSR development environment. This file contains the name of the location and name of the XSD for the service, which is used to build as WSDL at compile time.
11. **authorization.roles** and **authorization.rules –** these two files are used when a file-based authZ approach is used. However, this would result in the application being bounces when any authorization rules are changed. To avoid this, the authZ policy (collection of rules), along with roles, can be created and stored in the Enterprise Security Center. The security can then retrieve the policy dynamically on behalf of the app without a restart.

## Eclipse Project layout



## Define a contract (schema)

A developer will need to create an XML Schema (XSD) that represents the requests and responses that make up the service contract. The schema contains target namespace and one or more Request/Response objects. The root element name for Request object must contain the word Request in it. Example:

**<xs:element name="SampleRequest">**

The root element name for Response object must contain the word Response in it. Example:

**<xs:element name="SampleResponse">**

Let`s look at a SampleRequest object:

<xs:element name="SampleRequest">

<xs:complexType>

<xs:sequence>

<xs:element name="Attribute1" type="xs:string"></xs:element>

<xs:element name="Attribute2" type="xs:string"></xs:element>

</xs:sequence>

<xs:attributeGroup ref="versionAttributes"/>

</xs:complexType>

</xs:element>

The Request and Response object must contain the versionAttributes. The version attribute is important because as contract changes you can support multiple versions of the service without any impact to the existing clients. This versionAttribute would be defined using the:

**<xs:attributeGroup ref="versionAttributes"/>**

The versionAttributes is defined as:

<xs:attributeGroup name="versionAttributes">

<xs:attribute name="majorVersion" type="versionType" fixed="1" />

<xs:attribute name="minorVersion" type="versionType" />

</xs:attributeGroup>

Where versionType is a simpleType with a restriction:

<xs:simpleType name="versionType">

<xs:restriction base="xs:int">

<xs:minInclusive value="0"></xs:minInclusive>

</xs:restriction>

</xs:simpleType>

## Create objects from Schema using the jaxb compiler.

CSR has built-in support for Object to XML mapping (marshalling and unmarshalling). The developer can generate Java Objects from the service contract and then use those objects in the business logic instead of using an XML Document. The preferred OXM support is done through the standard Java 6 JAXB implementation. The marshalling and unmarshalling is performed by the framework and is abstracted away from the developer. The developer could only deal with Java Objects when writing the business code and not have any knowledge that XML is being used within the framework.

Using the jaxb compiler in your java installation invoke the xjc shell script in the bin directory for your platform and specify the path to your xsds.

*C:<Java\_Home>\bin\xjc.exe <directory where your xsds exist> -d <directory name for your generated files> –b <bindings file> -p <target package>*

**For Example:**

*C:\Java\jdk1.6.0\_21\bin\xjc.exe ..\src\main\config\properties\contract\xsds -d . -b bindings.xml -p com.fedex.objects*

Create a bindings.xml file and add it as one of the argument to xjc command. The bindings.xml file is as follows:

<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"

version="2.0"

xmlns:xjc= "http://java.sun.com/xml/ns/jaxb/xjc"

jaxb:extensionBindingPrefixes="xjc">

<jaxb:globalBindings fixedAttributeAsConstantProperty="true">

</jaxb:globalBindings>

</jaxb:bindings>

## Create POJO(s)

The next step is to create a sample Plain Old Java Object (POJO) This POJO will contain a method that accepts a request object that was created by running the xjc tool against the schema in /src/main/config/properties/contract/xsds directory. You can set the response object using the setter methods and return back your response object.

### Create End Points

Creating an end point is a little more involved and need to have all of the following:

1. The OperationEndpoint (package com.fedex.csr.common.operation) interface is the primary interface the developer will need to implement for their service. AbstractOperationEndpoint contains methods that would be common to all services and can be used if desired. The main method to focus on is the “execute” method.
2. Your end point needs to extend the AbstractOperationEndpoint and override the execute method. In the execute method cast the passed in parameter to your Request object and call the POJO process method. The return object for your POJO is the Response object so set the object using the context.
3. The OperationContext (package com.fedex.csr.common.operation) can only be accessed from the OperationEndpoint "execute" method and the Handler interface. It contains two MessageContexts, one which contains the request and another which contains the response. The developer uses these contexts to get and set requests and responses as appropriate. So, primarily these contexts keep the state of the current request and response.
4. Declare a private attribute that represents the pojo object declared earlier.
5. Have a setter method that spring will use to set our pojo object instance.
6. Within the CSR, this Object to XML Mapping is done on an as needed basis when calling MessageContext (refer to Section) getObject and getDocument methods. When calling getObject, if an object already exists, then the object will be returned, if only an XML Document exists, the request object will be unmarshaled into an object and the object will be returned. Similarly when calling getDocument, the XML Document will be returned if it exists, if only an object exists it will be marshaled to an XML Document, and the XML Document returned. So, the developer has an option to use XML Documents directly, but can use Java objects only when coding the service if that is preferred.

The following is an example of how to configure and create the most basic endpoint. Example Spring Configuration (operations.xml):

<bean id="sampleserviceendpoint"

class="com.fedex.endpoint.SampleEndpoint">

<property name="supportedOperations">

<list> <value>{http://sample.fedex.com/sampleservice}SampleRequest </value>

</list>

</property>

<property name="samplePojo" ref="sampleservice" />

</bean>

Here is an example of an implementation of the above endpoint.

**package** com.fedex.endpoint;

**import** com.fedex.csr.common.operation.AbstractOperationEndpoint;

**import** com.fedex.csr.common.operation.OperationContext;

**import** com.fedex.objects.SampleRequest;

**import** com.fedex.objects.SampleResponse;

**import** com.fedex.pojo.SamplePojo;

**public** **class** SampleEndpoint **extends** AbstractOperationEndpoint{

@Override

**public** **void** execute(OperationContext context) {

// Pull the samplerequest object from the context.

SampleRequest request =

(SampleRequest) context.getRequestContext().getObject();

// Call our POJO process method.

SampleResponse reply = samplePojo.ProcessRequest(request);

//

// set the pojo reply in the context.

context.getResponseContext().setObject(reply);

}

SampleResponse is a JAXB object that was generated from the service contract. Since the only Request mapped to this endpoint is a SampleRequest (which has no relevant content) and only one QName is mapped to this endpoint, there is no need to determine the type of Request or to do anything with the request. In this case pass the request to your ProcessRequest method in your pojo and set the return object. Your response object should be set it in the OperationContext. There is a 1-1 correlation between endpoints and POJOs.

### Multiple End Points

In order to configure a service that has multiple operations so that each request would map to an appropriate processor POJO, create all EndPoint beans and assign them to DefaultOperation’s property “endpoints” as a list. There is no need to create DefaultOperation beans for each EndPoints. Below you can see the modified operations.xml file for multiple EndPoints i.e.

<bean id="authenticationEndPoint" class="com.fedex.cpg.websvc.endpoint.PaymentServiceEndPoint">

<property name="supportedOperations">

<list>

<value>{http://xmlns.fedex.com/FDXREV.CPG}creditCardAuthenticationRequest</value>

</list>

</property>

<property name="paymentService" ref="paymentService" />

</bean>

<bean id="authorizationEndPoint" class="com.fedex.cpg.websvc.endpoint.PaymentServiceEndPoint">

<property name="supportedOperations">

<list>

<value>{http://xmlns.fedex.com/FDXREV.CPG}creditCardAuthorizationRequest</value>

</list>

</property>

<property name="paymentService" ref="paymentService" />

</bean>

<bean id="creditCardOperation" class="com.fedex.csr.common.operation.DefaultOperation">

<property name="requestValidationEnabled" value="false" />

<property name="responseValidationEnabled" value="false" />

<property name="supportedVersions">

<list>

<bean class="com.fedex.csr.common.operation.SupportedVersion">

<property name="majorVersion" value="1" />

</bean>

</list>

</property>

<property name="versionExtractor">

<bean class="com.fedex.cpg.websvc.utils.CPGVersionExtractor" />

</property>

<property name="endpoints">

<list>

<ref bean="authenticationEndPoint" />

<ref bean="authorizationEndPoint" />

</list>

</property>

</bean>

</beans>

### Spring Bean Wiring (operations & end points)

* **Service Definition**

As a service developer if you want to create WSDL from the XSD, you need to define definitionsConfig.xml and have the ANT task for building the WSDL. Below is an example ant task from Sample Service:

<taskdef name="buildwsdl" classname="com.fedex.csr.ant.BuildWsdl">

<classpath>

<fileset dir="${lib.framework.dir}" includes="\*.jar" />

<fileset dir="${lib.csr.dir}" includes="\*.jar" />

</classpath>

</taskdef>

There is no need to generate a WSDL separately. Service definition is defined in a file called definitionsConfig.xml and accomplished through the ServiceDefinition (package com.fedex.csr.ant.ServiceDefinition). The service must supply the port type name (interface name) for the service in addition to the XSDs that will form the service contract.

*Example Spring Configuration:*

<bean id="service.v1" class="com.fedex.csr.ant.ServiceDefinition">

<property name="supportCommonOperations" value="true" />

<property name="portTypeName" value="SampleService" />

<property name="xsds">

<list>

<value>file:src/main/config/properties/contract/xsds/SampleService.xsd</value>

</list>

</property>

<!-- <property name="targetNamespace" value="http://xmlns.fedex.com/csr/common"></property>

<property name="requestSuffix">

<list>

<value>input</value>

</list>

</property>

<property name="responseSuffix">

<list>

<value>output</value>

</list>

</property> -->

The name of the WSDL will be based on the Spring bean “id”, so in the example above the WSDL name would be service.v1.wsdl. The xsds property will include the list of xsds that should be used to build the WSDL. Setting the property name supportCommonOperations to true will add common.xsd to the classpath and will give the default operations ping and show operations.

* **Definitions.xml file**

In definitions.xml you define the same bean defined in the definitionsConfig.xml file and then set the wsdl property to the location of the WSDL.

<bean id="service.v1" class="org.springframework.ws.wsdl.wsdl11.SimpleWsdl11Definition">

<property name="wsdl" value="/WEB-INF/service.v1.wsdl" />

</bean>

It references the WSDLs, which were created in the build script and put in the wars WEB-INF directory.

* **Operations.xml file**

The Spring Bean configuration should go into a file called operations.xml in the operations directory refer to the project layout section in section [4.1](file:///C:\workspace\Framework\FrameworkRelease\internal-components\documents\CSR_Developers_Guide.docx). The Spring Beans in this file will be loaded one time at startup of the application.

<bean id="common.xsd.v1" class="org.springframework.core.io.ClassPathResource">

<constructor-arg value="common/contract/xsds/common\_v1.xsd" />

</bean>

<bean id="common.validator.v1" class="com.fedex.csr.common.contract.DefaultValidator">

<property name="majorVersion" value="1" />

<property name="minorVersion" value="1" />

<property name="xsd" ref="common.xsd.v1" />

</bean>

<bean id="common.mapper.v1" class="com.fedex.csr.common.oxm.JaxbMapper">

<property name="contextPath" value="com.fedex.xmlns.csr.common.v1" />

</bean>

<bean id="common.showOperationsEndpoint" class="com.fedex.csr.service.common.ShowOperationsEndpoint">

<!-- referenced from spring-runtime.xml -->

<property name="operationRegistry" ref="operationRegistry" />

<property name="supportedOperations">

<list>

<value>{http://xmlns.fedex.com/csr/common}showOperationsRequest</value>

</list>

</property>

</bean>

<bean id="common.pingEndpoint" class="com.fedex.csr.service.common.PingEndpoint">

<property name="supportedOperations">

<list>

<value>{http://xmlns.fedex.com/csr/common}pingRequest</value>

</list>

</property>

</bean>

<bean id=*"*common.appCertRotationEndpoint*"* class=*"*com.fedex.csr.service.common.AppCertRotationEndpoint*"*>

<property name=*"*supportedOperations*"*>

<list>

<value>

{http://xmlns.fedex.com/csr/common}appCertRotationRequest

</value>

</list>

</property>

</bean>

<bean id="common.operation" class="com.fedex.csr.common.operation.DefaultOperation">

<property name="supportedVersions">

<list>

<bean class="com.fedex.csr.common.operation.SupportedVersion">

<property name="validator" ref="common.validator.v1"/>

<property name="mapper" ref="common.mapper.v1" />

</bean>

</list>

</property>

<property name="handlerChain">

<list>

<!-- <ref bean="common.transactionlogging" /> -->

</list>

</property>

<property name="endpoints">

<list>

<ref bean="common.showOperationsEndpoint" />

<ref bean="common.pingEndpoint" />

<ref bean="common.appCertRotationEndpoint" />

</list>

</property>

</bean>

</beans>

* **Service Schema**

*Example Spring Configuration:*

<!--this is the bean that represents our service's schema. -->

<bean id="SampleService.v1" class= "org.springframework.core.io.ClassPathResource">

"<constructor-arg value ="contract/xsds/SampleService.xsd"/>

</bean>

* **Mapper**

Low level marshalling and unmarshalling (object to XML mapping) is handled by a Mapper. There is one implementation of Mapper, JaxbMapper (package com.fedex.csr.common.oxm), which marshals and unmarshals using standard Java JAXB. The example spring configuration shown below is part of operations.xml.

*Example Spring Configuration:*

The contextPath is the package name for JAXB generated objects. This is a mapper bean that maps xml to compatible java objects. In this case, the mapper's ID is mapper.v1 and whenever the mapper is used, it will create objects with the package name specified in your xjc command.

<bean id="mapper.v1" class="com.fedex.csr.common.oxm.JaxbMapper">

<property name="contextPath" value="com.fedex.objects" />

</bean>

* **Validator**

Validators are used to validate requests and responses. The easiest way to accomplish XSD validation is using a Validator (package com.fedex.csr.common.contract). This class takes a XSD and version information as arguments. An optional argument can be a XSLT, where this XSLT will then be used for additional validation that could not be performed by a XSD. . The example spring configuration shown below is part of operations.xml.

*Example Spring Configuration:*

<bean id="validator.v1" class="com.fedex.csr.common.contract.DefaultValidator">

<property name="majorVersion" value="1" />

<property name="minorVersion" value="0" />

<property name="xsd" ref="SampleService.v1" /> </bean>

* **POJO Declaration**

The example spring configuration shown below is part of operations.xml.

*Example Spring Configuration:*

This is the bean that represents your POJO object.

<bean id="sampleservice" class="com.fedex.pojo.SamplePojo"/>

* **End point Configuration**

The following is an example of how to configure and create the most basic endpoint. The example spring configuration shown below is part of operations.xml.

*Example Spring Configuration:*

<bean id="sampleserviceendpoint"

class="com.fedex.endpoint.SampleEndpoint">

<property name="supportedOperations">

<list> <value>{http://sample.fedex.com/sampleservice}SampleRequest </value>

</list>

</property>

<property name="samplePojo" ref="sampleservice" />

</bean>

**Notes:**

The value element in the supported operations list must match the target namespace defined in the xsd file for your project. In this case, the xsd file contains xmlns="http://sample.fedex.com/sampleservice". Next, you add the request object that the endpoint should handle. In this case we have only one request, named SampleRequest.

The samplePojo property represents the 'set' method in our pojo. You can see that we are using the bean that represents our pojo declared earlier to set the pojo object in our endpoint class definition above.

The Operation, specifically the DefaultOperation (package com.fedex.csr.common.operation) is the component that calls and uses the other components. It is the “glue” component and directs what components get called and in what order they are called.

SupportedVersion holds objects and information related to a specific major version. OperationEndpoint will contain the business implementation.

<bean id="sampleOperation" class="com.fedex.csr.common.operation.DefaultOperation">

<property name="supportedVersions">

<list>

<bean class="com.fedex.csr.common.operation.SupportedVersion">

<property name="validator" ref="validator.v1"/>

<property name="mapper" ref="mapper.v1" />

</bean>

</list>

</property>

<property name="endpoints">

<list>

<ref bean="sampleserviceendpoint" />

</list>

</property>

</bean>

**Notes:**

Here, we are creating a CSR operation. In this example, we are saying the operation should handle requests that have a version of '1' and the objects should be created using the mapper bean created above, which specifies the package name.

* **Security file**

Refer to the [Section 13](#_Security_with_CSR) in this document.

### Create the war file and deploy

When you create the war file you need to have this basic structure of having WEB-INF as the root folder.

* 1. WEB-INF/classes contains the classes folder with your package inside it.
  2. WEB-INF/classes/contracts/xsds. This is where your schema needs to reside.
  3. WEB-INF/classes/operations. This is where your definitions.xml and operations.xml file need to exist.
  4. WEB-INF . This is where your WSDL needs to exist. This is the WSDL that you built at build time.

After your war is deployed in JBoss/weblogic you can hit the service by going to the URL:

*http://servername:port/<context-root>/<service definition>.wsdl*

For example: http://localhost:8080/sampleservice/service.v1.wsdl. In this case I did not set my context root and that is why it is the name of my war file. The context root is set in weblogic.xml for weblogic and jboss-web.xml for JBoss server.

A SampleService.zip file is available from FedExtras (keyword: “fedextras”). You can unzip this folder and import it into your Eclipse workspace. You can build the project using the Ant script provided and this should generate your WSDL in target/WEB-INF directory and a war file under the target directory.

### Build.xml File

*The directory structure for a build file is:*

src/main/config/properties - Spring configuration files for the war

src/main/config/properties/contract/xsds - Schema files

C:\cs-framework\X.x.x\lib\CommonServiceRuntime.jar contains common\_v1.xsd for the common operations (i.e. ping, showOperations and from framework 7.0.0 appCertRotation operation for on-demand certificate rotation for service)

src/main/config/properties/operations

*Spring configuration files used for wiring should contain:*

1. operations.xml
2. definitions.xml
3. security.xml

src/main/resources/WEB-INF

web.xml and application server specific descriptors.

In order to generate the WSDL at build time your build script needs to have the ant task <buildwsdl> along with the required jars in the classpath. The ant task <buildwsdl> looks for definitionsConfig.xml at the root of this ant project

<!—- define the task here and reference framework libs -->

<taskdef name=*"buildwsdl"* classname=*"com.fedex.csr.ant.BuildWsdl"*>

<classpath>

<fileset dir=*"${lib.framework.dir}"* includes=*"\*.jar"* />

<fileset dir=*"${lib.csr.dir}"* includes=*"\*.jar"* />

</classpath>

</taskdef>

<!-- output to target WEB-INF directory -->

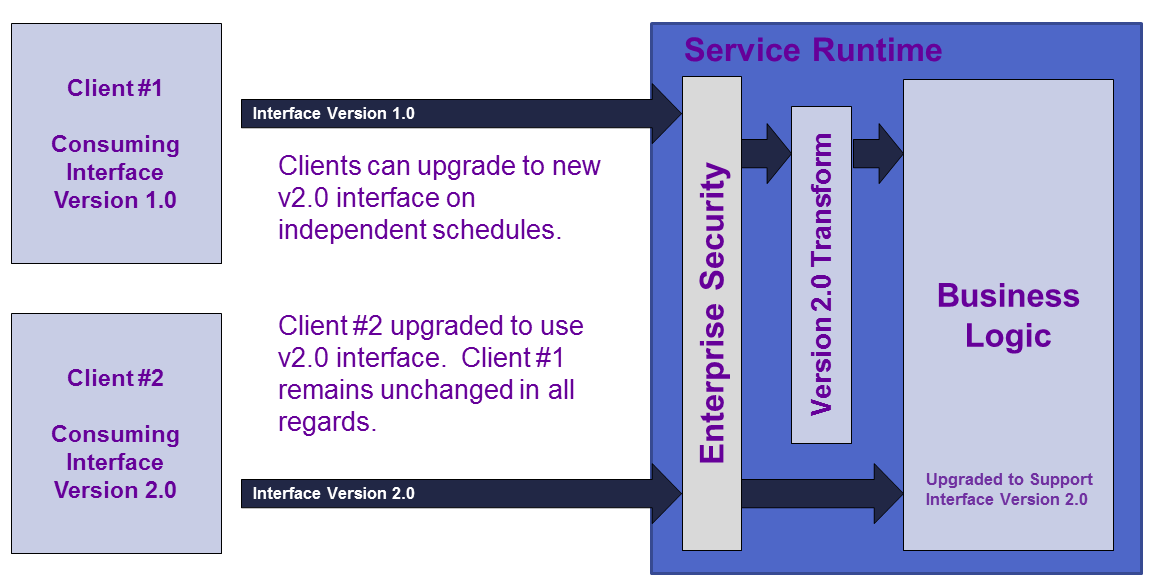
<!—- default definitions file is definitionsConfig.xml at the root of this ant project -->

<buildwsdl wsdlOutputDirectory=*"${target.webinf.dir}/"* />

# Programmatic Versioning

## Overview of Programmatic Versioning

Programmatic Versioning supports multiple versions of a request to the service. This is useful when the version of the service contract is changed but still it should not impact any of the clients that are using the service. The programmatic versioning enables the service to support both the clients that uses new changes as well as the clients that are still using the older version of the service contract. For example when the service is changed from version 1 that takes two attributes in a request, to version 2 that takes three attributes in a request then the service should support both version 1 with two attributes and version 2 with three attributes of the requests that come in from the clients.



To implement programmatic versioning, the following steps should be completed:

1. **Schema/contract –** Make necessary changes to the service contract.
2. **Objects –** Generate the objects based on the new service contract.
3. **POJOs –** Make necessary changes to POJOs to process the new request.
4. **Convert method –** Convert method is implemented /overridden in the sample endpoint class to convert the request from older version to newer version.
5. **Operations.xml –** Add a bean that represents the service’s schema, a mapper and a validator for each supported version.
6. **Definitions.xml –** Include the beans that are added in the operations.xml representing the service’s schema.
7. **definitionsConfig**.**xml –** Include the beans that are defined in the definitions.xml referring to the xsds location, so that the Wsdls are created and put under the WEB-INF directory.

## Code Implementation and Spring wiring for Programmatic Versioning

Please refer to [section 5](#_Sample_Application_Setup) for setting up the Sample Service before proceeding further as the example shown below is totally based on the Sample Service version 1. The structure/framework and files mentioned in the following example is same as that are used in Sample Service.

**Example:** Version 1 of the Sample Service takes two attributes in the request and sends two attributes in the response. Version 2 of the Sample Service takes one extra attribute that is three attributes in the request and sends three attributes in the response.

**Setting up of the Sample Service version 2 is shown below.**

### Schema/contract:

Refer to section [6.1.1](#_Define_a_contract) for the version 1 of the service contract. For Sample Service version 2, new request attribute and response attributes are added in the contract.

<xs:attributeGroup name=*"versionAttributes"*>

<xs:attribute name=*"majorVersion"* type=*"versionType"*

fixed=*"2"* />

<xs:attribute name=*"minorVersion"* type=*"versionType"* />

</xs:attributeGroup>

<xs:element name=*"SampleRequest"*>

<xs:complexType>

<xs:sequence>

<xs:element name=*"Attribute1"* type=*"xs:string"*></xs:element>

<xs:element name=*"Attribute2"* type=*"xs:string"*></xs:element>

// New Request attribute is added as a change from Version 1

<xs:element name=*"Attribute3"* type=*"xs:string"*></xs:element>

</xs:sequence>

<xs:attributeGroup ref=*"versionAttributes"*/>

</xs:complexType>

</xs:element>

<xs:element name=*"SampleResponse"*>

<xs:complexType>

<xs:sequence>

<xs:element name=*"ResponseAttribute1"* type=*"xs:string"*></xs:element>

<xs:element name=*"ResponseAttribute2"* type=*"xs:string"*></xs:element>

// New Response attribute is added as a change from Version 1

<xs:element name=*"ResponseAttribute3"* type=*"xs:string"*></xs:element>

</xs:sequence>

<xs:attributeGroup ref=*"versionAttributes"*/>

</xs:complexType>

</xs:element>

### Objects:

Generate the objects from the new schema/service contract with three attributes using jaxb compiler as documented in [Section 6.1.2](#_Create_objects_from).

### POJO:

Change the POJO to process the new request with three attributes and send the response with three attributes.

public SampleResponse ProcessRequest(SampleRequest request){

SampleResponse response = new SampleResponse();

response.setResponseAttribute1("Response Data1");

response.setResponseAttribute2("Response Data2");

//Add new response attribute to process the request with version 2 changes

response.setResponseAttribute3("Response Data3");

return response;}

### Convert Method:

The convert method must be overridden in the SampleEndPoint class in order to support the versioning. This method automatically gets invoked by the CSR Framework when a version 1 request comes in. This method copies the two attributes from version1 request and adds an extra third attribute to comply with version 2 of the Sample Service.

Here is the sample code to implement the convert method.

**public** Object convert(Object object, **int** fromVersion, **int** toVersion) {

**if** (object **instanceof** com.fedex.objects.v1.SampleRequest) {

com.fedex.objects.v1.SampleRequest sampleRequest = (com.fedex.objects.v1.SampleRequest) object;

com.fedex.objects.v2.SampleRequest returnRequest = **new** com.fedex.objects.v2.SampleRequest();

returnRequest.setAttribute1(sampleRequest.getAttribute1());

returnRequest.setAttribute2(sampleRequest.getAttribute2());

returnRequest.setAttribute3("Default Data");

**return** returnRequest;

}

**if** (object **instanceof** com.fedex.objects.v2.SampleResponse) {

com.fedex.objects.v2.SampleResponse sampleReply = (com.fedex.objects.v2.SampleResponse) object;

com.fedex.objects.v1.SampleResponse returnResponse = **new** com.fedex.objects.v1.SampleResponse();

returnResponse.setResponseAttribute1(sampleReply.getResponseAttribute1());

returnResponse.setResponseAttribute2(sampleReply.getResponseAttribute2());

**return** returnResponse;

}

**throw** **new** RuntimeException("no transform available for version "

+ fromVersion + " to version " + toVersion);

}

### Operations.xml

Spring wiring to set up Sample Service Version2:

**NOTE:** If your service is running only the latest version, you just need to have the latest XSD as a class path resource. And if the service is running in multiple versions, then they need to have all the XSD versions as a class path resource.

#### Service Schema Declaration:

<bean id="SampleService.v1" class="org.springframework.core.io.ClassPathResource">

<constructor-arg value="contract/xsds/SampleService.xsd" />

</bean>

//Add schema declaration for version 2 of the contract

<bean id="SampleService.v2" class="org.springframework.core.io.ClassPathResource">

<constructor-arg value="contract/xsds/SampleService.v2.xsd" />

</bean>

#### Mapper:

<bean id="mapper.v1" class="com.fedex.csr.common.oxm.JaxbMapper">

<property name="contextPath" value="com.fedex.objects.v1" />

</bean>

//Add new mapper for version 2 of service contract

<bean id="mapper.v2" class="com.fedex.csr.common.oxm.JaxbMapper">

<property name="contextPath" value="com.fedex.objects.v2" />

</bean>

#### Validator:

<bean id="validator.v1" class="com.fedex.csr.common.contract.DefaultValidator">

<property name="majorVersion" value="1" />

<property name="minorVersion" value="0" />

<property name="xsd" ref="SampleService.v1" />

</bean>

//Add new validator for version 2 of service contract

<bean id="validator.v2" class="com.fedex.csr.common.contract.DefaultValidator">

<property name="majorVersion" value="2" />

<property name="minorVersion" value="0" />

<property name="xsd" ref="SampleService.v2" />

</bean>

#### Defining Operation:

<list>

<bean class=*"com.fedex.csr.common.operation.SupportedVersion"*>

<property name=*"validator"* ref=*"validator.v1"*/>

<property name=*"mapper"* ref=*"mapper.v1"* />

</bean>

//Add a default operation for version 2

<bean class=*"com.fedex.csr.common.operation.SupportedVersion"*>

<property name=*"validator"* ref=*"validator.v2"*/>

<property name=*"mapper"* ref=*"mapper.v2"* />

</bean>

</list>

### Definitions.xml

Refer to [section 6.1.5](#_Definitions_file)  for more information about definitions file. In definitions.xml refer to the same bean defined in the operations.xml file and then set the wsdl property to the location of the WSDL version2.

<bean id="service.v1" class="org.springframework.ws.wsdl.wsdl11.SimpleWsdl11Definition">

<property name="wsdl" value="/WEB-INF/service.v1.wsdl" />

</bean>

// Add new bean id to set the new WSDL location

<bean id="service.v2" class="org.springframework.ws.wsdl.wsdl11.SimpleWsdl11Definition">

<property name="wsdl" value="/WEB-INF/service.v2.wsdl" />

</bean>

### DefinitionsConfig.xml

Include the beans that are defined in the definitions.xml referring the location of the xsds. Refer to section [6.4.2](#_Service_Definition) for more information on definitionsConfig file.

<bean id="service.v1" class="com.fedex.csr.ant.ServiceDefinition">

<property name="supportCommonOperations" value="true" />

<property name="portTypeName" value="SampleService" />

<property name="xsds">

<list>

<value>file:src/main/config/properties/contract/xsds/SampleService.v1.xsd</value>

</list>

</property>

</bean>

<bean id="service.v2" class="com.fedex.csr.ant.ServiceDefinition">

<property name="supportCommonOperations" value="true" />

<property name="portTypeName" value="SampleService" />

<property name="xsds">

<list>

<value>file:src/main/config/properties/contract/xsds/SampleService.v2.xsd</value>

</list>

</property>

</bean>

## Build and deployment:

Refer to the Sections [6.1.6](#_Create_the_war) and [6.1.7](#_Build_File).

# XSLT Versioning

The following section is based on the SampleService in [section](#_Toc260818816) 6; refer to it for initial setup of version 1 of SampleService (which contains 2 request attributes). Before proceeding further, refer to [section 7.2](#_Code_Implementation_and) to use the version 2 of the SampleService.xsd (which contains 3 request attributes). When implementing XSLT versioning, the convert() method does not need to be defined in the endpoint. To implement XSLT versioning, the following steps should be completed:

1. **Schema/contract –** Make necessary changes to the service contract refer to [section 7.2.1](#_Schema/contract:).
2. **Objects –** Generate the objects based on the new service contract refer to [section 7.2.2](#_Objects:).
3. **POJOs –** Make necessary changes to POJOs to process the new request refer to [section 7.2.3](#_POJO:).
4. **operations.xml –** Add a bean that represents the service’s schema, a mapper and a validator for each supported version refer to [section 7.2.5](#_Operations.xml). Wiring of the converters into the operations.xml is shown in [section 7.2.5](#_Operations.xml).
5. **definitions.xml –** Include the beans that are added in the operations.xml representing the service’s schema refer to [section 7.2.6](#_Definitions.xml).
6. **definitionsConfig**.xml **–** Include the beans that are defined in the definitions.xml referring to the xsds location, so that the Wsdls are created and put under the WEB-INF directory.

**NOTE: The Objects of older versions are not required.**

## XSLT for copying

Transforming XML to XML, which is from one version to another, can be achieved by coping technique or also known as identity transform. In this technique, a stylesheet is used to copy an input document to an output document without changing it. This copy.xslt can be imported into other stylesheets to do the transformations which reduces coding effort.

Following is the stylesheet copy.xslt that can be stripped out and used/imported into other stylesheets.

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

**<xsl:stylesheet** **version**=*"2.0"* **xmlns:xsl**=*"http://www.w3.org/1999/XSL/Transform"***>**

**<xsl:template** **match**=*"node() | @\*"***>**

**<xsl:copy>**

**<xsl:apply-templates** **select**=*"@\* | node()"***/>**

**</xsl:copy>**

**</xsl:template>**

**</xsl:stylesheet>**

## XSLT Request Conversion

A stylesheet is required to convert request from one version to another. In the example that we follow, if the version1 client sends a request with two attributes to the version2 SampleService that takes/requires three attributes, then it should be transformed to version2 request with three attributes in order to process the request. Following is the example of converting version1 request to version2:

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

**<xsl:stylesheet** **version**=*"2.0"*

**xmlns:xsl**=*"http://www.w3.org/1999/XSL/Transform"* **xmlns**=*"http://sample.fedex.com/sampleservice"*

**xmlns:tns**=*"http://sample.fedex.com/sampleservice"* **xmlns:xsi**=*"http://www.w3.org/2001/XMLSchema-instance"***>**

**<xsl:import** **href**=*"copy.xslt"***/>**

**<xsl:output** **method**=*"xml"* **version**=*"1.0"* **encoding**=*"UTF-8"*

**indent**=*"yes"* **/>**

**<xsl:template** **match**=*"@xsi:schemaLocation"***>**

<!-- <xsl:attribute

name="xsi:schemaLocation">[http://xmlns.fedex.com/example</xsl:attribute](http://xmlns.fedex.com/example%3c/xsl:attribute)> -->

**</xsl:template>**

**<xsl:template** **match**=*"@majorVersion"***>**

**<xsl:attribute** **name**=*"majorVersion"***><xsl:value-of** **select**=*"2"* **/></xsl:attribute>**

**</xsl:template>**

**<xsl:template** **match**=*"@minorVersion"***>**

**<xsl:attribute** **name**=*"minorVersion"***><xsl:value-of** **select**=*"0"* **/></xsl:attribute>**

**</xsl:template>**

**<xsl:template** **match**=*"tns:SampleRequest/tns:Attribute2"***>**

**<xsl:copy-of** **select**=*"."* **/>**

**<xsl:element** **name**=*"Attribute3"***>**Default Data**</xsl:element>**

**</xsl:template>**

**</xsl:stylesheet>**

## XSLT Response Conversion

Another stylesheet is required to convert response from one version to another. In the example that we follow, if the version2 SampleService returns a response with three attributes to a v1 client that can accept only two attributes then the response has to be transformed to version1 with two attributes in order to support v1 clients. Following is the example of converting version2 response to version1:

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

**<xsl:stylesheet** **version**=*"2.0"* **xmlns:xsl**=*"http://www.w3.org/1999/XSL/Transform"*

**xmlns**=*"http://sample.fedex.com/sampleservice"* **xmlns:tns**=*"http://sample.fedex.com/sampleservice"*

**xmlns:xsi**=*"http://www.w3.org/2001/XMLSchema-instance"***>**

**<xsl:import** **href**=*"copy.xslt"***/>**

**<xsl:output** **method**=*"xml"* **version**=*"1.0"* **encoding**=*"UTF-8"* **indent**=*"yes"***/>**

**<xsl:template** **match**=*"@xsi:schemaLocation"***>**

<!--

<xsl:attribute

name="xsi:schemaLocation"><http://xmlns.fedex.com/example>

</xsl:attribute>

-->

**</xsl:template>**

**<xsl:template** **match**=*"@majorVersion"***>**

**<xsl:attribute** **name**=*"majorVersion"***>**

**<xsl:value-of** **select**=*"1"***/></xsl:attribute>**

**</xsl:template>**

**<xsl:template** **match**=*"@minorVersion"***>**

**<xsl:attribute** **name**=*"minorVersion"***>**

**<xsl:value-of** **select**=*"0"***/></xsl:attribute>**

**</xsl:template>**

**<xsl:template**

**match**=*"tns:SampleResponse/tns:ResponseAttribute3"***>**

**</xsl:template>**

**</xsl:stylesheet>**

## Configuration of operations.xml

Declare the beans for each of the stylesheet, one for request and one for response.

<bean id="SampleService\_Converter-1-to-2" class="com.fedex.csr.common.contract.DefaultConverter">

<property name="fromMajorVersion" value="1" />

<property name="toMajorVersion" value="2" />

<property name="xslt">

<value>classpath:contract/xslts/SampleService-MajorVersion-1-to-2.xslt</value>

</property>

</bean>

<bean id="SampleService\_Converter-2-to-1" class="com.fedex.csr.common.contract.DefaultConverter">

<property name="fromMajorVersion" value="2" />

<property name="toMajorVersion" value="1" />

<property name="xslt">

<value>classpath:contract/xslts/SampleService-MajorVersion-2-to-1.xslt</value>

</property>

</bean>

**Converters:**

Refer to the beans that are declared for the request/response stylesheets.

<property name=*"converters"*>

<list>

<ref bean=*"SampleService\_Converter-1-to-2"* />

<ref bean=*"SampleService\_Converter-2-to-1"* />

</list>

</property>

<property name=*"endpoints"*>

<list>

<ref bean=*"sampleserviceendpoint"* />

</list>

</property>

### Definitions.xml

Refer to [section 6.1.5](#_Definitions_file)  for more information about definitions file. In definitions.xml refer to the same bean defined in the operations.xml file and then set the wsdl property to the location of the WSDL version2.

<bean id="service.v1" class="org.springframework.ws.wsdl.wsdl11.SimpleWsdl11Definition">

<property name="wsdl" value="/WEB-INF/service.v1.wsdl" />

</bean>

// Add new bean id to set the new WSDL location

<bean id="service.v2" class="org.springframework.ws.wsdl.wsdl11.SimpleWsdl11Definition">

<property name="wsdl" value="/WEB-INF/service.v2.wsdl" />

</bean>

### DefinitionsConfig.xml

Include the beans that are defined in the definitions.xml referring the location of the xsds. Refer to [section 6.4.2](#_Service_Definition) for more information on definitionsConfig file.

<bean id="service.v1" class="com.fedex.csr.ant.ServiceDefinition">

<property name="supportCommonOperations" value="true" />

<property name="portTypeName" value="SampleService" />

<property name="xsds">

<list>

<value>file:src/main/config/properties/contract/xsds/SampleService.v1.xsd</value>

</list>

</property>

</bean>

<bean id="service.v2" class="com.fedex.csr.ant.ServiceDefinition">

<property name="supportCommonOperations" value="true" />

<property name="portTypeName" value="SampleService" />

<property name="xsds">

<list>

<value>file:src/main/config/properties/contract/xsds/SampleService.v2.xsd</value>

</list>

</property> </bean>

## Build and deployment:

Refer to the Sections [6.1.6](#_Create_the_war) and [6.1.7](#_Build_File).

## Chained Converters

CSR Version 3.2 contains a new class for up/down conversion of requests and responses: **com.fedex.csr.common.contract.ChainedConverter**. The ChainedConverter can be defined in the operations.xml file of the webservice, and can be used when a series of stylesheets are defined for the up-convert and down-convert path. As its name implies, a series of **com.fedex.csr.common.contract.DefaultConverter** beans can be defined and used via the Spring configuration. This gives the developer additional flexibility to define which stylesheets should be invoked on up-convert/down-convert, based on the version of the incoming request. Also, as new versions of the service are introduced, the existing stylesheets that have been tested do not have to be changed. Only the newest stylesheets would need to be created and added into the conversion chain. This methodology is useful for legacy services that must support multiple levels of client requests.

Chained Versioning Process

INPUT=Track Request v4 (in w3c DOM form)

Up convert from v4 to v5  
(xslt: operates on w3c DOM)

Up convert from v6 to v7  
(xslt)

PreUpConvert  
(via CSR Plugin)

CSR “Endpoint”

3. Use setObject() to set ientities java object (mapper config to marshal back to w3c dom)

Track Request = completed v7 upconvert(in w3c DOM form)

PostUpConvert  
(via CSR Plugin)

EJB RMI Track Service v6

Down convert from v7 to v6  
(xslt)

Pre DownConvert  
(via CSR Plugin)

CSR “Endpoint”

1. Convert v7 Request to EC DOM  
   ientities **version 6** Java object
2. Invoke FAST Service via RMI,  
   get ientities **version 6** response object

OUTPUT: v4 Track Response (w3cDOM form)

Down convert from v6 to v5  
(xslt)

Down convert from v5 to v4  
(xslt operates on w3c DOM)

PostDownConvert.  
(via CSR Plugin)

## Converter Plugin

An optional feature of the ChainedConverter is the ability to implement the com.fedex.csr.common.contract.ConverterPlugin interface. This interface can be used if additional processing is needed on requests/responses before and after the up/down-conversion chain is executed. For example, incoming requests may need to be normalized to remove version numbers. This could be implemented in the preUpConvert() method of the ConverterPlugin. The following methods are provided as part of the interface:

* 1. **Document preUpConvert(OperationContext, Document)**: called before the up-convert chain is executed by the ChainedConverter.
  2. **Document postUpConvert(OperationContext, Document):** called after the up-convert chain is completed by the ChainedConverter.
  3. **Document preDownConvert(OperationContext, Document):** called before the down-convert chain is executed by the ChainedConverter.
  4. **Document postDownConvert(OperationContext, Document):** called after the down-convert chain is completed by the ChainedConverter.
  5. **public Map<String, String> getRequestStringParams(OperationContext context, Document document):** allows developer to build a Map of String params from the request and pass it to the stylesheet for up-converts.
  6. **public Map<String, Node> getRequestNodeParams(OperationContext context, Document document);** allows developer to build a Map of Node params from the request and pass it to the stylesheet for up-converts.
  7. **public Map<String, String> getResponseStringParams(OperationContext context, Document document);** allows developer to build a Map of String params from the response and pass it to the stylesheet for down-converts.
  8. **public Map<String, Node> getResponseNodeParams(OperationContext context, Document document);** allows developer to build a Map of String params from the response and pass it to the stylesheet for down-converts.
  9. **public void logConvertedDocument(OperationContext context, Document document, int fromMajorVersion, int toMajorVersion, boolean isRequest);** provides a helper method to log the converted request/response document.

If not all of the pre/post methods are needed, helper methods are provided to turn off via a Boolean (returning false ensures the method is not called):

* 1. **public boolean isPostDownConvertEnabled()**
  2. **public boolean isPostUpConvertEnabled()**
  3. **public boolean isPreUpConvertEnabled()**
  4. **public boolean isPreUpConvertEnabled()**

## Chained Converter/Converter Plugin Setup

To implement a ChainedConverter class, you will still define the converter beans in operations.xml. Each DefaultConverter bean references a stylesheet for a particular up/down convert:

<bean id=*"SampleService\_Converter-1-to-2"* class=*"com.fedex.csr.common.contract.DefaultConverter"*>

<property name=*"fromMajorVersion"* value=*"1"* />

<property name=*"toMajorVersion"* value=*"2"* />

<property name=*"xslt"*>

<value>classpath:contract/xslts/SampleService-MajorVersion-1-to-2.xslt

</value>

</property>

</bean>

In addition, a ChainedConverter bean should be defined, which will reference the list of up and down converter beans:

<!-- NEW for chained converters -->

<bean id=*"SampleService\_ChainedConverter"* class=*"com.fedex.csr.common.contract.ChainedConverter"*>

<property name=*"latestSupportedVersion"* value=*"3"*/>

<property name=*"timingsInfoEnabled"* value=*"true"*/>

<property name=*"upConverters"*>

<list>

<ref bean=*"SampleService\_Converter-1-to-2"* />

<ref bean=*"SampleService\_Converter-2-to-3"* />

</list>

</property>

<property name=*"downConverters"*>

<list>

<ref bean=*"SampleService\_Converter-3-to-2"* />

<ref bean=*"SampleService\_Converter-2-to-1"* />

</list>

</property>

<!-- an optional converter plugin -->

<property name=*"converterPlugin"* ref=*"SampleServiceConverterPlugin"* />

</bean>

If the converterPlugin bean was created it is referenced as well. It implements the ConverterPlugin interface.

Finally, the DefaultOperation bean implments the new ChainedConverter bean that was created, and has its corresponding property set to true:

<bean id=*"sampleOperation"* class=*"com.fedex.csr.common.operation.DefaultOperation"*>

<property name=*"supportedVersions"*>

<list>

<bean class=*"com.fedex.csr.common.operation.SupportedVersion"*>

<property name=*"validator"* ref=*"validator.v1"*/>

</bean>

<bean class=*"com.fedex.csr.common.operation.SupportedVersion"*>

<property name=*"validator"* ref=*"validator.v2"*/>

</bean>

<bean class=*"com.fedex.csr.common.operation.SupportedVersion"*>

<property name=*"validator"* ref=*"validator.v3"*/>

<property name=*"mapper"* ref=*"mapper.v3"* />

</bean>

</list>

</property>

<!-- NEW for chained converters -->

<property name=*"chainedConverterEnabled"* value=*"true"*></property>

<property name=*"converters"*>

<list>

<ref bean=*"SampleService\_ChainedConverter"* />

</list>

</property>

.

.

.

</bean>

Note that the JAX-B mapper is only needed for the version that the service is running, which in this case is mapper.v3.

Also, if needed, there is a method to access the original request that was sent in to the service, available from the OperationContext object:

SampleRequest origRequest = (SampleRequest) context.getOriginalRequest();

This can be useful for reference when undergoing a multi-step versioning transformation.

# Transaction Handlers

## Overview of Transaction Handlers

Handlers are used to place common functionality around the execution of an operation and isolate this common functionality from more business logic-specific functionality in the OperationEndpoint. An example would be logging the request and response, where a single handler is written and applied to any operation, isolating the OperationEndpoint from this task. Transaction Handlers can also be “chained” for a particular implementation of com.fedex.csr.common.operation.DefaultOperation via Spring bean wiring. The Transaction Handler chains are set in the DefaultOperation bean when it is created. The typical order of events of a CSR operation execution is as follows (transaction handler operations are **bold**):

* 1. validate request (if enabled)
  2. upConvert
  3. **execute request handler(s)**
  4. endpoint execution
  5. **execute response handler(s)**
  6. downConvert
  7. validate response (if enabled)

If a response is set in the OperationContext of an handleRequest method, then:

* 1. No other request handlers will be called.
  2. OperationEndpoint's execute() method will not be called.
  3. handleResponse() will only be called on handlers where handleRequest() has been called.
  4. The response will be returned.

The user can create a Transaction Handlers as they see fit by implementing the com.fedex.csr.common.handler.Handler class. Custom Transaction Handlers must override 3 methods:

1. **public boolean handleRequest(OperationContext operationContext,**

**OperationEndpoint endpoint)**

Called after the request has been validated and up-conversion has taken place, but before the OperationEndpoint's execute method is called.

1. **public boolean handleResponse(OperationContext operationContext,**

**OperationEndpoint endpoint)**

Called after the OperationEndpoint's execute method is called, but before down-conversion and validation has taken place on the response.

1. **public boolean handleFault(OperationContext operationContext,**

**OperationEndpoint endpoint)**

Called whenever a fault has occurred (an exception was thrown). This is typically used for cleanup. Most implementations will probably just need to return true.

## Code Implementation/Spring Wiring for Transaction Handlers

To implement a custom transaction handler, a class must be created that will implement the CSR Handler interface:

**package** com.fedex.sampleservice.utility;

**import** com.fedex.csr.common.handler.Handler;

**public** **class** SampleTransactionHandler **implements** Handler {

**private** **final** FedExLoggerInterface logger = FedExLogger

.*getLogger*(SampleTransactionHandler.**class**);

@Override

**public** **boolean** handleRequest(OperationContext operationContext,

OperationEndpoint endpoint) {

logger.info("SampleTransactionHandler: handleRequest() method");

**return** **true**;

}

@Override

**public** **boolean** handleResponse(OperationContext operationContext,

OperationEndpoint endpoint) {

**return** **true**;

}

@Override

**public** **boolean** handleFault(OperationContext operationContext,

OperationEndpoint endpoint) {

**return** **true**;

}}

In addition, the CSR project’s /src/main/config/properties/operations/operations.xml file will need to be edited to add the following changes:

1. Add in beans for transaction handlers:

<bean id=*"myTransHandler"* class=*"com.fedex.sampleservice.utility.SampleTransactionHandler"* />

1. Add in the handler chain to the DefaultOperation bean:

<bean id=*"sampleOperation"* class=*"com.fedex.csr.common.operation.DefaultOperation"*>

... (rest of bean properties go here)

<property name=*"handlerChain"*>

<list>

<ref bean=*"myTransHandler"* />

</list>

</property>

</bean>

You should see the transaction handlers called in the CSR stack trace when the service is invoked:

1.0 2010-12-21 13:43:53,187 GMT-0600 WTC-200604-w1 [http-127.0.0.1-8080-1] DEBUG APP000001 DefaultOperation.processRequestHandlers [TxId=d8327f34-f4f3-478e-959a-f9d2e84e12e6]calling request handler of type com.fedex.sampleservice.utility.SampleTransactionHandler

1.0 2010-12-21 13:43:53,187 GMT-0600 WTC-200604-w1 [http-127.0.0.1-8080-1] INFO APP000001 SampleTransactionHandler.handleRequest [TxId=d8327f34-f4f3-478e-959a-f9d2e84e12e6]SampleTransactionHandler: handleRequest() method

1. 2010-12-21 13:43:53,218 GMT-0600 WTC-200604-w1 [http-127.0.0.1-8080-1] DEBUG APP000001 DefaultOperation.processResponseHandlers [TxId=d8327f34-f4f3-478e-959a-f9d2e84e12e6]calling response handler of type com.fedex.sampleservice.utility.SampleTransactionHandler

## Using a Transaction Handler to Prevent Logging of Sensitive Data

Due to infoSec rules, the logging of sensitive data in the XML response/request for a particular service is prohibited. “Sensitive” data includes Level I security data such as Credit Card Number, Taxpayer ID, etc. Initial implementations of CSR logged the entire XML request/response if the log level is set to DEBUG or higher. To alleviate this potential InfoSec issue, a custom Transaction Handler can be built and deployed with a user’s project.

This new handler, *com.fedex.csr.common.handler.LogNoSensitiveData.class*, is packaged with the 3.0.1 release of CSR. This handler is created by the app in its operations.xml and takes an XSLT as an argument. In the XSLT, the user can define which elements are “sensitive” and mask out the data field using asterisks or some other suitable character.

The steps to implement this handler include (our SampleService project is referenced below):

1. **Add an XSLT to your project:**

src/main/config/properties/contract/xslts/logNoSensitiveData.xslt. This is a basic XSLT that uses the identity transform to copy all xml verbatim, and also contains a template match for a specified element in the sampleservice namespace, in this case “Attribute1”. The translate function is used in this template to replace specified chars with “masking” chars, such as ‘s’, ‘\_’,

or ‘\*’. The example XSLT is provided below. (The developer will need to adapt the XSLT to meet their project needs and determine which requests/responses in the WSDL contain “sensitive” data):

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:op="http://sample.fedex.com/sampleservice">

<xsl:param name="rep-chars" select="'sssssssssssssssssssssss\_\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*dddddddddd'"/>

<xsl:param name="letters" select="'?:;[]{}|=+-\_)(\*^%$#@!`~ ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789'"/>

<xsl:template match="@\* | node()">

<xsl:copy>

<xsl:apply-templates select="@\* | node()"/>

</xsl:copy>

</xsl:template>

<xsl:template match="op:Attribute1">

<xsl:copy>

<xsl:apply-templates select="@\*"/>

<xsl:value-of select="translate(., $letters, $rep-chars)"/>

</xsl:copy>

</xsl:template>

**</xsl:stylesheet>**

1. **Add the transaction handler for logging sensitive data to operations.xml:**

<!-- 1. add the sensitive data logger class as a bean -->

<bean id="sensitiveLogger" class="com.fedex.csr.common.handler.LogNoSensitiveData">

<property name="xslt">

<value>classpath:contract/xslts/logNoSensitiveData.xslt</value>

</property>

</bean>

1. **Add a handler chain, with the sensitiveLogger bean to the sampleOperation bean in operations.xml:**

<bean id="sampleOperation" class="com.fedex.csr.common.operation.DefaultOperation">

<property name="supportedVersions">

<list>

<bean class="com.fedex.csr.common.operation.SupportedVersion">

<property name="validator" ref="validator.v1"/>

<property name="mapper" ref="mapper.v1" />

</bean>

</list>

</property>

**<!-- 2. add the handler chain with the sensitive logger bean -->**

**<property name="handlerChain">**

**<list>**

**<ref bean="sensitiveLogger"/>**

**</list>**

**</property>**

<property name="endpoints">

<list>

<ref bean="sampleserviceendpoint" />

</list>

</property>

</bean>

At this point, the project can be re-built and deployed in the local container. Since we defined “Attribute1” element as containing sensitive data, it’s data field will be masked out in the application log:

1.0 2011-07-01 07:52:12,687 GMT-0500 WTC-200604-L2 [http-127.0.0.1-8080-1] INFO 4427 LogNoSensitiveData.handleRequest [TxId=2ac68ae6-0141-4ec2-9d69-83ba99920692]in handleRequest()

1.0 2011-07-01 07:52:12,703 GMT-0500 WTC-200604-L2 [http-127.0.0.1-8080-1] ALL 4427 LogNoSensitiveData.handleRequest [TxId=2ac68ae6-0141-4ec2-9d69-83ba99920692]converted request doc:FedExLogEntry{message:<?xml version="1.0" encoding="UTF-8"?><SampleRequest xmlns="http://sample.fedex.com/sampleservice" xmlns:ns2="http://xmlns.fedex.com/csr/common" majorVersion="1" minorVersion="0"><Attribute1>\*\*\*\*\*\_ssssss&amp;s</Attribute1><Attribute2>world</Attribute2></SampleRequest>, objects():null, msgCode:, sourceClassName:, sourceMethodName:}

# Document versus Object

The com.csr.common.operation.MessageContext interface supports two methods of returning the request or response:

* **getObject()**
* **getDocument()**

While your business POJO will typically deal with the Object version of the request (via the getObject() method), at times it may be helpful to retrieve this as a Document Object Model (DOM) Document. The getDocument() method will return a *org.w3c.dom.Document* object, which will allow the user to operate on the request/response using all of the methods available to this Document, such as getElementById(), getChildNodes(), etc.

# Custom Version Extractors

Within the XML Schema Document (XSD), CSR derived services employ a version attribute. In the following sample of an XSD schema, the version is defined as a simpleType element named “versionType”. In the sample, a restriction has also been used to limit the version to integers of zero or greater. Other options are available such as defining allowable ranges of values.

<xs:simpleType name=*"versionType"*>

<xs:restriction base=*"xs:int"*>

<xs:minInclusive value=*"0"*></xs:minInclusive>

</xs:restriction>

</xs:simpleType>

Along with the above definition, you can use the attributeGroup element to define other element characteristics. In the below sample, two attributes are grouped to define a version as being composed of both a major and a minor version (minor version value not used) designation. Note that both the major and minor versions are of the type “versionType” and so must adhere to the limitations described above.

<xs:attributeGroup name=*"versionAttributes"*>

<xs:attribute name=*"majorVersion"* type=*"versionType"* fixed=*"1"* />

<xs:attribute name=*"minorVersion"* type=*"versionType"* default=*"2"* />

</xs:attributeGroup>

Also note in the previous sample that “majorVersion” has been given a *fixed* value of 1. This means only a major version of 1 will be allowed. As an alternative, the ‘default’ attribute as shown in the “minorVersion” definition will assign a value of 2 however other values are acceptable as long as they adhere to the “versionType” definition. To use such an attributeGroup, it is necessary to reference it in the XSD where the response and/or requests are defined. In the following XSD example, a response object called “SampleResponse” is been define. The “versionAttributes” attributeGroup defined above is referenced and this imposes the constraints set in that section of the XSD.

<xs:element name="*SampleResponse*">

<xs:complexType>

<xs:sequence>

<xs:element name="*RespAttr1*" type="*xs:string*"></xs:element>

</xs:sequence>

<xs:attributeGroup ref="*versionAttributes*"/>

</xs:complexType>

</xs:element>

To access the version, a custom version extractor can be created that implements the VersionExtractor interface. In the below XML sample a class implementing VersionExtractor, named LocatorVersionExtractor, is configured within the default operation bean definition.

<bean id="locatorOperation" class="com.fedex.csr.common.operation.DefaultOperation">

<property name="supportedVersions">

<list>

       <bean class="com.fedex.csr.common.operation.SupportedVersion">

          <property name="validator" ref="validator.v1"/>

            <property name="mapper" ref="mapper.v1" />

         </bean>

      </list>

</property>

<property name="versionExtractor">

    <bean class="com.fedex.endpoint.LocatorVersionExtractor" />

   </property>

   <property name="endpoints">

    <list>

       <ref bean="locatorserviceendpoint" />

      </list>

   </property>

</bean>

# Validation

Validation is used to validate request and response objects. There are two types of validation possible in CSR:

* **Default validation**
* **Programmatic validation.**

As mentioned in End point section the DefaultOperation (package com.fedex.csr.common.operation) is the component that calls and uses the other components. It is the glue component and directs what components get called and in what order they are called. DefaultOperation has property SupportedVersion which has a property validator that holds the information about default validator. Here in this sample com.fedex.csr.common.contract.DafaultValidator is the validation bean. Default operation has two properties i.e. requestValidationEnabled and responsevalidationEnabled, by changing the values of these two properties user can control the default validation.

Default validation basically checks for the validity of the arguments sent to the sample service by the client. It checks for the version, null values and empty strings. The default schema validator validates the *entire request* and collects all the schema errors and returns a *ValidatorException* with all the validation errors with proper element names. (From Framework Release 5.0.0) It also checks for the value of argument1 to make sure that it is according to the specific standards (via the SampleService.xslt stylesheet). The value of argument1 should be 5 characters long and any input with a longer or smaller length is caught and is reported as an error to the user. Below you can see the property file and the error message that is reported to the user in case of invalid value of argument1.

<bean id="sampleOperation" class="com.fedex.csr.common.operation.DefaultOperation">

<property name="supportedVersions">

<list>

<bean class="com.fedex.csr.common.operation.SupportedVersion">

<property name="validator" ref="validator.v1"/>

<property name="mapper" ref="mapper.v1" />

</bean>

</list>

</property>

<property name="endpoints">

<list>

<ref bean="sampleserviceendpoint" />

</list>

</property>

</bean>

<bean id="validator.v1" class="com.fedex.csr.common.contract.DefaultValidator">

<property name="majorVersion" value="1" />

<property name="minorVersion" value="0" />

<property name="xsd" ref="SampleService.v1" />

<property name="xslt" ref="SampleService.xslt" />

</bean>

**Error Message:**

SOAP Header: <?xml version="1.0" encoding="UTF-8"?>

<SOAP-ENV:Body xmlns:SOAP-ENV="<http://schemas.xmlsoap.org/soap/envelope/>">

<SOAP-ENV:Fault>

<faultcode>SOAP-ENV:Server</faultcode>

<faultstring xml:lang="en">Fault</faultstring>

<detail>

<cause>UnrecoverableClientError</cause>

<code>StylesheetError</code>

<desc>validation failure for SampleRequest majorVersion=1 xsltFile=SampleService.xslt Error: Stylesheet directed termination:You must supply 5 letters for Attribute1.</desc>

</detail>

</SOAP-ENV:Fault>

</SOAP-ENV:Body>

## Default validation

By default, default validation is enabled for request object and disabled for response object. In order to disable default validation for request object the value of requestValidationEnabled bean property should be set to false.

<bean id="sampleOperation" class="com.fedex.csr.common.operation.DefaultOperation">

<property name="requestValidationEnabled" value="false"/>

<property name="supportedVersions">

<list>

<bean class="com.fedex.csr.common.operation.SupportedVersion">

<property name="validator" ref="validator.v1"/>

<property name="mapper" ref="mapper.v1" />

</bean>

</list>

</property>

<property name="endpoints">

<list>

<ref bean="sampleserviceendpoint" />

</list>

</property>

</bean>

By default validation is disabled for response object. In order to enable default validation for response object the value of responseValidationEnabled bean property should be set to false.

<bean id="sampleOperation" class="com.fedex.csr.common.operation.DefaultOperation">

<property name="responseValidationEnabled" value="true"/>

<property name="supportedVersions">

<list>

<bean class="com.fedex.csr.common.operation.SupportedVersion">

<property name="validator" ref="validator.v1"/>

<property name="mapper" ref="mapper.v1" />

</bean>

</list>

</property>

<property name="endpoints">

<list>

<ref bean="sampleserviceendpoint" />

</list>

</property>

</bean>

## Programmatic validation:

In order to use programmatic validation Endpoint class should extend the AbstractOperationEndpoint and override the validate methods. Operations.xml needs to be updated to remove the validator property in supportedversion bean and to add property “major version”. There are two validate methods one for request and one for response:

**Sample endpoint class to implement programmatic validation:**

package com.fedex.endpoint;

import com.fedex.csr.common.operation.AbstractOperationEndpoint;

import com.fedex.csr.common.operation.OperationContext;

import com.fedex.csr.common.operation.OperationException;

import com.fedex.objects.SampleRequest;

import com.fedex.objects.SampleResponse;

import com.fedex.pojo.SamplePojo;

public class SampleEndpoint extends AbstractOperationEndpoint{

private SamplePojo samplePojo;

public void setSamplePojo(SamplePojo pojo) {

this.samplePojo = pojo;

public void execute(OperationContext context) {

SampleRequest request =

(SampleRequest) context.getRequestContext().getObject();

SampleResponse reply = samplePojo.ProcessRequest(request);

context.getResponseContext().setObject(reply);

}

public void validateRequest(Object object, int majorVersion) {

User should put the validation logic here.

}

public void validateResponse(Object object, int majorVersion) {

User should put the validation logic here.

}}

**Sample operations.xml to disable the default operation and enable the programmatic validation**.

<bean id="sampleOperation" class="com.fedex.csr.common.operation.DefaultOperation">

<property name="supportedVersions">

<list>

<bean class="com.fedex.csr.common.operation.SupportedVersion">

<property name="majorversion" value=”1"/>

<property name="mapper" ref="mapper.v1" />

</bean>

</list>

</property>

<property name="endpoints">

<list>

<ref bean="sampleserviceendpoint" />

</list>

</property>

</bean>

# Security with CSR

The Common Service Runtime handles client authentication and authorization of the client to “access” your service automatically. If more fine-grained authorization is desired, it can be implemented per the Development Framework User Guide. The security configuration in CSR is primarily handled by the *security.xml* file that must be present in the *operations* folder with the *operations.xml* and *definitions.xml* files.

## Security.xml

There must be a *security.xml* present in the operations folder with the *operations.xml* and *definitions.xml*. The *security.xml* contains the following required and optional elements:

1. **Bean definition for “theJudge”**
   * Optional – If one is not specified, then the default Judge will be used that provides Token Authentication and Authorization of the "client" to "svcAction" on "serviceId"
2. **Bean definition for “ServiceId”**
   * Required – Defines the unique name for the service and must be provided to clients as they will need to provide it to generate a token via *getToken(…)* to call your service.
3. **Bean definition for “svcAction”**
   * Required – Defines the action in the authorization policy (either file based or ESC based) that is used to determine if the client is allowed to “access” your service.
4. **Bean definitions for the classes needed to retrieve an ESC based policy**
   * Required – If using an ESC based authorization policy
   * Commented – If using a file based authorization policy
5. **Bean definition for CSSBootstrap with a file based policy**
   * Required – If using a file based authorization policy
   * Commented – If using an ESC based authorization policy
6. **Bean definition for CSSBootstrap with an ESC based policy**
   * Required – If using an ESC based policy
   * Commented – If using a file based policy

## The Judge

The judge handles the default authentication and authorization for a client to “access” your service. There is a default Judge configured in CSR to provide this functionality. The option below should be uncommented in the *security.xml* if you are going to override the default Judge with another Judge with additional features. Additional Judges will be added in later releases to help provide more fine-grained authorization for your service.

<bean id="theJudge" class="com.fedex.csr.security.TheJudgeImpl">

<property name="serviceId" ref="serviceId" />

<property name="svcAction" ref="svcAction" />

<property name="authenticator" ref="securityAuthenticator" />

<property name="authorizor" ref="securityAuthorizor" />

</bean>

### The Judge and Jury

The judge and jury both authenticate and authorize. Below is the code snippet taken from the security.lesserjudge.xml.

<bean id="theJudge" class="com.fedex.csr.security.TheJudgeAndJuryImpl">

<property name="serviceId" ref="serviceId" />

<property name="svcAction" ref="svcAction" />

<property name="authenticator" ref="securityAuthenticator" />

<property name="authorizor" ref="securityAuthorizor" />

</bean>

### The Lesser Judge

The lesser judge only authenticates. Below is the code snippet for the lesser judge i.e.

<bean id="theJudge" class="com.fedex.csr.security.TheLesserJudgeImpl">

<property name="serviceId" ref="serviceId" />

<property name="authenticator" ref="securityAuthenticator" />

</bean>

## Service Id (or Service Name)

The service ID is a required entry the specifies the string value representing your service. This value is what a client would need to generate a security token via the security API. The main point to remember is that a client MUST know the value that you enter here in order to communicate with your service.

<bean id="serviceId" class="java.lang.String">

<constructor-arg type="java.lang.String" value="123\_SampleService" />

</bean>

## Service Action

This configuration entry defines the "action" that will be automatically supplied to the authorization method in the Judge to be used by isAllowed to authorize the client to "access" the service. It must match an action in the security policy and ultimately there must be a rule that is something like AuthNClient is allowed to [svcAction] [serviceId] e.g. AuthNClient is allowed to access 123\_SampleService)

<bean id="svcAction" class="java.lang.String">

<constructor-arg type="java.lang.String" value="access" />

</bean>

## Configuration for an ESC Based Authorization Policy

It is highly recommended that your service use an ESC based authorization policy to manage access to your service and more fine-grained access if requested by the data owner. Below is the configuration that would need to be in the security.xml file to enable the CSSBootstrap to initialize the Security API to retrieve the policy from the data store for this service. To learn more about ESC based policies, go to keyword “ESC”. The *defaultUri* in the *wsAPITemplate* will need to change between DEV/TEST/PROD levels.

<!-- Configuration needed by the Security API to retrieve the policy from CDS (ESC based policy) -->

<!-- Comment this section out if you are not using an ESC based authorization policy -->

<bean id=*"cdsClient"* class=*"com.fedex.cds.CdsClient"*>

<property name=*"webServiceTemplate"* ref=*"wsAPITemplate"*/>

</bean>

<bean id=*"cdsSecurityBase"* class=*"com.fedex.cds.CdsSecurityBase"*>

<property name=*"cdsClient"* ref=*"cdsClient"*/>

</bean>

<bean id=*"cdsSecurityAction"* class=*"com.fedex.cds.CdsSecurityAction"* parent=*"cdsSecurityBase"*/>

<bean id=*"cdsSecurityRole"* class=*"com.fedex.cds.CdsSecurityRole"* parent=*"cdsSecurityBase"*/>

<bean id=*"cdsSecurityAppRole"* class=*"com.fedex.cds.CdsSecurityAppRole"* parent=*"cdsSecurityBase"*/>

<bean id=*"cdsSecurityGroupRole"* class=*"com.fedex.cds.CdsSecurityGroupRole"* parent=*"cdsSecurityBase"*/>

<bean id=*"cdsSecurityUserRole"* class=*"com.fedex.cds.CdsSecurityUserRole"* parent=*"cdsSecurityBase"*/>

<bean id=*"cdsSecurityExtendedRule"* class=*"com.fedex.cds.CdsSecurityExtendedRule"* parent=*"cdsSecurityBase"*/>

<bean id=*"cdsSecurityExtendedXRef"* class=*"com.fedex.cds.CdsSecurityExtRuleXRef"* parent=*"cdsSecurityBase"*/>

<bean id=*"cdsSecurityRule"* class=*"com.fedex.cds.CdsSecurityRule"* parent=*"cdsSecurityBase"*/>

<bean id=*"cdsSecurityResource"* class=*"com.fedex.cds.CdsSecurityResource"* parent=*"cdsSecurityBase"*/>

<bean id=*"wsAPITemplate"* class=*"org.springframework.ws.client.core.WebServiceTemplate"*>

<property name=*"messageFactory"*>

<bean class=*"org.springframework.ws.soap.saaj.SaajSoapMessageFactory"*>

<property name=*"messageFactory"*>

<bean class=*"com.sun.xml.internal.messaging.saaj.soap.ver1\_1.SOAPMessageFactory1\_1Impl"*/>

</property>

</bean>

</property>

<property name=*"defaultUri"* value=*"http://cds-level2.ute.fedex.com/CommonDataService/clear/"*/>

<property name=*"marshaller"* ref=*"cds-marshaller"*/>

<property name=*"unmarshaller"* ref=*"cds-marshaller"*/>

<property name=*"interceptors"*>

<list>

<ref local=*"wsSecurityAPIInterceptor"*/>

</list>

</property>

</bean>

<bean id=*"wsSecurityAPIInterceptor"* class=*"com.fedex.cds.client.security.SpringClientWsSecurityTokenInterceptor"*>

<property name=*"defaultServiceAppId"* value=*"943415\_cds"*/>

<property name=*"webServiceTemplate"* ref=*"wsAPITemplate"*/>

</bean>

<bean id=*"cds-marshaller"* class=*"org.springframework.oxm.jaxb.Jaxb2Marshaller"*>

<property name=*"contextPath"* value=*"com.fedex.xmlns.cds2"*/>

</bean>

<!-- End configuration for retrieving ESC based policies -->

<!-- Security Bootstrap for v1.5 -->

<!-- Use this configuration with the ESC based policy -->

<!-- Comment out if using the file based policy -->

<bean id=*"securityBootstrap"* class=*"com.fedex.security.bootstrap.CSSBootstrap15"* init-method=*"buildPolicy"* destroy-method=*"cancelTimerTasks"* depends-on=*"cdsSecurityAction,cdsSecurityRole,cdsSecurityAppRole,cdsSecurityGroupRole,cdsSecurityUserRole,cdsSecurityExtendedRule,cdsSecurityExtendedXRef,cdsSecurityRule,cdsSecurityResource"*>

<property name=*"client"* value=*"true"*/>

<property name=*"service"* value=*"true"*/>

<property name=*"clientProperties"* value=*"client.properties"*/>

<property name=*"securityProperties"* value=*"security.properties"*/>

</bean>

## Configuration for a File Based Authorization Policy

If for some reason you may not be able to utilize the ESC based policy, there is a file based policy as well. To configure CSR to use your file based policy, you need to make sure that the configuration below is uncommented. For additional information about file based policies, please read the Security section of the Development Framework User Guide.

<!-- Example Usage:

<constructor-arg index="0" type="boolean"> true or false to be configured as a client (generate tokens)

<constructor-arg index="1" type="boolean"> true or false to be configured as a service (validate tokens and authorization)

<constructor-arg index="2" type="java.lang.String"> location and name of client.properties file

<constructor-arg index="3" type="java.lang.String"> location and name of security.properties file

constructor-arg index="4" type="java.lang.String"> location and name of authorization.rules file

<constructor-arg index="5" type="java.lang.String"> location and name of authorization.roles file

-->

<bean id=*"securityBootstrap"* class=*"com.fedex.security.bootstrap.CSSBootstrap"*>

<constructor-arg index=*"0"* type=*"boolean"*><value>false</value></constructor-arg>

<constructor-arg index=*"1"* type=*"boolean"*><value>true</value></constructor-arg>

<constructor-arg index=*"2"* type=*"java.lang.String"*><value>client.properties</value></constructor-arg>

<constructor-arg index=*"3"* type=*"java.lang.String"*><value>security.properties</value></constructor-arg>

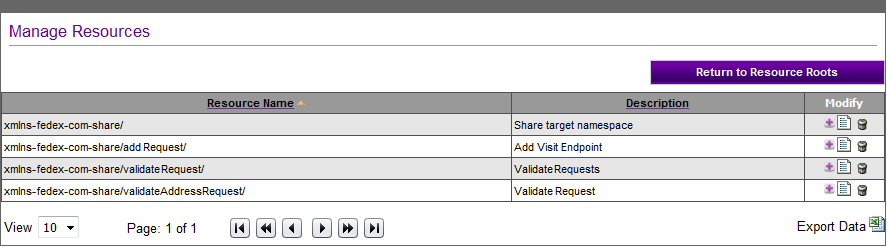
<constructor-arg index=*"4"* type=*"java.lang.String"*><value>authorization.rules</value></constructor-arg>

<constructor-arg index=*"5"* type=*"java.lang.String"*><value>authorization.roles</value></constructor-arg>

</bean>

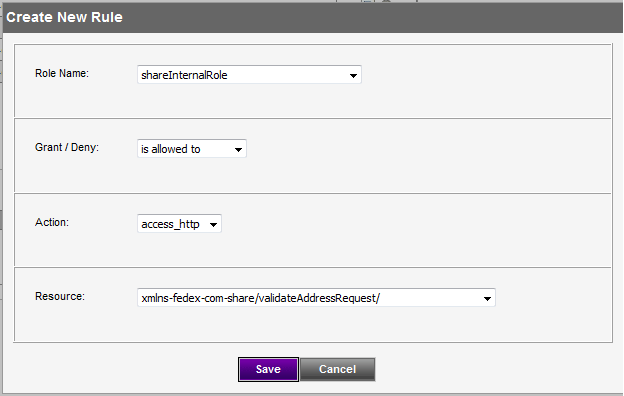
## Judge/Jury Security setup in ESC

1. Login to ESC, select your symphony profile name and the application ID.
2. Click on the **”ACTION”** tab and create a new action and enter the action description. For example: ***“access\_http”*** or ***“access\_https”****.*
3. Click on the **“RESOURCE”** tab and create a ROOT shared target namespace resource and enter the description. For example: ***“****xmlns-fedex-com-share/****”****.*
4. Now click on the ROOT resource you just created and start creating the child resources as shown in the screenshot given below. It is important to note that each child resource is the operation you are trying to secure. In this example, child resources or the operations we are trying to secure are addRequest, validateRequest, etc.



1. Create appropriate roles by selecting the **“ROLE”** tab.
2. Now click on the **“POLICY”** tab and click on **“Create New Rule”** button. Now select appropriate values from the drop down menu for each of the following: “Role Name”, “Grant/Deny”, “Action”, and “Resource”. For example:

Role you created in step 5.



Resource you created in step 4.

Action you create in step 2.

# Workflow Counters

The Common Service Runtime provides a workflow counter and workflow handler. The workflow counter provides the ability to establish limits on the number of current requests that will be processed. Requests that are sent that exceed the upper limit of the workflow counter will receive a SOAP Fault from the service. There are two configuration files that are needed to support workflow counters, the operations.xml and the workflow.properties files. If you set the logging level to debug, you will be able to see the workflow counters increment and decrement as requests are processed.

## Operations.xml

To enable workflow counters for your service, please add the following entries in the operation.xml: Configure the bean definition for the workflow handler. Specify the location and name of the workflow.properties file. The handler will first attempt to load the properties file from the classloader, if that fails; it will print a warning message and attempt to load from the file system.

<!-- Workflow handler -->

<bean id=*"workflowHandler"* class=*"com.fedex.csr.workflow.WorkflowHandler"*>

<property name=*"workflowPropertiesFile"* value=*"workflow.properties"*/>

</bean>

Add the following property (highlighted in yellow) to the bean definition for sampleOperation to add the workflowHandler bean to the handlerChain so that it is invoked in requests and responses.

<bean id="sampleOperation" class="com.fedex.csr.common.operation.DefaultOperation">

<property name="supportedVersions">

<list>

<bean class="com.fedex.csr.common.operation.SupportedVersion">

<property name="validator" ref="validator.v1"/>

<property name="mapper" ref="mapper.v1" />

</bean>

</list>

</property>

<property name="handlerChain">

<list>

<ref bean="workflowHandler" />

</list>

</property>

<property name="endpoints">

<list>

<ref bean="sampleserviceendpoint" />

</list>

</property>

</bean>

## Workflow.properties

The workflow.properties file provides the upper limits of each workflow and which requests are mapped to which workflow. As mentioned above, the workflow.properties needs to be in the classpath or on the file system in a location that is accessible by the application. Here is an example of the workflow.properties file:

#List of workflow names

workflow.segregation.names=Shipping,Sample,Rating,Routing

#Mapping of the workflow names with the upper limits for the workflow counters

Shipping=25

Sample=4

Rating=10

Routing=50

#List of possible requests that will be mapped to workflow request names

workflow.requests=SampleRequest,shippingRequest,ratingRequest,routingRequest,groundShippingRequest

#Mappings of the requests to workflow names

shippingRequest=Shipping

ratingRequest=Rating

SampleRequest=Sample

routingRequest=Routing

groundShippingRequest=Shipping

## SOAP Fault Exception

When a workflow counter is exceeded, the workflow handler will throw a SOAP Fault back to the client.

<SOAP-ENV:Body xmlns:SOAP-ENV="<http://schemas.xmlsoap.org/soap/envelope/>">  
    <SOAP-ENV:Fault>  
        <faultcode>SOAP-ENV:Server</faultcode>  
        <faultstring xml:lang="en">Fault</faultstring>  
        <detail>  
            <cause>UnrecoverableClientError</cause>  
            <code>Workflow Exceeded</code>  
            <desc>Max workflow limit exceeded attempting processing request [SampleRequest], please try again later.</desc>  
        </detail>  
    </SOAP-ENV:Fault>  
</SOAP-ENV:Body>

## Fault message sent to the client for Governor

<SOAP-ENV:Fault>

<faultcode>SOAP-ENV:Server</faultcode>

<faultstring xml:lang="en">Fault</faultstring>

<detail>

<cause>Transient</cause>

<code>UsageLevelExceeded</code>

<desc>

Client appId 4067 exceeded governor limit max:10 requests per min; Retry in 58 secs; Client Id=4067 Planned=10 hits/min; Requests denied =18; Ceiling max=10; Interval Expiry=Thu Nov 17 14:50:52 CST 2011

</desc>

</detail>

</SOAP-ENV:Fault>

## Fault message sent to the client for Workflow

<SOAP-ENV:Fault>

<faultcode>SOAP-ENV:Server</faultcode>

<faultstring xml:lang="en">Fault</faultstring>

<detail>

<cause>UnrecoverableClientError</cause>

<code>WorkflowExceeded</code>

<desc>Workflow name [Sample] exceeded counter max: 1, exceeded: 1</desc>

</detail>

</SOAP-ENV:Fault>

# Governor and Workflow Enhancements

The previous version of Common Service Runtime supported workflow mechanism to control the concurrent requests on a web service using the Workflow counters and had an inbuilt feature called Governor. In this release, the Governor feature can now be implemented/controlled via the workflow.properties file.

Previous version of workflow counter functionality controlled the concurrent requests at service level using the configuration setting in workflow properties file but was not able to control the concurrency for a service at the application level. At the same time, the Governor functionality controlled the user transactions for an interval of time. The owner was able to allow these two features independently, but it did not allow the application owner to control the user specific transactions and restriction of the user level concurrency was not available either. Governor and the Workflow use same file for their properties but can be used independently.

## Workflow Enhancements

The code modification example is given below. As you can see that now the owner can specify number of parallel connections at the application level. For example, 4068.Routing.wfc=2 basically means that the app ID 4068 using sample service can have 2 parallel connections at one time. If there are no specific settings for an app ID, then the default value (Sample=5 in this example) will be used.

#List of workflow names

workflow.segregation.names=Shipping,Sample,Rating,Routing

#Mapping of the workflow names with the upper limits for the workflow counters

Shipping=25

Rating=2

Sample=5

Routing=5

#appId.workflow=integer name controls the appId level workflow counters

4068.Routing.wfc=2

4067.Sample.wfc=3

4068.Sample.wfc=3

#List of possible requests that will be mapped to workflow request names

workflow.requests=SampleRequest,shippingRequest,RatingRequest,routingRequest,groundShippingRequest

#Mappings of the requests to workflow names

shippingRequest=Shipping

RatingRequest=Rating

SampleRequest=Sample

routingRequest=Routing

groundShippingRequest=Shipping

## Governor Enhancements

The code modification example is given below. A you can see that there is a new option added in the code i.e. 4068.maxTransactionsPerMinute=2. This line of code basically tells the service that app ID 4068 can have 2 maximum transactions in two minutes. If the same app ID attempts to make another transaction over the specified limit, the service sends a response letting the requestor know that their request cannot be completed at the moment and that they should try again after N number of seconds.

The first thing that is checked for a specific request is the maxTransactionsPerMinute condition. If an app ID passes this condition, then other conditions like Rating=2, Sample=5 or Routing=5 are checked for that specific app ID if available. But it is important to mention here that Governor and Workflow can be used independently and if Governor properties are not available, then Workflow properties are used for that specific app ID. If no Workflow properties are specified for an app ID, then default value is used.

#List of workflow names

workflow.segregation.names=Shipping,Sample,Rating,Routing

#Mapping of the workflow names with the upper limits for the workflow counters

Shipping=25

Rating=2

Sample=5

Routing=5

#appId.workflow=integer name controls the appId level workflow counters

4068.Routing.wfc=2

4067.Sample.wfc=3

4068.Sample.wfc=3

#List of possible requests that will be mapped to workflow request names

workflow.requests=SampleRequest,shippingRequest,RatingRequest,routingRequest,groundShippingRequest

#Mappings of the requests to workflow names

shippingRequest=Shipping

RatingRequest=Rating

SampleRequest=Sample

routingRequest=Routing

groundShippingRequest=Shipping

#governing TPM for appId- appid.maxTransactionsPerMinute=number(integer)of hits in a minute

4068.maxTransactionsPerMinute=2

4067.maxTransactionsPerMinute=3

## Governor Override via Plugin

A governor plugin has been developed to allow greater flexibility by the app teams on implementing desired behavior when service limits are exceeded. This plugin is based on the following interface:

**public** **interface** GovernorPlugin {

/\*\*

\* Throw exception when request exceeded governor limit or just return to proceed.

\* **@param** clientAppId

\* **@param** plannedHits

\* **@param** totalExceededHits

\* **@param** intervalExpiration

\* **@param** accountDescription

\*/

**public** **void** handleExceededGovernorLimit(String clientAppId, **long** plannedHits, **long** totalExceededHits, Calendar intervalExpiration, String accountDescription);

}

For example, instead of returning a SOAP Fault to the calling client, the service owner may wish to execute alternative functionality, such as:

1. Proceed normally.
2. Update response headers to send info/warning messages to client with or without processing the Request.
3. Throw custom exception without processing the request.

To implement the plugin, the service owner can 1) create a class that extends the com.fedex.csr.usage.AbstractGovernorPlugin.AbstractGovernorPlugin class, placing the new functionality in the handleExceededGovenrnorLimit() method, or 2) extend the GovernorPlugin interface directly. If normal message processing is desired, simply return from this method and the request will be processed. This plugin can be created in operations.xml and wired into the Governor bean as follows:

<bean id=*"sampleOperation"* class=*"com.fedex.csr.common.operation.DefaultOperation"*>

<property name=*"requestValidationEnabled"* value=*"true"*/>

<property name=*"supportedVersions"*>

<list>

<bean class=*"com.fedex.csr.common.operation.SupportedVersion"*>

<property name=*"validator"* ref=*"validator.v1"*/>

<property name=*"mapper"* ref=*"mapper.v1"* />

</bean>

</list>

</property>

<property name=*"handlerChain"*>

<list>

<ref bean=*"workflowHandler"*/>

</list>

</property>

<property name=*"endpoints"*>

<list>

<ref bean=*"sampleserviceendpoint"* />

</list>

</property>

</bean>

<bean id=*"governorPlugin"* class=*"com.fedex.governor.SSGovernorPlugin"* />

<bean id=*"workflowHandler"* class=*"com.fedex.csr.workflow.WorkflowHandler"*>

<property name=*"workflowPropertiesFile"* value=*"workflow.properties"*/>

<property name=*"governorPlugin"* ref=*"governorPlugin"*></property>

</bean>

For an example implementation of the Governor plugin, refer to the “SampleServiceGovernorWorkflow” project distributed with “SamplesAndDemos4.1.0” download from FedExtras.

## Comparison between Workflow and Governor

|  |  |  |
| --- | --- | --- |
| **Type** | **Governor** | **Workflow** |
| **Purpose** | Transactions Per Minute per app id | Concurrent Requests per workflow or workflow/appId |
| **Method of control** | <App\_ID>.maxTransactionsPerMinute  Via Workflow.properties | <App\_id>.<workflowname>.wfc OR  <workflowname>.wfc  Via Workflow.properties |
| **Restrict by Client ID** | Yes (Version 3.0.1 and above) | Yes (Version 3.0.1 and above) |
| **Restrict by operation type** | No | Yes |
| **Bean Creation** | Governor Bean created by CSR.  User creates WF handler in operation.xml to provide workflow.properties object to CSR. | User creates WF handler to get workflow.properties object in operations.xml |
| **When to Use** | If the service knows it’s TPM (transactions per minute) and wants to restrict a client’s TPM. | If service knows it’s supported concurrent transactions and wants to restrict a client’s transactions. |

# Supported Operations

The following supported operations are available in CSR:

1. **Ping:** A ping operation will hit the end point and returns a true or false based on the availability of the Web service. This allows the client to check if the web-service is available to send the requests. This Web service operation allows a caller to ‘ping’ the service and determine if the service is responsive.

**Inputs:** None.

**Output:** PingReply object.

1. **GetSupportedOperations():** Service operation method lists all the operations that are available for endpoints.

**Inputs:** None.

**Output:** List of strings, where each string contains the target namespace/request type. This list is specified in the service’s operations.xml file in the war deployment.

# Client Side Helper Classes

## JaxWS Implementation

The following helper classes are available in CSR.

1. **JaxWSLoggingHandler**
2. **JaxWsSecurityHandler**
3. **JaxWsPropertiesHandler**

In order to use helper classes, the users are required to use DefaulthandlerResolver class or create their own handler resolver class. The handler resolver class created by the user should implement HandlerResolver interface and override getHandlerChain() abstract method.

**Sample handler resolver class:**

**import** java.util.ArrayList;

**import** java.util.List;

**import** javax.xml.ws.handler.Handler;

**import** javax.xml.ws.handler.HandlerResolver;

**import** javax.xml.ws.handler.PortInfo;

**import** javax.xml.ws.handler.soap.SOAPMessageContext;

**import** com.fedex.csr.common.clientadapter.ws.JaxWsLoggingHandler;

**import** com.fedex.csr.common.clientadapter.ws.JaxWsSecurityHandler;

**class** MyHandlerResolver **implements** HandlerResolver {

**private** List<Handler> handlerList = **null**;

**private** String \_employeeNumber;

**public** MyHandlerResolver(String employeeNumber) {

handlerList = **null**;

\_employeeNumber = employeeNumber;

}

@Override

**public** List<Handler> getHandlerChain(PortInfo portInfo) {

**if** (handlerList == **null**) {

handlerList = **new** ArrayList<Handler>();

handlerList.add(**new** JaxWsSecurityHandler());

handlerList.add(**new** JaxWsLoggingHandler());

handlerList.add(**new** JaxWsPropertisHandler());

handlerList.add(**new** JaxWsPropertiesChainingHandler());

}

**return** handlerList;

}

}

### JaxWsLoggingHandler :

This handler class is used to log SOAP request and response messages and HTTP properties. In order to use this handler class this handler needs to be added to the handlerList of handler resolver class. This Logging handler does not impact performance as it only logs at the debug level.

public List<Handler> getHandlerChain(PortInfo portInfo) {

if (handlerList == null) {

handlerList = new ArrayList<Handler>();

handlerList.add(new JaxWsLoggingHandler());

}

return handlerList;

}

### JaxWsSecurityHandler:

This handler class is used to set the security token in soap header. Add this handler to the chain and use ClientInvocationContext to set the endpoint service app id and (optional) employee id in making the request.

**Example:**

With only the Application ID

ClientInvocationContext cContext = ClientInvocationContext.get();

cContext.setEndpointServiceAppId("344\_JCLS");

**Example:**

With Both Application ID and the employee ID

ClientInvocationContext cContext = ClientInvocationContext.get();

cContext.setEndpointServiceAppId("344\_JCLS");

cContext.setUserId("XXXXXX");

In order to use this handler class this handler needs to be added to the handlerList of handler resolver class.

**public** List<Handler> getHandlerChain(PortInfo portInfo) {

**if** (handlerList == **null**) {

handlerList = **new** ArrayList<Handler>();

handlerList.add(**new** JaxWsSecurityHandler());

}

**return** handlerList;

}

#### Service Calling Other Services:

Let us consider a service named FedEx\_Serv (app id: FedEx\_Serv) that needs to talk to services A, B and C respectively (app ids: A, B and C). Service A handle user’s credit card data, service B deals with the shipping details and the service C deals with the user account details. In this scenario, following code example shows how FedEx\_Serv will talk to service A, B and C respectively.

ClientInvocationContext cContextAppIdA = ClientInvocationContext.get();

cContextAppIdA.setEndpointServiceAppId("A");

ClientInvocationContext cContextAppIdB = ClientInvocationContext.get();

cContextAppIdB.setEndpointServiceAppId("B");

ClientInvocationContext cContextAppIdC = ClientInvocationContext.get();

cContextAppIdC.setEndpointServiceAppId("C");

#### JaxWsPropertiesHandler:

When a Property is configured, it will automatically be pulled out of the Protocol Header and made available to the service. Two handler classes are provided to set properties on the client side. This JaxWsPropertiesHandler class is used to handle transaction Id and tracer in ClientInvocationContext. Add this handler to the chain and use ClientInvocationContext to set the transaction Id and log transaction.

**Example:**

*ClientInvocationContext.get().setTransactionId("TEST");*

*ClientInvocationContext.get().setLogTransaction(InvocationContext.LOG\_TRANSACTION\_FULL);*

public List<Handler> getHandlerChain(PortInfo portInfo) {

if (handlerList == null) {

handlerList = new ArrayList<Handler>();

handlerList.add(new JaxWsPropertisHandler());

}

return handlerList;

}

**JaxWsPropertiesChainingHandler:**

This handler class takes care of “chainable” properties. If a property **isChainable**, then it is allowed to go outbound from the service to another CSR service.

**public** List<Handler> getHandlerChain(PortInfo portInfo) {

**if** (handlerList == **null**) {

handlerList = **new** ArrayList<Handler>();

handlerList.add(**new** JaxWsSecurityHandler());

handlerList.add(**new** JaxWsLoggingHandler());

handlerList.add(**new** JaxWsPropertiesHandler());

handlerList.add(**new** JaxWsPropertiesChainingHandler());

}

**return** handlerList;

}

## Spring REST Implementation

The following helper classes are available in CSR.

1. **RestCompressionInterceptor**
2. **RestErrorHandler**
3. **RestLoggingInterceptor**
4. **RestPropertiesChainingInterceptor**
5. **RestPropertiesInterceptor**
6. **RestSecurityInterceptor**

In order to use helper classes, the users are required to use Spring RESTTemplate class. These are interceptors for modifying the request / response and a single class to assist with error resolution.

### Sample Spring Context

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

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=[*http://www.w3.org/2001/XMLSchema-instance*](http://www.w3.org/2001/XMLSchema-instance)

xmlns:p=*"http://www.springframework.org/schema/p"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:util=*"http://www.springframework.org/schema/util"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/util*

*http://www.springframework.org/schema/util/spring-util.xsd"*>

<bean id=*"restTemplate"* class=*"org.springframework.web.client.RestTemplate"*>

<constructor-arg>

<!-- this is needed for the logging interceptor -->

<bean class=*"org.springframework.http.client.BufferingClientHttpRequestFactory"*>

<constructor-arg>

<bean class=*"org.springframework.http.client.HttpComponentsClientHttpRequestFactory"*>

<constructor-arg ref=*"httpClient"* />

</bean>

</constructor-arg>

</bean>

</constructor-arg>

<property name=*"messageConverters"* ref=*"messageConvertersList"* />

<property name=*"interceptors"* ref=*"interceptorList"* />

<property name=*"errorHandler"* ref=*"errorHandler"* />

</bean>

<bean id=*"httpClient"* class=*"org.apache.http.impl.client.DefaultHttpClient"*>

<constructor-arg>

<bean class=*"org.apache.http.impl.conn.PoolingClientConnectionManager"* />

</constructor-arg>

</bean>

<util:list id=*"messageConvertersList"*>

<bean class=*"org.springframework.http.converter.xml.MarshallingHttpMessageConverter"*>

<property name=*"marshaller"* ref=*"jaxbMarshaller"* />

<property name=*"unmarshaller"* ref=*"jaxbMarshaller"* />

</bean>

</util:list>

<bean id=*"jaxbMarshaller"* class=*"org.springframework.oxm.jaxb.Jaxb2Marshaller"*>

<property name=*"contextPath"* value=*"com.fedex.jaxb.v1"* />

</bean>

<util:list id=*"interceptorList"*>

<bean class=*"com.fedex.csr.common.clientadapter.rest.RestPropertiesChainingInterceptor"*/>

<bean class=*"com.fedex.csr.common.clientadapter.rest.RestPropertiesInterceptor"*/>

<bean class=*"com.fedex.csr.common.clientadapter.rest.RestSecurityInterceptor"*>

<property name=*"defaultServiceAppId"* value=*"123\_SampleService"* />

</bean>

<!-- Compression should be the LAST interceptor -->

<bean class=*"com.fedex.csr.common.clientadapter.rest.RestLoggingInterceptor"*>

<property name=*"logOnlyExtraHeaders"* value=*"false"* />

</bean>

<bean class=*"com.fedex.csr.common.clientadapter.rest.RestCompressionInterceptor"*>

<property name=*"logStats"* value=*"true"* />

<property name=*"alwaysCompressRequest"* value=*"true"* />

</bean>

</util:list>

<bean id=*"errorHandler"* class=*"com.fedex.csr.common.clientadapter.rest.RestErrorHandler"*/>

</beans>

### RestCompressionInterceptor

This interceptor works with the CSR2 compression filter on the server side. It will compress the request and the response. It has two configurations.

* logStats: This will log statistics of compression levels to the STATS log defined in log4j2.xml.
* alwaysCompressRequest: This will force compression of the request even if the compressed size is bigger than the non compressed size. This is to match the CSR2 filters capability for only accept compressed requests.

### RestErrorHandler

This is an error handler to be used with the RESTTemplate. It behaves similar to the default error handler. The only difference is if the CSR service throws a 500 error with a CSRError XML object as the HTTP body, this handler will convert the HTTP Response body (CSR Error XML) to a new ClientException object. The ClientException object will contain the information in the HTTP Response Body. Then this ClientException will be thrown instead of the normal HTTPServerErrorException object.

### RestLoggingInterceptor

This will log the outgoing request headers and body along with the incoming response headers and body. If an exception occurs during processing, it will also notify that it was unable to log the response.

### RestPropertiesChainingInterceptor

This will set all the values in the thread local HeaderContext as headers in the HTTP request.

### RestPropertiesInterceptor

This will set all the values in the thread local ClientInvocationContext as headers in the HTTP request. Please see the examples in the previous section.

### RestSecurityInterceptor

This has the same functionality as the JaxWsSecurityHandler section. This will populate the X-CSR-SECURITY\_TOKEN header with the token needed for security.

# Logger, TransID and Tracer Flag

CSR will generate a Transaction ID on every Request or use the Transaction ID supplied in the protocol header. When using framework logging, the Transaction ID will be added to every log message. This aids debugging when searching for a transaction in a log file. This also allows the Transaction ID to be passed from one service to the next service, aiding in debugging a transaction that spans multiple services.

* **Default Transaction Logging:**

In this example, the service has a class named MyPojo that simply writes a log message. Upon examining the log output, notice that a Transaction ID was supplied automatically.

|  |
| --- |
| **CLASS MyPojo** |
| import com.fedex.framework.logging.\*;  FedExLoggerInterface logger = FedExLogger.getLogger("MyPojo");  logger.info("Testing transaction logging."); |
| **Log Output** |
| INFO APP000001 MyPojo.ProcessRequest [**TxId=9a7fe9b9-4e22-4d3c-ae43-712d712e2f4e**]Logging from my main class |

In the next example, an additional class named AnotherPojo is added. MyPojo logs a message and then calls a method in AnotherPojo which also logs a message. Even though different classes are logging, each use the same Transaction ID since they are included in a single transaction.

|  |  |
| --- | --- |
| **CLASS MyPojo** | |
| import com.fedex.framework.logging.\*;  FedExLoggerInterface logger = FedExLogger.getLogger("MyClassName");  logger.info("Testing transaction logging.");  AnotherPojo ap = new AnotherPojo();  ap.logAnotherMessage(); | |
| **Log Output** |
| INFO APP000001 MyPojo.ProcessRequest [**TxId=9a7fe9b9-4e22-4d3c-ae43-712d712e2f4e**]Logging from my main class |
| **CLASS AnotherPojo** | | | |
| FedExLoggerInterface logger = FedExLogger.*getLogger*("AnotherPojo");  logger.error("Logging from a method in another class"); | | |
| **Log Output** | | | |
| INFO APP000001 MyPojo.ProcessRequest [**TxId=9a7fe9b9-4e22-4d3c-ae43-712d712e2f4e**]Logging from my main class  INFO APP000001 AnotherPojo.logAnotherMessage [**TxId=9a7fe9b9-4e22-4d3c-ae43-712d712e2f4e**]Logging from a method in another class | | | |

* **Assigning values to Transaction IDs.**

Calling the setTransactionID() method on the ClientInvocationContext allows the ID to be set to a specified value. In the example below, the client application sets the Transaction ID and the server uses it for each log message. NOTE: The Handler Resolver class of the client must include the com.fedex.csr.common.clientadapter.ws.JaxWsPropertiesHandler class of CSR:

handlerList.add(**new** JaxWsPropertiesHandler());

|  |
| --- |
| **Class MyClient** |
| ClientInvocationContext cic = ClientInvocationContext.*get*();  cic.setTransactionId("FDX-TRANSID-1002");  logger.error("Logging with my Transaction ID"); |
| **Log Output** |
| INFO APP000001 SamplePojo.ProcessRequest [TxId=FDX-TRANSID-1002]Logging with my Transaction ID. |

* **Tracer Bullet**

In a production or performance testing environment, logging should be minimal. In some cases there rises the need to enable logging for troubleshooting purposes. The tracer bullet provides two modes for enhanced transaction logging.

**Minimal –** Logs the request and response for a single transaction regardless of logging level.

**Full –** Logs every call to the Logger for a single transaction regardless of logging level. The Tracer Bullet is turned by setting the logTransaction attribute to either InvocationContext.LOG\_TRANSACTION\_MIN or InvocationContext.LOG\_TRANSACTION\_FULL.

|  |
| --- |
| **Class MyClient** |
| ClientInvocationContext cic = ClientInvocationContext.*get*();  cic.setLogTransaction(InvocationContext.*LOG\_TRANSACTION\_MIN);*  cic.setTransactionId("FDX-9999");  cic.setEndpointServiceAppId("123\_SampleService");  cic.setUserId("123456");  logger.info("Testing INFO with transaction logging set to min); |

# Overriding Request/Response Suffixes

CSR-enabled Web services can employ custom name suffixes for relevant requests and responses. To specify custom request response suffixes modify the bean definition to include a property for both the request and response. By default, CSR services are configured to use “Request” and “Response” as suffixes (example: ratecheckRequest & ratecheckResponse).

The properties, “requestSuffix” and “responseSuffix”, can be modified to allow essentially any arbitrary string. In the below sample, we configure a custom suffix for incoming requests “incoming”. Additionally, we configure two suffixes, “outgoing” and “reply”. When multiple suffixes are configured, either can be used. These suffix overrides are made in the definitionsConfix.xml file of the CSR service.

<bean id="samplebean" class="com.fedex.xyz.common.contract.MyBean">

<property name="requestSuffix">

    <list>

       <value>incoming</value>

      </list>

   </property>

   <property name="responseSuffix">

    <list>

       <value>outgoing</value>

         <value>reply</value>

      </list>

   </property>

</bean>

In addition to updating the definitionsConfig.xml, the new response object will need to be generated (via xjc) and referenced in the endpoint. In this example, the new objects would have the names of “SampleOutgoing” and “SampleReply”.

# Error handling

CSR includes an error handling mechanism and provides some tools for customization of error handling behavior.

## To produce a Soap Fault that includes a custom error message:

1) Create a custom exception class that extends the CSR ClientException class. For example:

package com.fedex.myApplication.exception;

import com.fedex.csr.common.exception.ClientException;

public class InvalidFIDException extends ClientException {

private static final long serialVersionUID = 6383439175614136304L;

private static final String EXCEPTION\_ID = "InvalidFedExIdError";

public InvalidFIDException (String code) {

super(EXCEPTION\_ID, code);

}

public InvalidFIDException (String code, String mssg, Throwable cause) {

super(EXCEPTION\_ID, code, message, cause);

}

public InvalidFIDException (String code, String message) {

super(EXCEPTION\_ID, code, message);

}

public InvalidFIDException (String code, Throwable cause) {

super(EXCEPTION\_ID, code, cause);

}

}

2) Throw the new custom exception type from the POJO. Note: Make sure you don't intercept the exception in the endpoint. For example:

if ((request.getFedExId() == null) || (request.getFedExId().isEmpty())) {

String errorMessage = "FID received is either null or empty.";

throw new InvalidFIDException (ExceptionCodes.INVALID\_FID, errorMessage);

}

In this case, the SOAP fault returned to the client will include the message, "FID received is either null or empty."

## Custom exception mapper

1) In the operation.xml, define a bean for an exception mapper.

<bean id="exceptionMapper" class="com.fedex.mypackage.exception.MyException"> </bean>

2) Also in operations.xml where you define the DefaultOperation, create an exception mapper property referencing the bean you created.

<bean id="myOperation" class="com.fedex.csr.common.operation.DefaultOperation">

<property name="exceptionMapper" ref="exceptionMapper" />

</bean>

3) Then create a class that implements the ExceptionMapper interface and defines a map() method. The map() method takes some exception as an argument. It then determines which of the three client exception types (Transient, UncrecoverableClientError, UncrecoverableInternalError) is most appropriate and returns that type.

public ClientException map(Exception e) {

if (e instanceof TransientClientException) {

TransientClientException tce = new TransientClientException(“TransientClientException”,“ some message”,e);

return tce;

} else if (e instanceof UnrecoverableClientErrorClientException) {

UnrecoverableClientErrorClientException uce = new UnrecoverableClientErrorClientException(“UnrecoverableClientErrorClientException”,“ some message”,e);

return uce;

} else if (e instanceof NumberFormatException) {

UnrecoverableInternalErrorClientException uie = new UnrecoverableInternalErrorClientException(“NumberFormatException”,“some message”,e);

return uie;

}

Additionally, if you create your own custom exception that extends the ClientException abstract class, you can use that in conjunction with a custom ExceptionMapper, returning it instead of the standard set of three ClientExceptions:

if (e instanceof InvalidFIDException) {

InvalidFIDException cce = new InvalidFIDException(“myCustomValidationException”,“some message”,e);

return tce;

}

…and define your custom exception:

public class InvalidFIDException extends ClientException {

private static final long serialVersionUID = 6383439175614136304L;

private static final String EXCEPTION\_ID = "InvalidFedExIdError";

public InvalidFIDException (String code) {

super(EXCEPTION\_ID, code);

}

public InvalidFIDException (String code, String mssg, Throwable cause) {

super(EXCEPTION\_ID, code, message, cause);

}

public InvalidFIDException (String code, String message) {

super(EXCEPTION\_ID, code, message);

}

public InvalidFIDException (String code, Throwable cause) {

super(EXCEPTION\_ID, code, cause);

}

}

Note that the Handler interface used to create custom transaction handlers also contains a handleFault() method that would be overridden. This can also be used to introduce error handling logic. See section 9 for more information.

# Wily Monitoring Enablement

The Common Service Runtime (CSR) includes a configurable ProbeBuilderDirective (PBD) file that can be used in conjunction with CA Wily Introscope. By modifying the standard ProbeBuilder List (PBL) to refer to the included PBD, CSR-enabled web services can generate a set of metrics related to user traffic and performance.

## Prerequisites

Prior to enabling the CSR decorator, the Wily agent must be installed and a domain must be setup. Go to keyword “wily” for more information. Additionally, the ALPS configuration must be modified to support Wily agent startup. You can get more details by contacting Wily group (**Keyword: wily**)

## Usage Details

The included PBD file is located at */opt/fedex/devframework/{framework version}/lib/csr.pbd*. Initially, this file is configured to generate a standard set of metrics on web services. Metrics are accumulated each time a request comes in to the web service. Without modification, these metrics include:

* Average execution time in milliseconds.
* A count of the total number of processed service requests.
* The number of requests currently being processed and not yet completed.
* The number of invocations per interval. This interval will change based on the view period selected in the View pane in the Investigator.
* A count of the total number of processed service requests for each client (by client application ID).

## Technical Details

All metrics are collected by employing a single decorator class in the Common Service Runtime. This class aggregates usage and performance data and provides a single source from which the Wily agent can collect metrics for display in the Wily Investigator console. To enable this feature, perform the following steps:

1. Edit the standard PBL file. This file is included under the Wily installation and will vary depending on the application server. Typical locations are:

{wily home}/current/weblogic-typical.pbl (for a WebLogic server)

{wily home}/current/jboss-typical.pbl (for a JBoss server)

1. Location the “Directives Files” section and add a reference to the PBD file included with the Development Framework. For example:

#######################

# Directives Files

# ================

# One directives file name per line. Relative names

# are resolved against the location of this file.

toggles-typical.pbd

jvm.pbd

j2ee.pbd

java2.pbd

struts.pbd

taglibs.pbd

oraclejdbc.pbd

sqlagent.pbd

errors.pbd

summary-metrics-6.1.pbd

sql-agent-summary-metrics-6.1.pbd

jboss4x.pbd

jsf-toggles-typical.pbd

jsf.pbd

**/opt/fedex/devframework/3.0.1/lib/CSR.pbd**

1. Restart your application server instance and redeploy your web service instance. In the Wily investigator, a section entitled “CSR” will appear in the domain tree and all relevant metrics can be found under that location. Note that metrics will only appear after the web service starts accepting client traffic. For more information on the Wily agent or Investigator, please refer to the documentation at keyword “wily.”

# CSR Fault Barrier

The fault barrier is a perfect example of using the principle of separation of concerns. A cross-cutting set of exception handling functionality is encapsulated in one place – the fault barrier(refer to [Figure 1](#_CSR_Overview)).

If something exceptional happens down in the lower layers of the Service Runtime stack then a non-checked exception is thrown that describes the issue. These exceptions are usually CSR defined classes extending RuntimeExceptions, or Spring framework exceptions.

Sitting up high in the Service Runtime stack is a layer called the fault barrier. The fault barrier has a number of responsibilities;

* it catches every exception thrown below the barrier
* is the only place that catches non-checked exceptions.
* it logs the exception.
* It translates the exception to the appropriate variant of client exceptions

This relieves the code lower down in the stack from these onerous and cluttering tasks, and ensures consistent handling of the exceptions.

Client exceptions are exceptions that are shared with the Service Runtime client. Currently there are 3 variations that all share a common bas class – ClientException:

* TransientClientException – the client is advised to retry the service call as this error is not expected to occur again.
* UnrecoverableClientErrorClientException – there is an error caused by the client and the client is required to fix the error before retrying. E.g. a request validation error.
* UnrecoverableInternalErrorClientException – there is an error in the service. Retrying the service request will be of no avail.

Finally the protocol adapter will take the client exception and map it to the specific protocol. For example, when using SOAP, thrown exceptions are exposed to the client as a SOAP Fault.

For CSR 3.2, a new exception was introduced to the Fault Barrier:

* ResponseMessageException – if this exception is thrown by the endpoint, the fault barrier will process and return a SOAP response (containing the XML response documents) to the client, instead of a SOAP fault. This may be desired behavior in certain situations, when a service owner does not wish to return a SOAP fault to the client.

# Defining Log Level for CSR Client Exceptions (From Framework Release 5.0.0)

Service-User can define Log Level for CSR Client Exceptions.

By defining the below bean in operations.xml service can have its own log level for any CSR Client Exceptions. By default CSR logs Exceptions as ERROR.

<!—

The implementation of Custom Logging Level for Client Exceptions in CSR.

Please define a com.fedex.csr.common.contract.DefaultCustomLoggingClientExceptions' bean with the property 'clientExceptions'. Please provide a map with single or multiple Key and Value pairs for the property 'clientExceptions'.

Accepted Keys in the map are CSR Client Exceptions (CanonicalName). All exception classes must extend com.fedex.csr.common.exception.ClientException class.

Accepted Values in the map are FedExLevel. The Value should be one of these TRACE/DEBUG/INFO/WARN/ERROR/FATAL.

Example Custom Log Level bean.

-->

<bean id="customLoggingExceptions class="com.fedex.csr.common.contract.DefaultCustomLoggingClientExceptions">

<property name="clientExceptions">

<map>

<entry key="com.fedex.csr.common.contract.ValidatorException" value="INFO" />

<entry key="com.fedex.csr.usage.ServiceLevelException" value="WARN" />

</map>

</property>

</bean>

# Framework metrics

For auditing purposes, CSR is designed to send a brief, lightweight SOAP message which is received by the Development Framework Team. This overhead of this function is negligible and will only occur once at start time. If for any reason the SOAP message fails to send, the CSR component will give up, bypass the operation and complete service startup. It will not reattempt this action unless the application is restarted. The content of this “phone-home” message is the version of the framework used and the application ID. By default, this action is enabled and disabling is strongly discouraged.

To disable this function, place the following parameter in the fp.properties file:

send.metrics=off

# REST Feature

Starting in version 7.1.0, CSR can process REST based transactions along with the original SOAP based transactions. It will create a URL for each of the endpoints defined and call the standard CSR stack (all features).

## URL Structure

The URL will be formed from the new servlet context (rest) along with the endpoints defined in operations.xml. For example, if the normal endpoint for Sample Service is /sampleservice, then the new servlet root will be /sampleservice/rest. The endpoints defined in operations.xml shall derive from this new endpoint. Below is an example from Sample Service.

*/sampleservice:* Handles the SOAP transactions.  
*/sampleservice/rest/\*.wadl:* Returns the WADL for the rest services. Able to return specific versions.  
Example: SampleService.v1.wadl  
*/sampleservice/rest/\*.xsd*: Optional, returns the schemas defined based on version of WADL.  
Example: SampleService.v1-SampleService.v1.xsd  
*/sampleservice/rest/showOperations*: Handles the REST showOperations request.  
*/sampleservice/rest/ping*: Handles the REST ping request.  
/sampleservice/rest/Endpoints\*: Handles the endpoints defined in operations.xml.

## Transactions

The transactions sent / received to the REST service will be the same as the body of the existing SOAP transactions. Below is an example.

***SOAP Transaction sent to /sampleservice***

<soapenv:Envelope xmlns:soapenv=[*http://schemas.xmlsoap.org/soap/envelope/*](http://schemas.xmlsoap.org/soap/envelope/) xmlns:com=*"http://xmlns.fedex.com/csr/common"*>  
 <soapenv:Header>  
 <wsse:Security  
 xmlns:wsse=*"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"*>  
 <wsse:UsernameToken  
 xmlns:wsu=*"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"*>  
 <wsse:Username>{Token}</wsse:Username>  
 </wsse:UsernameToken>  
 </wsse:Security>  
 </soapenv:Header>  
 <soapenv:Body>  
 <com:pingRequest majorVersion=*"1"* minorVersion=*"0"* />  
 </soapenv:Body>  
</soapenv:Envelope>

***XML Transaction sent to /sampleservice/rest/ping***

<com:pingRequest majorVersion=*"1"* minorVersion=*"0"* xmlns:com=*"http://xmlns.fedex.com/csr/common"*/>

## Security

The existing SOAP transactions can have the token in the SOAP header or the HTTP header. The REST implementation requires it to be in the HTTP header. The token must be URL encoded.

*Example:*

**import** java.net.URLEncoder;

**import** org.springframework.http.HttpHeaders;

headers.add("X-CSR-SECURITY\_TOKEN",URLEncoder.*encode*(tokenGenerator.getToken(),"UTF-8"));

## Implementation

Implementation of the REST features will require the below entry in the web.xml. The existing servlet used for SOAP can ONLY handle SOAP transactions. This will give a basic WADL without any schemas.  
<servlet>  
 <description>Common Service Runtime REST</description>  
 <servlet-name> spring-rest</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath:spring-rest-servlet.xml</param-value>  
 </init-param>  
 <load-on-startup>1</load-on-startup>  
</servlet>  
  
<servlet-mapping>  
 <servlet-name>spring-rest</servlet-name>  
 <url-pattern>/rest/\*</url-pattern>  
</servlet-mapping>

Version 7.1.0 requires an additional dependency of [Apache Schema Core](http://itg.prod.fedex.com/sf/docman/do/downloadDocument/projects.ea_framework/docman.root.userdocumentation/doc387346/16http:/central.maven.org/maven2/org/apache/ws/xmlschema/xmlschema-core/2.1.0/xmlschema-core-2.1.0.jar). This has been resolved in version 7.1.1.

<dependency>

<groupId>org.apache.ws.xmlschema</groupId>

<artifactId>xmlschema-core</artifactId>

<version>2.1.0</version>

<scope>compile</scope>

<optional>true</optional>

</dependency>

## WADL Support

The REST implementation will generate a WADL to be used by SOAP-UI or other compatible tools. Below is an example of a standard WADL. The document title (SAMPLESERVICE WADL) will be derived from the fp.properties file. This file will be built every time to show the proper URL in the resources base property. This WADL will show ALL endpoints defined in operations.xml.

<?xml version=*"1.0"* encoding=*"UTF-8"* standalone=*"yes"*?>  
<application xmlns=*"http://wadl.dev.java.net/2009/02"*>  
 <doc title=*"SAMPLESERVICE WADL"* />  
 <resources base=*"http://localhost:8080/sampleservice/rest"*>  
 <resource path=*"/showOperations"*>  
 <method name=*"POST"*>  
 <request><representation mediaType=*"application/xml"* /></request>  
 <response status=*"200"*><representation mediaType=*"application/xml"* /></response>  
 <response status=*"500"*><representation mediaType=*"application/xml"* /></response>  
 </method>  
 </resource>  
 <resource path=*"/ping"*>  
 <method name=*"POST"*>  
 <request><representation mediaType=*"application/xml"* /></request>  
 <response status=*"200"*><representation mediaType=*"application/xml"* /></response>  
 <response status=*"500"*><representation mediaType=*"application/xml"* /></response>  
 </method>  
 </resource>  
 <resource path=*"/Rating"*>  
 <method name=*"POST"*>  
 <request><representation mediaType=*"application/xml"* /></request>  
 <response status=*"200"*><representation mediaType=*"application/xml"* /></response>  
 <response status=*"500"*><representation mediaType=*"application/xml"* /></response>  
 </method>  
 </resource>  
 </resources>  
</application>

## Enhance WADL Support

If a special bean is defined by the developer, then additional XSD information will be added to the WADL, and only endpoints that are in the XSDs will be visible. This is similar to the ant bean that is used to generate the WSDL for SOAP transactions. Each bean definition will create a new WADL based on the wadlName parameter. Simply add the additional context file with the bean definitions to the web.xml.

**Web.xml change**

<servlet>  
 <description>Common Service Runtime REST</description>  
 <servlet-name> spring-rest</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath:spring-rest-servlet.xml WEB-INF/mycontext.xml</param-value>  
 </init-param>  
 <load-on-startup>1</load-on-startup>  
</servlet>  
  
<servlet-mapping>  
 <servlet-name>spring-rest</servlet-name>  
 <url-pattern>/rest/\*</url-pattern>  
</servlet-mapping>

***Bean Definition*** (Autowired via Spring)

<!—creates SampleService.v1.wadl -->  
<bean class=*"com.fedex.ea.framework.csr.rest.WadlHelper"*>  
 <property name=*"wadlName"* value=*"SampleService.v1"*/>  
 <property name=*"supportCommonOperations"* value=*"true"* />  
 <property name=*"xsds"*>  
 <list>  
 <value>classpath:contract/xsds/SampleService.v1.xsd</value>  
 </list>  
 </property>  
</bean>

<!—creates SampleService.v2.wadl -->  
<bean class=*"com.fedex.ea.framework.csr.rest.WadlHelper"*>  
 <property name=*"wadlName"* value=*"SampleService.v2"*/>  
 <property name=*"supportCommonOperations"* value=*"false"* />  
 <property name=*"xsds"*>  
 <list>  
 <value>classpath:contract/xsds/SampleService.v2.xsd</value>  
 </list>  
 </property>  
</bean>

***Enhanced WADL***

<?xml version=*"1.0"* encoding=*"UTF-8"* standalone=*"yes"*?>  
<application xmlns=*"http://wadl.dev.java.net/2009/02"*>  
 <doc title=*"SampleService.v2 WADL"* />  
 <grammars>  
 <include href=*"http://localhost:8080/sampleservicecv/rest/SampleService.v2-SampleService.v2.xsd"* />  
 <include href=*"http://localhost:8080/sampleservicecv/rest/SampleService.v2-csrerror\_v1.xsd"* />  
 </grammars>  
 <resources base=*"http://localhost:8080/sampleservicecv/rest"*>  
 <resource path=*"/Sample"*>  
 <method name=*"POST"*>  
 <request>  
 <representation xmlns:ns2=[*http://sample.fedex.com/sampleservice*](http://sample.fedex.com/sampleservice) element=*"ns2:SampleRequest"* mediaType=*"application/xml"* />  
 </request>  
 <response status=*"200"*>  
 <representation xmlns:ns2=[*http://sample.fedex.com/sampleservice*](http://sample.fedex.com/sampleservice) element=*"ns2:SampleResponse"* mediaType=*"application/xml"* />  
 </response>  
 <response status=*"500"*>  
 <representation xmlns:ns2=[*http://framework.ea.fedex.com/csr/exception*](http://framework.ea.fedex.com/csr/exception) element=*"ns2:exception"* mediaType=*"application/xml"* />  
 </response>  
 </method>  
 </resource>  
 </resources>  
</application>

## XSD Support

If the bean described in the previous requirement is defined, then the XSD definitions will also be available from the URL /sampleservice/rest/\*.xsd. Below are examples.  
<http://localhost:8080/sampleservicecv/rest/SampleService.v1-csrerror_v1.xsd>  
<http://localhost:8080/sampleservicecv/rest/SampleService.v1-SampleService.v1.xsd>  
<http://localhost:8080/sampleservicecv/rest/SampleService.v1-common_v1.xsd>

## Exception Handling

If an exception is thrown by the endpoint, then the REST implementation will throw a 500 HTTP code and send an XML representation of the exception. If it is a CSR Client exception, then the code and exception ID will be shown. Below is an example.

<CSRException xmlns=*"http://framework.ea.fedex.com/csr/exception"*>

<code>InvalidClientRequest</code>

<exceptionId>UnrecoverableClientError</exceptionId>

<transactionId>521e47b5-b06f-425e-b1c2-d2b92be2c12f</transactionId>

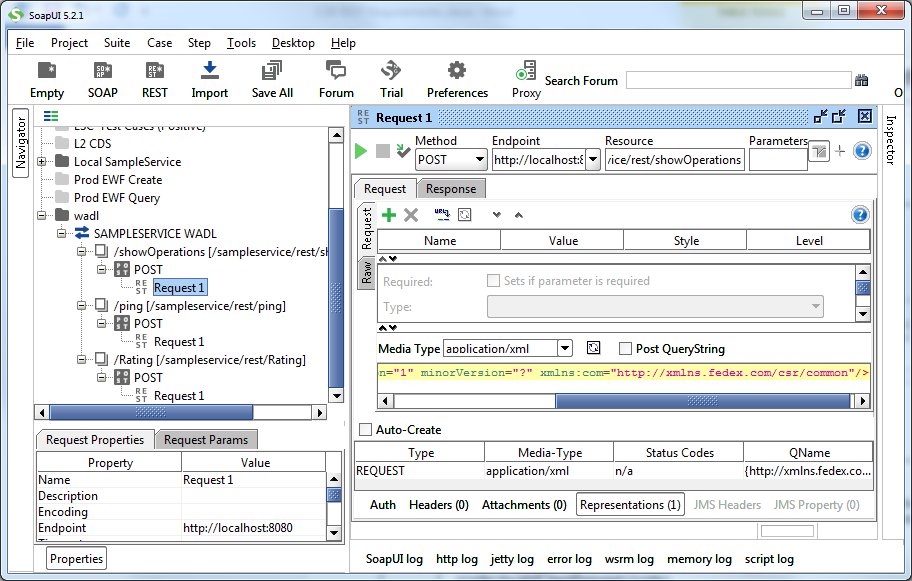
<dateTime>2016-07-26T09:59:36.191-05:00</dateTime>

<message>Security Error: Security Token not found in client request header</message>

</CSRException>

## SOAP UI Support

SOAP UI supports importing WASDL definitions. With the enhanced WADL features, it can automatically create the request object and do validation on the response objects. Below is an example.



## Extendible

The CSR REST implementation will NOT allow the developer to add additional REST endpoints using the Spring controllers annotation. The mapping of the endpoints is locked down and based on operations.xml. Extending the CSR stack is still allowed as described in this document.

# Tomcat 7 Deployment

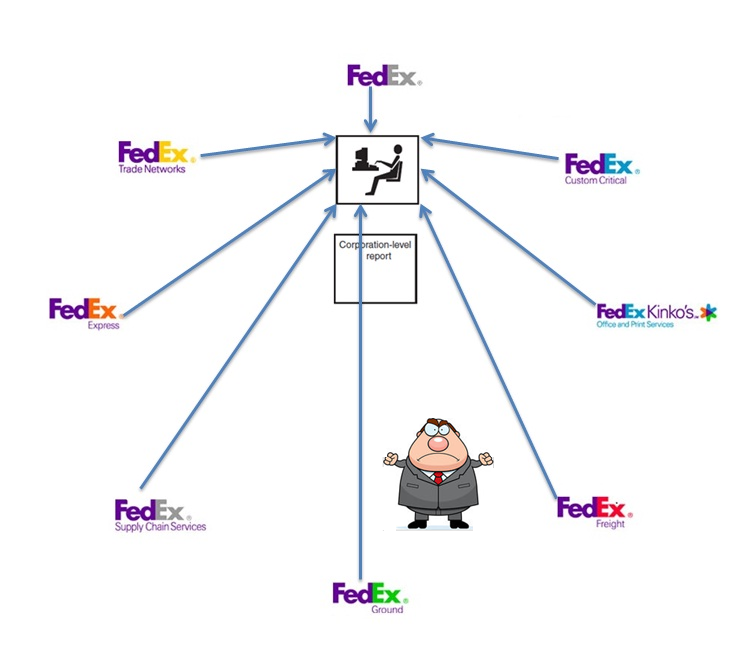
Please include the serializer.jar inside the deployment package (war/ear). The serializer.jar can be found in the C:\cs-framework\5.1.0\extras\csr\lib directory of the local installation of the development framework.

# Appendix A: FedEx Web service Prototype

In this section we will talk about a prototype Web service we want to design for FedEx. This example will help us understand the whole process behind a Web service and all of the components involved to accomplish each request.

FedEx is a corporation that is composed of several different companies i.e. FedEx Express, FedEx Ground, FedEx Office, FedEx Freight, FedEx Custom Critical, FedEx Trade Networks, FedEx SupplyChain, and FedEx Services. Each company of FedEx has its own information technology department depending on their business needs of that individual company and is different and independent of the other company’s IT departments. Each of the FedEx companies has an accounting team who is required to determine how that company is performing. They create reports showing their performance and send them to corporate headquarters each quarter. It is important to note that these reports are formatted differently by each company’s accounting department because they all use different technologies. Corporate accounting department receives these different reports, extracts the information from them, and enters it into a corporate accounting system. This system combines all the company’s information and produces a corporate-level report that shows the corporation’s performance for the quarter.

The Framework user of FedEx is fundamentally dissatisfied with this procedure for determining profitability. His primary concern is that he only gets the information that he needs once per quarter. Now he wants to receive the reports every week.



The corporate accounting department is not happy with the current state of affairs either because the report from each company is in different formats which require them to compile the final report, which is a tedious, error prone and time consuming process.

Several characteristics of this system make it a good candidate for a Web services–based system because of the following reasons:

* The volume of data to be transferred is not extremely large because all data is text based hence performance is not an issue.
* As each company has multiple branches all over the world, so an Internet-based approach is attractive.
* The security concerns associated with allowing HTTP through the firewall are considered manageable if they use a combination of authentication and encryption.
* All the IT departments in the individual companies can use programming languages that they are familiar with to create their servers.
* Most importantly, each company’s IT department will not have to do any modifications to their original IT infrastructure and they can build the Web service upon their current system.

## Designing the Server and Clients

The first step is to decide which computers will serve as the client and the servers. The client is defined as a consumer of services, and the server is defined as the provider. However, two different types of transmissions are commonly used in the type of Web service that we are designing here:

* **Request/Response**
* **Solicit/Response**

With the request/response transmission, the client initiates the action by making a request to the server. In solicit/response transmission, the server makes the first move by soliciting a response from the client. Logically, we could envision this example either way. We could have 8 clients (8 companies) interact with one server. On the other hand, we could create 8 servers and one client. Either approach can be made to work, so this is completely dependent on what sorts of business needs we have and what problems we are trying to solve. Normally, more expertise is required to create a Web service than is required to create a client. The Web service programmer must create, by hand or with tools, both the implementation of the service and the Web services Description Language (WSDL) document. The client, on the other hand, can use the WSDL to generate a good-sized portion of the client code. For this reason, we typically want a design in which the server is defined once and the clients are defined once for each subsidiary.

## Transmission Primitives

The second step that we must take is whether to require the clients to initiate communication with the Web server or to have the server solicit the responses from the client. There is another logical reason to prefer solicit/response transmission style. If we were using a request/response style transmission, what would the server do if no communication were received from a certain client by a certain date or time? It would have to send a message asking the client to provide the missing data. Because this “pull” message must exist anyway to handle the case in which a client hasn’t sent the data, it makes sense to create the system using solicit/response transmission style.

The next logical step in creating a system is the design of the messages that will be exchanged. In our case, we have decided that the corporate computer is going to be our server and that the communications will follow solicit/response transmission style.

This suggests that two messages need to be sent from the server to the clients:

* Is the report ready for uploading?
* Please give me your report.

Likewise, the clients need two different response messages:

* Yes, the report is ready. (Or no, it is not ready.)
* Here is my report.

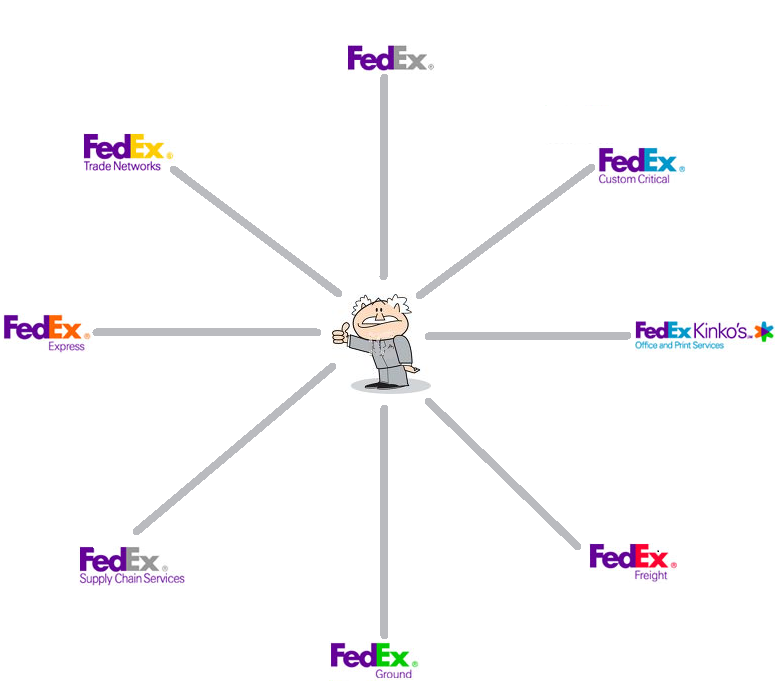
The usage of the system would be fairly simple. On a certain date each month, the server solicits the report from each client by sending out the “Is the report ready?” message. If it is ready, the client answers “Yes.” Upon receiving the “Yes” response, the server sends a “Please give me your report” message. Following that, the client gathers all the data and sends the “Here is my report” message containing the report. If the client answers “No” because the report is not ready, a time for the next retry is assigned and the server sleeps until it is time to try it again. Now that all the reports are solicited and received in an electronic version that follows a specific format, the amount of work required to produce weekly reports is greatly reduced.

## System Design

All that is left is to actually program the parts. The details of the coding process will depend on what tools you choose to implement with. One of the big advantages of Web services is the fact that every one of the 8 different accounting systems can use a different software development tool to create its client code. In addition, the logic in each client will be different because it has to obtain its data from its own accounting system. We will create our example by following these steps:

1. The first step in the design is to write the server software. The WDSL document is then created from the server software, either by hand or automatically.
2. The WSDL is then sent to the programming staff in each subsidiary. Using this document as a guide, they create the client program either by hand or using the development tool that fits in best with their programming language background and interfaces best with their legacy system.
3. As each of the individual subsidiaries completes its work, it is run against a special test version of the Web service.
4. When all the subsidiaries complete their work, the whole system is tested and the results are compared to data that is known to be correct.

Below is the new design of the reporting system which is more efficient and the Framework user is receiving the reports every week on time and all of this was possible because of the Web services.



# Appendix B: CSR Service Layer

Detailed design for a portion of the Service layer of CSR is outlined in this section. It includes all the design that is common to every service operation, and does not include the design that is specific to service operations.

The Service Layer supports the following:

1. Mapping XML Requests to Operations
2. Executing Common Rules
3. Executing the Operation
4. Expanding the Service with Decorators

## Detailed Design: Spring Beans

The Spring Bean configuration for the service layer is found in src/main/config/spring-runtime.xml and is bundled as part of CommonServiceRuntime.jar

**Service**

The entry point into the service layer is ServiceImpl (package com.fedex.csr.service). Every protocol layer calls this same class for service execution, so this class is common between all service adapters. This class is lightweight, delegating work to the other classes described below. The ServiceImpl invoke() method processes all requests. This method is marked as @Transactional:

@Transactional(propagation = Propagation.REQUIRED, rollbackFor = Exception.**class**)

**public** Document invoke(Document requestDocument) { …

**Mapping XML Requests to Operations**

The primary class used to map operations is the OperationRegistry (package com.fedex.csr.service). OperationRegistry makes use of the Spring BeanPostProcessor interface. Whenever spring beans are loaded at startup, the BeanPostProcessor listens for beans of a specific type. In this case the OperationRegistry looks for beans with the Operation interface. The supported QNames (qualified XML names) are retrieved from the Operation and used as a key for the lookup HashMap. Therefore the OperationRegistry knows about all QNames that can be handled by the Service and the mapping from QName to the Operation.

Java’s javax.xml.QName is a qualified XML name containing the namespace, local part, and prefix. Typically, the prefix is ignored in CSR configuration and every request should have both a namespace and local part. The local part will be the root element for the request.

### Executing Common Rules

Rules are executed that are common to all operations. The two primary rules include one for security and one for governance. The setup of these rules is found in the com.fedex.csr.rules package.

**Executing the Operation**

Once rule processing is complete the 'execute' method on the Operation is called.

**Expanding the Service with Decorators**

The ServiceImpl class can be decorated to extend its abilities. The FaultBarrierServiceDecorator provides a location for catching, logging, and mapping exceptions.

The WileyServiceDecorator gathers some information when instrumenting with the Wiley tool.



Figure . CSR Service Layer Class Diagram

# Appendix C: Glossary

|  |  |
| --- | --- |
| **Term** | **Definition** |
| CSR | Common Service Runtime |
| WSDL | Web service Definition Language |
| HTTP | Hyper Text Transfer Protocol |
| SOAP | Simple Object Access Protocol |