Progress FUSE

FUSE™ Services Framework

Getting Started Developing Services

[DRAFT]

PROGRESS SOFTWARE

Getting Started Developing Services

Publication date 17 Jul 2009

Legal Notices

Table of Contents

List of Figures

List of Examples

Chapter 1. Introduction to FUSE Services Framework

Overview

FUSE Services Framework is an open source services framework based on Apache CXF[1]. FUSE Services Framework provides a small footprint engine for creating reusable services as part of a integration solution. You can use FUSE Services Framework to service-enable new and legacy applications in an enterprise integration infrastructure.

With FUSE Services Framework you can build and develop services using a variety of container servers, languages, messaging systems, and protocols. The flexible deployment model of FUSE Services Framework supports standalone deployment and deployment in lightweight containers such as Apache Tomcat, FUSE ESB, Spring-based, and J2EE.

Features

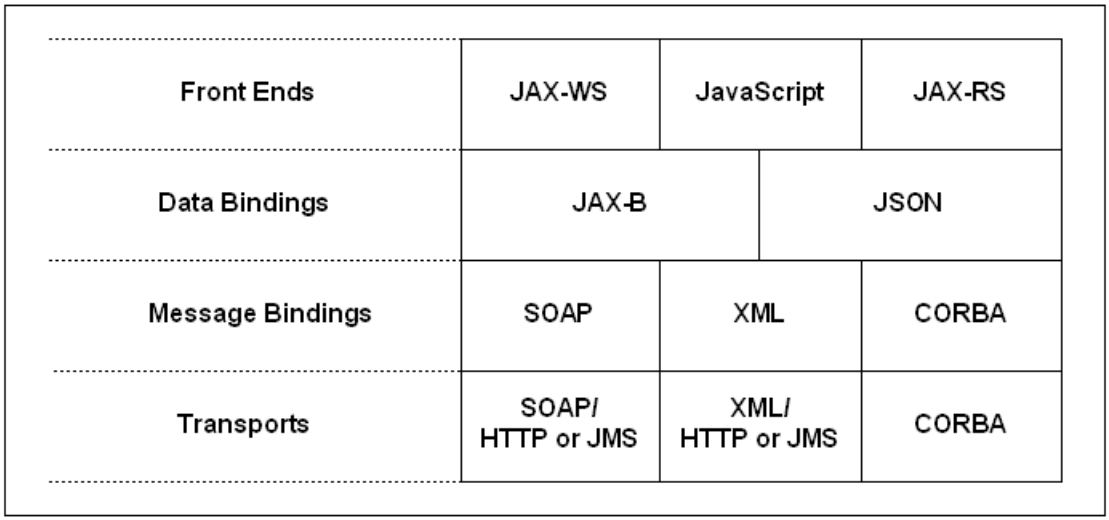
Key features of FUSE Services Framework include the following:

- Support for web services standards — FUSE Services Framework supports a variety of web service standards including SOAP, WSDL, WS-Addressing, WS-Policy, WS-ReliableMessaging, and WS-Security.
- Support for various front ends — FUSE Services Framework supports a variety of front end programming options, including JAX-WS web services and RESTful services, and implements the JAX-WS and JAX-RS APIs. FUSE Services Framework supports both contract first development with WSDL and code first development starting from Java.
- Support for binary and legacy protocols — FUSE Services Framework supports multiple protocols including SOAP, XML, HTTP, and RESTful HTTP, and works over a variety of transports such as HTTP/S and JMS. FUSE Services Framework provides a pluggable architecture that supports both XML and non-XML type bindings, such as JSON and CORBA, in combination with any type of transport.
- Ease of use — FUSE Services Framework provides simple APIs to quickly build code-first services, Maven plug-ins to make tooling integration easy, JAX-WS and JAX-RS API support, and Spring 2.0 XML for easy configuration.

[1] http://cxf.apache.org/

Figure 1.1 summarizes the options that FUSE Services Framework supports for your front ends, data bindings, messages bindings, and transports:

Figure 1.1. FUSE Services Framework service development options



FUSE Integration Designer

FUSE Integration Designer is an Eclipse based development environment you can use to create web services based on FUSE Services Framework. FUSE Integration Designer includes a wizard to assist you in:

- Creating a new web service
- Configuring a web service for deployment
- Deploying a web service to a server

When your web service is deployed, the wizard assists you in generating the client proxy and a sample application to test the web service. When you have completed testing, you can publish your web service to a UDDI business registry using an export wizard.

See the Fuse Integration Designer [2] product web page for information about installing and working with FUSE Integration Designer.

[2] http://fusesource.com/products/fuse-integration-designer/

Front end options

Front ends provide a programming model to interact with FUSE Services Framework. A front end provides functionality through interceptors that are added to services and endpoints.

FUSE Services Framework enables you to create your front end using any of the following options:

- **JAX-WS** — Develop your services using either a Java-first or WSDL-first development approach.
  - See "Developing Web Services" on page 15 for information.
  - See "WSDL-first JAX-WS Service Development" on page 29 and "Java-First JAX-WS Service Development" on page 39 for samples that use this development approach.
- **JAX-RS** — Develop RESTful services using the JAX-RS APIs.
  - See "Developing RESTful Services" on page 19 for information.
  - See "WSDL-first JAX-WS Service Development" on page 29 for a sample that uses this development approach.
- **JavaScript** — Write your services in JavaScript, using FUSE Services Framework code generation tools to produce proxy code and support classes.
  - See "Developing Services in JavaScript" on page 23 for information.

Supported data bindings

Data bindings implement the mapping between XML and Java by converting data to and from XML. Data bindings can also produce XML schema and provide support for wsdl2java code generation, although not all data bindings support all of this functionality.

In FUSE Services Framework, data binding components are responsible for mapping between XML and Java objects. Each data binding implements a particular discipline for mapping, such as JAXB or XML Beans.

There are three parts to a data binding:

- Mapping the live data as it comes into and out of services
- Providing XML schema based on Java objects for dynamic ?wsdl URLs and java2ws

- Generating Java code from WSDL for wsdl2java and dynamic clients

All data bindings provide the live data mapping. The other two parts are optional.

Your choice of data binding is determined by the front end programming option you use. FUSE Services Framework supports multiple data bindings, including:

- **JAXB** — The default data binding, used with JAX-WS front ends. See "JAXB data bindings" on page 17.
- **JSON** — Used with JAX-RS and JavaScript front ends. See "JSON data bindings" on page 20.

See "Basic Data Binding Concepts" in *Developing Applications Using JAX-WS* for information about using data bindings with FUSE Services Framework.

Supported message bindings

Message bindings map a service's messages to a particular protocol. FUSE Services Framework supports the following message bindings:

- **SOAP** — This is the default binding. It maps messages to SOAP and can be used with the various WS-* modules inside FUSE Services Framework.
- **XML** — The pure XML binding avoids serialization of a SOAP envelope and just sends a raw XML message.
- **HTTP** — Maps a service to HTTP using RESTful semantics.
- **CORBA** — Maps messages from CORBA services. CORBA bindings are described using a variety of WSDL extensibility elements within the WSDL binding element. In most cases, the CORBA binding description is generated automatically using the wsdl2corba utility. Usually, it is unnecessary to modify generated CORBA bindings.

See Bindings in *Using the Bindings and Transports* for information about using message bindings with FUSE Services Framework.

Supported transports

FUSE Services Framework uses a transport abstraction layer to hide transport-specific details from the binding and front end layers. FUSE Services Framework supports SOAP or XML over HTTP or JMS, and CORBA transports.

See Transports in *Using the Bindings and Transports* for information about using transports with FUSE Services Framework.

Deploying services

You can deploy services developed with FUSE Services Framework to the following containers:

- **OSGi** — Once installed in an OSGi container, applications can use many of the advanced FUSE features.
- **Spring** — You can deploy any Spring-based application into a Spring container, including a FUSE Services Framework service endpoint.
- **Servlet** — You can deploy and run a FUSE Services Framework endpoint in any servlet container.

See *Configuring and Deploying Endpoints* for information about these service deployment options.

Getting started

"Running the FUSE Services Framework Samples" on page 25 describes some of the samples available to help you get started developing services with FUSE Services Framework:

- "WSDL-first JAX-WS Service Development" on page 29 describes a sample web service developed with a WSDL-first approach. This sample includes a configuration file that enables schema validation.
- "Java-First JAX-WS Service Development" on page 39 describes a sample web service developed with a Java-first approach.
- "JAX-RS Service Development" on page 44 describes a sample RESTful service developed with JAX-RS.

Chapter 2. Developing Web Services

Overview

With FUSE Services Framework you can develop your web services with JAX-WS using either a Java-first or WSDL-first approach to development:

- **Java-first** — Considered easier to create and favored for tactical integrations. Java-first services use Java annotations in the code. WSDL and XSD artifacts are generated on-the-fly. See "Java-first development with JAX-WS" for more information.
- **WSDL-first** — Preferred for strategic service-oriented architecture (SOA). WSDL-first services tend to be modular, platform agnostic, and have better attention to versioning. See "WSDL-first development with JAX-WS" for more information.

Service-oriented design abstracts data into a common exchange format, typically an XML grammar defined in XML Schema. The JAX-WS specification calls for XML Schema types to be marshaled into Java objects, in accordance with the Java Architecture for XML Binding (JAXB) specification. JAXB defines bindings for mapping between XML Schema constructs and Java objects, and defines rules for how to marshal the data. It also defines an extensive customization framework for controlling how data is handled. See "JAXB data bindings" on page 17 for more information.

Java-first development with JAX-WS

You can develop your services from Java code using the JAX-WS APIs, bypassing the WSDL contract. The code can be a class, or classes, from a legacy application that is being upgraded. It can also be a class that is currently used as part of a non-distributed application with features that you want to use in a distributed manner. To use the class, you annotate the Java code and generate a WSDL document from the annotated code. If you do not want to work with WSDL at all, you can create the entire application without ever generating WSDL.

You might have Java code that already implements a set of functionality that you want to expose as part of a service oriented application, or you might want to avoid using WSDL to define your interface. Using JAX-WS annotations, you can add the information required to service enable a Java class. You can also create a Service Endpoint Interface (SEI) that can be used in place of a WSDL contract. If you want a WSDL contract, FUSE Services Framework provides tools to generate a contract from annotated Java code.

See the following references for more information:

- "Java-First JAX-WS Service Development" on page 39 explains how to work with a sample using Java-first service development.
- *Developing Applications Using JAX-WS* provides information about Java-first front end programming using JAX-WS.
- *Tool Reference* describes the code generation tools available with FUSE Services Framework, including the java2ws tool.
- The use case "Creating and Hosting a Web Service" in the *Logisticx Tutorial Guide* [1] examines sample code that uses notations and an SEI in place of a WSDL contract.

WSDL-first development with JAX-WS

Another way to develop services is to start with a WSDL contract. The WSDL contract provides an implementation-neutral way of defining the operations a service exposes and the data that is exchanged with the service. FUSE Services Framework provides tools to generate JAX-WS annotated code from a WSDL contract. The code generators create all of the classes necessary to implement any abstract data types defined in the contract.

In the top-down method of developing a service provider, you start from a WSDL document that defines the operations and methods the service provider will implement. Using the WSDL document, you generate starting point code for the service provider. Adding the business logic to the generated code is done using Java programming APIs.

See the following references for more information:

- "WSDL-first JAX-WS Service Development" on page 29 explains how to work with a sample using WSDL-first service development.
- *Developing Applications Using JAX-WS* provides information about WSDL-first front end programming using JAX-WS.
- *Tool Reference* describes the code generation tools available with FUSE Services Framework, including wsdl2java and wsdl2js tools.

[1] http://fusesource.com/docs/logistix/index.html

JAXB data bindings

You use Java Architecture for XML Binding (JAXB) data bindings with JAX-WS front ends. JAXB is the default data binding for FUSE Services Framework. If you do not specify different data bindings, you will automatically get JAXB data bindings.

JAXB enables you to store and retrieve data in memory in any XML format, without the need to implement a specific set of XML loading and saving routines for the program's class structure. JAXB allows you to map Java classes to XML representations, enabling you to:

- Marshal Java objects into XML
- Unmarshal XML back into Java objects

JAXB is particularly useful when your service specification is complex and changing. Regularly changing the XML Schema definitions to keep them synchronized with the Java definitions can be time consuming and error prone.

JAXB uses Java annotation combined with files found on the classpath to build the mapping between XML and Java. JAXB supports both code-first and schema-first programming. Schema-first programming supports the ability to create a client proxy, dynamically, at runtime.

See the following references for information about JAXB data bindings:

- "Basic Data Binding Concepts" in *Developing Applications Using JAX-WS* provides information about using data bindings with FUSE Services Framework.
- JAXB section[2] of the Apache CXF User's Guide[3] provides additional information about JAXB data bindings.

Transports

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

See *Using the Bindings and Transports* for information about using transports with FUSE Services Framework.

[2] http://cwiki.apache.org/CXF20DOC/jaxb.html
[3] http://cwiki.apache.org/CXF20DOC/index.html

Chapter 3. Developing RESTful Services

RESTful services

The Representational State Transfer (REST) architectural style has a stateless client-server architecture in which web services are treated as resources that can be identified by their URLs. Web service clients can access these resources using a small set of remote methods that describe the action to be performed on the resource. REST style systems can take full advantage of the scalability features of HTTP such as caching and proxies.

In REST, servers expose resources using a URI, and clients access these resources using the four HTTP verbs (GET, POST, PUT, and DELETE). As clients receive representations of a resource, they are placed in a state. When they access a new resource, typically by following a link, they change, or transition, their state.

REST systems are highly scalable and highly flexible. With REST systems, clients are less affected by changes to servers because:

- Resources are accessed and manipulated using the four HTTP verbs
- Resources are exposed using a URI
- Resources are represented using standard grammars

The existing web architecture is an example of a system designed on REST principles. Web browsers act as clients accessing resources hosted on Web servers. The resources are represented using HTML or XML grammars that all Web browsers can consume. The browsers can also easily follow the links to new resources.

JAX-RS front end development

FUSE Services Framework supports JAX-RS (JSR-311), the Java API for RESTful Web Services according to the REST architectural style. JAX-RS provides a standard way to build RESTful services in Java, using annotations to simplify the development and deployment of web service clients and endpoints.

Some features of using JAX-RS to develop RESTful services are:

- The URI bindings are local to the resource beans, which can be arbitrarily nested. This feature simplifies refactoring.
- The loose coupling between the objects returned by the resource methods and providers is clean, making it easy to drop in support for new representations without changing the code of your resource beans (or controllers). Instead, you can just modify an annotation.
- Static typing can be useful when binding URIs and parameters to your controller, for example, having parameters and String, integer, and Date fields prevents you from having to explicitly convert things in your controller.

See the following references for more information:

- "JAX-RS Service Development" on page 44 explains how to work with a sample using JAX-RS service development.
- *Developing RESTful Services* provides information about JAX-RS front end programming.
- The Java web site for the JAX-RS specification[1] provides information about the Java API for RESTful Web Services.

JSON data bindings

You use JavaScript Object Notation (JSON) bindings with JAX-RS front ends. JSON is a lightweight data format for data exchange. It is a text-based, human-readable format for representing simple data structures and associative arrays (called objects).

An advantage of JSON is that is very easy for JavaScript developers to use, because once it is evaluated it immediately becomes a JavaScript object. JSON is supported through Jettison, a StAX implementation that reads and writes JSON. Jettison intercepts calls to read/write XML and instead read/writes JSON.

For information about using JSON data bindings, see the JSON web site[2] and the JSON section[3] of the Apache CXF User's Guide[4].

[1] http://jcp.org/en/jsr/detail?id=311
[2] http://json.org/
[3] http://cwiki.apache.org/CXF20DOC/json-support.html
[4] http://cwiki.apache.org/CXF20DOC/index.html

Transports

HTTP is the underlying transport for the Web. It provides a standardized, robust, and flexible platform for communicating between endpoints. Because of these factors it is integral to RESTful architectures. See *Using the Bindings and Transports* for information about using transports with FUSE Services Framework.

Chapter 4. Developing Services in JavaScript

Overview

JavaScript is a popular dynamic and lightweight programming language that enables developers to quickly create functionality that runs on a large number of platforms. FUSE Services Framework supports both services and clients in JavaScript. FUSE Services Framework embeds Rhino, which provides a friendly programming environment for JavaScript implementations of services. FUSE Services Framework also provides a generator that produces clients for web services with SOAP bindings.

Service development

The pattern used to develop services written in JavaScript is similar to JAX-WS provider implementations that handle requests and responses (either SOAP messages or SOAP payloads) as DOM documents.

Writing a service in JavaScript requires that you define the JAX-WS style metadata and implement the service's business logic. Java providers typically use Java annotations to specify JAX-WS metadata. Since JavaScript does not support annotations, you use ordinary JavaScript variables to specify metadata for JavaScript implementations. FUSE Services Framework treats any JavaScript variable in your code whose name equals or begins with `WebServiceProvider` as a JAX-WS metadata variable.

FUSE Services Framework code generation tools

FUSE Services Framework provides tools for writing service consumers in JavaScript. These tools enable you to generate code from existing applications and download JavaScript from FUSE Services Framework-based services. JavaScript client-side support allows you to create JavaScript service consumers that can communicate natively with SOAP/HTTP service providers. The code generators produce proxy code and support classes for communicating directly with a service provider. Using the generated code, you can use JavaScript to build web applications that access the back end services. The consumers use asynchronous communication to access the services, making interaction as smooth as possible.

See the following references for more information:

- *Developing Applications with JavaScript* provides information about JavaScript front end programming.
- *Tool Reference* describes the code generation tools available with FUSE Services Framework.
- "Basic Data Binding Concepts" in *Developing Applications Using JAX-WS* provides information about using data bindings with FUSE Services Framework, including the wsdl2js tool.
- Bindings in *Using the Bindings and Transports* provides information about using message bindings with FUSE Services Framework.
- *Using the Bindings and Transports* provides information about using transports with FUSE Services Framework.

Chapter 5. Running the FUSE Services Framework Samples

The following sections describe how to set up your environment and run the WSDL-first, Java-first, and JSX-RS service development samples provided with FUSE Services Framework.

Before Running the Samples

Overview

The following sections summarize the required software and setup steps for working with the FUSE Services Framework samples. Refer to the Installing FUSE Services Framework[1] for detailed installation and setup instructions.

Requirements

To work with the FUSE Services Framework samples as described in the following sections, you must have an active internet connection. These instructions use the Maven build engine, which connects to one or more Maven repositories on the internet to download JAR files that are determined to be dependencies of the current build.

If you prefer to run the samples using Apache Ant, wsdl2java, javac, or java, see the ReadMe files for each sample for instructions. Also see the top level ReadMe file in the `install_dir\samples` directory for additional setup requirements common to all the samples.

To work with the FUSE Services Framework samples, you must have the following software installed:

- FUSE ESB 3.3.0.6 or higher — See "Installing FUSE ESB" .
- FUSE Services Framework 2.2 or higher — See "Installing FUSE Services Framework" .
- Apache Maven 2.0.6 or higher — See "Installing and setting up Maven" .
- A Java 5 or Java 6 JDK — See the Sun Java Downloads[2] web site for the downloadable files and installation instructions.

Refer to Installing FUSE Services Framework[3] for detailed instructions on installing and setting up the required software.

[1] http://fusesource.com/docs/framework/2.1/install_guide/install_guide.pdf
[2] http://java.sun.com/javase/downloads/index.jsp
[3] http://fusesource.com/docs/framework/2.1/install_guide/install_guide.pdf

Installing FUSE ESB

Download FUSE ESB 3.4.x or FUSE ESB 4.0.x from the fusesource[4] downloads page.

The instructions for installing your version of FUSE ESB are available at the *FUSE Documentation*[5] web site.

Installing FUSE Services Framework

Installing FUSE Services Framework also installs the `samples` directory, which contains sub-directories for all the samples.

Download FUSE Services Framework 2.x from the fusesource[6] downloads page.

The instructions for installing FUSE Services Framework are available at the *FUSE Documentation*[7] web site.

Installing and setting up Maven

Download and install Maven. See the Apache Maven Project[8] web site for the downloadable files and installation instructions.

After installing Maven, you must change the following settings in your operating system environment:

1. Set the `M2_HOME` environment variable to point to the Maven root directory.
2. Add the Maven `bin` directory to your PATH:
   - On Windows: `%M2_HOME%\bin`
   - On UNIX: `$M2_HOME/bin`

[4] http://fusesource.com/downloads
[5] http://fusesource.com/documentation/fuse-esb-documentation/
[6] http://fusesource.com/downloads
[7] http://fusesource.com/documentation/
[8] http://maven.apache.org/

About Maven

Maven is a project management tool that encompasses a Project Object Model (POM), a set of standards, a project lifecycle, a dependency management system, and logic for executing plugin goals at defined phases in a lifecycle.

A POM is an XML file containing information about the project and configuration details. Configuration information can include project dependencies, plugins or goals to be executed, project versions, and so on. Maven uses the configuration information in this file to build the project.

The POMs, along with dependencies that are downloaded and artifacts that are generated when you run Maven commands, are stored on your system in a Maven repository. This repository is located:

- On Windows, in:
  `...\Documents and Settings\username\.m2\repository\`
- On UNIX, in: `.../home/username/.m2/repository/`

POMs and generated resources for the FUSE Services Framework samples are stored in the Maven `\repository` directory in:
`\org\apache\cxf\samples\sample_name\1.0\`.

See the Apache Maven documentation[9] for more information.

[9] http://maven.apache.org/guides/index.html

WSDL-first JAX-WS Service Development

Overview

The WSDL-first JAX-WS sample is a simple Hello World application developed using the JAX-WS APIs. The sample demonstrates using a WSDL-first development approach to:

- Run a simple client against a standalone server using SOAP 1.1 over HTTP
- Configure FUSE Services Framework to enable schema validation on the client and/or server side

This sample includes a WSDL file that defines the operations and the data exchanged with the service. When you run this sample, the client and server applications send greetings back and forth. The client sends one of the messages with an invalid length string, which causes the server to throw an exception defined in the WSDL file.

When you enable schema validation, the server and client throw marshaling and unmarshaling exceptions defined in a configuration file that you add to your `CLASSPATH` (see "Enabling schema validation" on page 33).

What happens when you run the sample

The `hello_world.wsdl` file for this sample defines four operations for the service:

Example 5.1. WSDL-First Sample: hello_world.wsdl

```
<wsdl:portType name="Greeter">
   <wsdl:operation name="sayHi">
      <wsdl:input message="tns:sayHiRequest" name="sayHiRequest"/>
      <wsdl:output message="tns:sayHiResponse" name="sayHiResponse"/>
   </wsdl:operation>
   <wsdl:operation name="greetMe">
      <wsdl:input message="tns:greetMeRequest" name="greetMeRequest"/>
      <wsdl:output message="tns:greetMeResponse" name="greetMeResponse"/>
   </wsdl:operation>
   <wsdl:operation name="greetMeOneWay">
      <wsdl:input message="tns:greetMeOneWayRequest" name="greetMeOneWayRequest"/>
   </wsdl:operation>
   <wsdl:operation name="pingMe">
      <wsdl:input name="pingMeRequest" message="tns:pingMeRequest"/>
      <wsdl:output name="pingMeResponse" message="tns:pingMeResponse"/>
      <wsdl:fault name="pingMeFault" message="tns:pingMeFault"/>
   </wsdl:operation>
   </wsdl:portType>
```

The client and server do the following when you run the sample:

1. The client invokes `sayHi` and the server responds.
2. The client invokes `greetMe` and the server responds.
3. The client invokes `greetMe` with an invalid length string and expects an error. The server executes `pingMe` and throws a `PingMeFault` exception.
4. The client invokes `greetMeOneWay` and expects no response from the server.

Using Maven

These instructions use Maven to build and run the sample, using the `pom.xml` file located the base directory of this sample. If you prefer to use wsdl2java, javac, and java to build and run the sample, see the ReadMe file located in your `install_dir\samples\wsdl_first` directory.

See "About Maven" on page 28 and "Installing and setting up Maven" on page 27 for more information about using Maven with the FUSE Services Framework samples.

Building and running the sample

Initially, you will run the sample without using a configuration file. Later you will add the file, `cxf.xml`, to the `CLASSPATH` to run the sample with schema validation.

To build and run the sample:

1. Open a console to the sample directory, `install_dir\samples\wsdl_first` and enter the following command to build the sample:

```
mvn install
```

Maven builds the sample and downloads required JAR files to your Maven repository:

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed - org.apache.cxf.samples:wsdl_first:jar:1.0
[INFO]    task-segment: [install]
[INFO] ------------------------------------------------------------------------
...
[INFO] Installing install_dir\samples\wsdl_first\target\wsdl_first-1.0.jar to
...\.m2\repository\org\apache\cxf\samples\wsdl_first\1.0\wsdl_first-1.0.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
...
```

2. Enter the following command to start the server:

   ```
   mvn -Pserver
   ```

   When the server is started, the console displays the message `Server ready....`

3. Open another console at the same location and enter the following command to start the client:

   ```
   mvn -Pclient
   ```

   The client starts and sends a greeting to the server. The client and server consoles show the messages sent back and forth as the sample runs.

Examining the output

The messages appear rapidly in the consoles as the sample runs. When the messages have all been sent, you can scroll up in the console windows to observe the following events:

1. The client invokes `sayHi`, and receives a response from the server:

   ```
   Client Console
   Invoking sayHi...
   Server responded with: Bonjour
   ```

Messages in the server console indicate that the server executes the `sayHi` operation:

```
Server Console
Mar 23, 2009 9:45:57 AM demo.hw.server.GreeterImpl sayHi
INFO: Executing operation sayHi
Executing operation sayHi
```

2. The client (in this example, `Bob`) then invokes `greetMe` and receives a response from the server:

```
Client Console
Invoking greetMe...
Server responded with: Hello Bob
```

Messages in the server console indicate that the server executes the `greetMe` operation:

```
Server Console
... demo.hw.server.GreeterImpl greetMe
INFO: Executing operation greetMe
Executing operation greetMe
Message received: Bob
```

3. The client invokes `greetMe` with an invalid length string and expects an exception:

```
Client Console
Invoking greetMe with invalid length string, expecting excep
tion...
```

The server executes `pingMe` and throws a `PingMeFault` exception:

```
Server Console
...demo.hw.server.GreeterImpl greetMe
INFO: Executing operation greetMe
Executing operation greetMe
Message received: Invoking greetMe with invalid length string, expecting exception...

...demo.hw.server.GreeterImpl pingMe
INFO: Executing operation pingMe, throwing PingMeFault exception
Executing operation pingMe, throwing PingMeFault exception
```

4. The client invokes `greetMeOneWay` and gets no response because this is a one-way operation:

```
Client Console
Invoking greetMeOneWay...
No response from server as method is OneWay
```

5. The client invokes `pingMe` and expects an exception:

```
Client Console
Invoking pingMe, expecting exception...
Expected exception: PingMeFault has occurred: PingMeFault raised by server
FaultDetail major:2
FaultDetail minor:1
```

The server throws an exception, executes `greetMeOneWay`, and exits:

```
Server Console
...org.apache.cxf.phase.PhaseInterceptorChain doIntercept
INFO: Application has thrown exception, unwinding now: org.apache.hello_world_soap_http.Ping
MeFault:
PingMeFault raised by server
...demo.hw.server.GreeterImpl greetMeOneWay
INFO: Executing operation greetMeOneWay
Executing operation greetMeOneWay

Hello there Bob
```

To remove the code generated from the WSDL file and the `*.class` files, run `mvn clean`.

Enabling schema validation

By default, message parameters are not validated, but you can use a configuration file to enable message parameter validation. This sample includes a configuration file, `cxf.xml`, that changes the default behavior to enable schema validation on the client proxy and server endpoint:

- A JAX-WS client proxy created for the port `{http://apache.org/hello_world_soap_http}SoapPort` will have schema validation enabled.

  The `cxf.xml` configuration file includes the following bean to enable schema validation on the client:

Example 5.2. Schema Validation on the Client

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

- A JAX-WS server endpoint created for the port `{http://apache.org/hello_world_soap_http}SoapPort` will have schema validation enabled.

  The `cxf.xml` configuration file includes the following bean to enable schema validation on the server endpoint:

Example 5.3. Schema Validation on the Server Endpoint

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

When you run this sample with the `cxf.xml` configuration file added to the `CLASSPATH`, the client's second `greetMe` invocation causes an exception (a marshaling error) on the client side. This exception occurs before the request with the invalid parameter is sent.

You can comment out the definition of the `jaxws:client` element in the `cxf.xml` configuration file, then run the sample again to observe that the client's second `greetMe` invocation still throws an exception. This time the exception is caused by an unmarshaling error on the server side.

You can comment out both elements, or remove the `cxf.xml` file from your `CLASSPATH`, to restore the default behavior.

Running the sample with schema validation

To run the sample with schema validation.

1. Add `cxf.xml` to your `CLASSPATH` environment variable.

2. Start the server and client again. After the sample runs, scroll through the server and client consoles to observe the messages exchanged.

This time, when the client invokes `greetMe` with an invalid length string, the client catches an WebServiceException before the message is sent, and generates a marshaling error:

```
Client Console
Caught expected WebServiceException:
Marshalling Error: cvc-maxLength-valid: Value 'Invoking greetMe with invalid
length string, expecting exception...' with length = '67' is not facet-valid
with respect to maxLength '30' for type 'MyStringType'.
```

3. Edit the configuration file to comment out the definition of the `<jaxws:client>`. When you run the sample this time, the second `greetMe` invocation throws an exception that is caused by an unmarshaling error on the server side:

```
Server Console
...org.apache.cxf.phase.PhaseInterceptorChain doIntercept
INFO: Interceptor has thrown exception, unwinding now
org.apache.cxf.interceptor.Fault: Unmarshalling Error: cvc-maxLength-valid:
Value 'Invoking greetMe with invalid length string, expecting exception...' with
length = '67' is not facet-valid with respect to maxLength '30' for type 'MyStringType'.
...
Caused by: javax.xml.bind.UnmarshalException
- with linked exception:
[org.xml.sax.SAXParseException: cvc-maxLength-valid: Value 'Invoking greetMe with
invalid length string, expecting exception...' with length = '67' is not facet
-valid with respect to maxLength '30' for type 'MyStringType'.]
```

Messages in the client console show the that the WebServiceException is caught and an unmarshalling error is generated:

```
Client Console
Invoking greetMe with invalid length string, expecting exception...
Caught expected WebServiceException:
Unmarshalling Error: cvc-maxLength-valid: Value 'Invoking greetMe with invalid length string,

expecting exception...' with length = '67' is not facet-valid
with respect to maxLength '30' for type 'MyStringType'.
```

4. Edit the configuration file to comment out both the `<jaxws:client>` and `<jaxws:endpoint>` elements to restore the default behavior. Alternatively, remove `cxf.xml` from your `CLASSPATH`.

Understanding the sample

When you run the Maven `mvn install` command, Maven compiles the Java files and creates Java class files. Maven creates the `install_dir\samples\wsdl_first\target` directory, which includes the client and server class files.

When you run the Maven `mvn -Pserver` and `mvn -Pclient` commands, Maven starts the server and client and executes the operations in the WSDL and Java class files.

The sample files in the `wsdl_first` directory include the following:

- `pom.xml` — This file is used by the Maven tooling when creating the service unit and required files for packaging and deploying the service into a container.
- `install_dir\samples\wsdl_first\wsdl\hello_world.wsdl` — Using the WSDL-first approach, the first step in designing services is to define your services in WSDL and XML Schema before writing any code. Examine this file to see how the service is defined, including the following:
  - The WSDL types for each of the elements used with the service, for example:

```
<wsdl:types>
   ...
   <simpleType name="MyStringType">
      <restriction base="string">
         <maxLength value="30" />
      </restriction>
   </simpleType>
   <element name="greetMe">
      <complexType>
         <sequence>
            <element name="requestType"
               type="tns:MyStringType"/>
         </sequence>
      </complexType>
   </element>
...
```

- The operations used with the service. Example 5.1 on page 29 shows the operations defined for the service, including the three greetings sent by the client and the `pingMe` that defines the `PingMeFault` exception thrown for invalid length string greetings.
- The WSDL binding and transport used by the service. This sample specifies a SOAP binding and a transport that corresponds to the HTTP binding in the SOAP specification:

```
...
<wsdl:binding name="Greeter_SOAPBinding"
   type="tns:Greeter">
<soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http"/>
```

Examine the WSDL file to see all the details of the service definition.

- `install_dir\samples\wsdl_first\cxf.xml` — This configuration file enables schema validation when included on the `CLASSPATH`. See "Enabling schema validation" on page 33.
- `install_dir\samples\wsdl_first\src\demo\hw\client\*.java` — Java files that define the client classes. Examine `Client.java` to see how the client invokes the operations defined in `hello_world.wsdl`.
- `install_dir\samples\wsdl_first\src\demo\hw\server\*.java` — Java files that define the server classes.

Deploying the sample into an OSGi container

OSGi is a mature, lightweight, component system that solves many challenges associated with medium and large scale development projects. Through the use of bundles complexity is reduced by separating concerns and ensuring dependencies are minimally coupled via well defined interface communication. This also promotes the reuse of components in much the same way that SOA promotes the reuse of services. And, since each bundle effectively is given an isolated environment, and since dependencies are explicitly defined, versioning and dynamic updates are possible. These are just a few of the many benefits of OSGi. Users wishing to learn more should check out http://www.osgi.org/Main/HomePage.

Before you can install an application into an OSGi container, you must package it into one or more OSGi bundles. An OSGi bundle is a JAR that contains extra information that is used by the OSGi container. This extra information specifies the packages this bundle exposes to the other bundles in the container and any packages on which this bundle depends.

See the chapter "Deploying to an OSGi Container" in *Configuring and Deploying FUSE™ Services Framework Endpoints* [10], for more information about deploying to an OSGi container, and for instructions that use this WSDL-first sample to demonstrate the steps of installing an application into an OSGi container.

[10] http://fusesource.com/docs/framework/2.2/deploy_guide/index.html

Java-First JAX-WS Service Development

Overview

This sample provides an example of service development using a code first approach using the JAX-WS APIs. The `HelloWorld.java` file included with this sample defines the service, and the `Client.java` file defines a `Client` class with a `Client()` method that calls different users.

Using Maven

These instructions use Maven to build and run the sample, using the `pom.xml` file located the base directory of this sample. If you prefer to use wsdl2java, javac, and java to build and run the sample, see the ReadMe file located in your `install_dir\samples\java_first_jaxws` directory.

See "About Maven" on page 28 and "Installing and setting up Maven" on page 27 for more information about using Maven with the FUSE Services Framework samples.

Building and running the sample

To build and run the sample using Maven commands:

1. Open a console to the sample directory, `install_dir\samples\java_first_jaxws` and enter the following command to build the sample:

   ```
   mvn install
   ```

   Maven builds the sample and downloads required JAR files to your Maven repository:

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed - org.apache.cxf.samples:java_first_jaxws:jar:1.0
[INFO]    task-segment: [install]
[INFO] ------------------------------------------------------------------------
...
[INFO] Installing install_dir\samples\java_first_jaxws\target\java_first_jaxws-1.0.jar to
...\.m2\repository\org\apache\cxf\samples\java_first_jaxws\1.0\java_first_jaxws-1.0.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
...
```

2. Enter the following command to start the server:

```
mvn -Pserver
```

When the server is started, the console displays the message `Server ready....`

3. Open another console at the same location and enter the following command to start the client:

```
mvn -Pclient
```

The client starts and sends a greeting to the server. Messages in the client and server consoles show the messages sent as the sample runs.

Examining the output

The messages between the client and server appear rapidly in the consoles as messages are exchanged. When the messages have all been sent, you can scroll up in the console windows to track the following events:

1. The client creates the `HellpWorld` service, and receives responses from the server:

```
Client Console
INFO: Creating Service {http://server.hw.demo/}
HelloWorld from class demo.hw.server.HelloWorld
Hello World
```

Messages in the server console indicate that `sayHi`, `sayHiToUser`, and `getUsers` are called:

```
Server Console
sayHi called
sayHiToUser called
sayHiToUser called
sayHiToUser called
getUsers called
```

2. The client receives responses to the `sayHiToUser` from the server, and prints a list of users from the `getUsers` call:

```
Client Console
Hello World
Hello Galaxy
Hello Universe

Users:
   1: World
   2: Galaxy
   3: Universe
```

3. To remove the code generated from the WSDL file and the `*.class` files, run `mvn clean`.

Understanding the sample

When you run the Maven `mvn install` command, Maven compiles the Java files and creates Java class files. Maven creates the `install_dir\samples\java_first_jaxws\target` directory, which includes the client and server class files.

When you run the Maven `mvn -Pserver` and `mvn -Pclient` commands, Maven starts the server and client and executes the operations in the Java class files.

The sample files in the `wsdl_first` directory include the following:

- `pom.xml` — This file is used by the Maven tooling when creating the service unit and required files for packaging and deploying the service into a container.

- install_dir\samples\java_first_jaxws\src\demo\hw\server\hello_world.java
  — This file defines the Hello World web service in this sample:

Example 5.4. Java-First Sample: hello_world.java

```
@WebService
public interface HelloWorld {

   String sayHi(String text);

   /* Advanced usecase of passing an Interface in.  JAX-WS/JAXB does not
   * support interfaces directly.  Special XmlAdapter classes need to
   * be written to handle them
   */
   String sayHiToUser(User user);

   /* Map passing
   * JAXB also does not support Maps.  It handles Lists great, but Maps are
   * not supported directly.  They also require use of a XmlAdapter to map
   * the maps into beans that JAXB can use.
   */
   @XmlJavaTypeAdapter(IntegerUserMapAdapter.class)
   Map<Integer, User> getUsers();
}
```

- install_dir\samples\java_first_jaxws\src\demo\hw\server*.java
  — Additional Java files that define classes used by the server when this sample runs.
- install_dir\samples\java_first_jaxws\src\demo\hw\client\client.java
  — Java file that defines the client class.

The `Client.java` file defines the `Client()` method:

Example 5.5. Java-First Sample: client.java

```
...
private Client() {
}

   public static void main(String args[]) throws Exception {
   Service service = Service.create(SERVICE_NAME);
   // Endpoint Address
   String endpointAddress = "http://localhost:9000/helloWorld";

   // Add a port to the Service
   service.addPort(PORT_NAME, SOAPBinding.SOAP11HTTP_BINDING, endpointAddress);

   HelloWorld hw = service.getPort(HelloWorld.class);
   System.out.println(hw.sayHi("World"));

   User user = new UserImpl("World");
   System.out.println(hw.sayHiToUser(user));

   //say hi to some more users to fill up the map a bit
   user = new UserImpl("Galaxy");
   System.out.println(hw.sayHiToUser(user));

   user = new UserImpl("Universe");
   System.out.println(hw.sayHiToUser(user));

   System.out.println();
   System.out.println("Users: ");
   Map<Integer, User> users = hw.getUsers();
   for (Map.Entry<Integer, User> e : users.entrySet()) {
      System.out.println("  " + e.getKey() + ": " + e.getValue().getName());
   }
}
...
```

The `Client()` method in this Java file calls the Java classes defined on the server to get the users `World`, `Galaxy`, and `Universe`, and to map the users and print out a list of users. Examine the `client.java` file and the Java files for the server to see how this Java-first sample is coded and implemented.

JAX-RS Service Development

Overview

This sample includes a basic REST-based web service developed using JAX-RS (JSR-311). The client code in this sample demonstrates how to send HTTP GET/POST/PUT/DELETE requests. The server code demonstrates how to build a RESTful endpoint through JAX-RS (JSR-311) APIs.

What happens when you run the sample

The following events occur on the client and server when you run this sample:

1. A RESTful customer service is provided on the URL `http://localhost:9000/customers`. Users access this URI to operate on a customer.
2. The server responds to an HTTP GET request to the URL `http://localhost:9000/customerservice/customers/123` and returns an XML document with customer information for a customer instance whose `id` is 123.
3. The server responds to an HTTP GET request to the URL `http://localhost:9000/customerservice/orders/223/products/323` and returns an XML document with product information for product 323 that belongs to order 223.
4. The server responds to an HTTP POST request to the URL `http://localhost:9000/customerservice/customers` and adds a customer named Jack.
5. The server responds to an HTTP PUT request to the URL `http://localhost:9000/customerservice/customers` and updates the customer instance whose `id` is 123.

Using Maven

These instructions use Maven to build and run the sample. If you prefer to use wsdl2java, javac, and java to build and run the sample, see the ReadMe file located in your `install_dir\samples\jax_rs\basic\` directory.

See "About Maven" on page 28 and "Installing and setting up Maven" on page 27 for more information about using Maven with the FUSE Services Framework samples.

Building and running the sample using Maven

The `pom.xml` file located in the base directory of this sample is used to build and run the demo.

To build and run the sample:

1. Open a console to the sample directory, `install_dir\samples\jax_rs\basic\` and enter the following command to build the sample:

   ```
   mvn install
   ```

   Maven builds the sample and downloads required JAR files to your Maven repository:

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed - org.apache.cxf.samples:jax_rs_basic:jar:1.0
[INFO]    task-segment: [install]
[INFO] ------------------------------------------------------------------------
...
[INFO] Installing install_dir\samples\jax_rs\basic\target\jax_rs_basic-1.0.jar to
C:\Documents and Settings\kpatras\.m2\repository\org\apache\cxf\samples\jax_rs_basic\
1.0\jax_rs_basic-1.0.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
...
```

2. Enter the following command to start the server:

   ```
   mvn -Pserver
   ```

   When the server is started, the console displays the message `Server ready....`

3. Open another console at the same location and enter the following command to start the client:

```
mvn -Pclient
```

The client starts and sends a greeting to the server. Messages in the client and server consoles show the messages sent as the sample runs.

Examining the output

The messages between the client and server appear rapidly in the consoles as messages are exchanged. When the messages have all been sent, you can examine the output in the client and server consoles to observe the following events:

1. The client requests customer information:

```
Sent HTTP GET request to query customer info
```

The server invokes `getCustomer` to get customer information:

```
invoking getCustomer, Customer id is: 123
```

The client receives the customer information in an XML document:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Customer>
   <id>123</id>
   <name>John</name>
</Customer>
```

2. The client requests product information:

```
Sent HTTP GET request to query sub resource product info
```

The server invokes `getOrder` and `getProduct`:

```
invoking getOrder, Order id is: 223
invoking getProduct with id: 323
```

The client receives the product information in an XML document:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Product>
   <description>product 323</description>
   <id>323</id>
</Product>
```

3. The client request an update to the customer information:

```
Sent HTTP PUT request to update customer info
```

The server invokes `updateCustomer`:

```
invoking updateCustomer, Customer name is: Mary
```

The client receives a response with status code:

```
Response status code: 200
Response body:
```

4. The client requests adding the customer:

```
Sent HTTP POST request to add customer
```

The server invokes `addCustomer`:

```
invoking addCustomer, Customer name is: Jack
```

The client receives a response with an XML document, and the sample ends:

```
Response status code: 200
Response body:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Customer>
   <id>124</id>
   <name>Jack</name>
</Customer>

Client Invoking is succeeded!
```

Understanding the sample

When you run the Maven `mvn install` command, Maven compiles the Java files and creates Java class files. Maven creates the `install_dir\\samples\jax_rs\basic\target` directory, which includes the client and server class files.

When you run the Maven `mvn -Pserver` and `mvn -Pclient` commands, Maven starts the server and client and executes the operations in the Java class files.

The sample files in the `jax_rs\basic` directory include the following:

- `pom.xml` — This file is used by the Maven tooling when creating the service unit and required files for packaging and deploying the service into a container.
- `install_dir\samples\jax_rs\basic\src\demo\jaxrs\client\*.java` — Java files that define the client classes, including `Client.java`, which sends HTTP GET, PUT, and POST requests:

Example 5.6. JAX-RS Sample: Client.java

```
...
private Client() {
}

public static void main(String args[]) throws Exception {
   ...
   // Sent HTTP GET request to query customer info
   System.out.println("Sent HTTP GET request to query customer info");
   URL url = new URL("http://localhost:9000/customerservice/customers/123");
   InputStream in = url.openStream();
   System.out.println(getStringFromInputStream(in));

   // Sent HTTP GET request to query sub resource product info
   ...
   System.out.println("Sent HTTP GET request to query sub resource product info");
   url = new
      URL("http://localhost:9000/customerservice/orders/223/products/323");
   in = url.openStream();
   System.out.println(getStringFromInputStream(in));

    // Sent HTTP PUT request to update customer info
   ...
   System.out.println("Sent HTTP PUT request to update customer info");
   Client client = new Client();
   String inputFile = client.getClass().getResource("update_customer.txt").getFile();
   File input = new File(inputFile);
   PutMethod put = new
      PutMethod("http://localhost:9000/customerservice/customers");
   RequestEntity entity = new FileRequestEntity(input, "text/xml; charset=ISO-8859-1");
   put.setRequestEntity(entity);
   HttpClient httpclient = new HttpClient();
   ...
   // Sent HTTP POST request to add customer
   System.out.println("Sent HTTP POST request to add customer");
   inputFile = client.getClass().getResource("add_customer.txt").getFile();
```

```
   input = new File(inputFile);
   PostMethod post = new
      PostMethod("http://localhost:9000/customerservice/customers");
   post.addRequestHeader("Accept" , "text/xml");
   entity = new FileRequestEntity(input, "text/xml; charset=ISO-8859-1");
   post.setRequestEntity(entity);
   httpclient = new HttpClient();
 ...
}
```

- `install_dir\samples\jax_rs\basic\src\demo\jaxrs\server\*.java` — Java files that define the server classes. These files include `Server.java`, which creates a new instance of the RESTful service, `CustomerService()`:

Example 5.7. JAX-RS Sample: Server.java

```
package demo.jaxrs.server;

import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
import org.apache.cxf.jaxrs.lifecycle.SingletonResourcePro
vider;

public class Server {

   protected Server() throws Exception {
      JAXRSServerFactoryBean sf = new JAXRSServerFactory
Bean();
      sf.setResourceClasses(CustomerService.class);
      sf.setResourceProvider(CustomerService.class,
     new SingletonResourceProvider(new CustomerService()));

      sf.setAddress("http://localhost:9000/");

      sf.create();
   }

   public static void main(String args[]) throws Exception
 {
      new Server();
      System.out.println("Server ready...");
   ...
}
```

Chapter 6. Next Steps

Further reading

For more information about developing services with FUSE Services Framework, see *Using the Library*. This guide provides links to the complete set of FUSE Services Framework documentation, and suggests reading paths for the following:

- Service consumer developers
- Java-first service developers
- WSDL-first service developers
- RESTful service developers
- JavaScript service and consumer developers

Additional samples

The FUSE Services Framework `installation_dir\samples` directory includes additional samples you can examine and run to more fully explore developing services using JAX-WS and JAX-RS. These samples include:

- Java-first development with JAX-WS:
  - `java_first_jaxws_factory_bean` — Use JAX-WS factory beans
  - `java_first_pojo` — Use a code-first, POJO approach
  - `java_first_spring_support` — Use Spring beans and set up a HTTP servlet transport
- WSDL-first development with JAX-WS:
  - `wsdl_first_dynamic_client` — Use a dynamic client against a standalone server using SOAP over HTTP
  - `wsdl_first_https` — Develop a service that uses HTTPS communication
  - `wsdl_first_pure_xml` — Use an XML binding with the doc-literal style

- `wsdl_first_rpclit` — Use the RPC-literal style binding
- `wsdl_first_soap12` — Implement SOAP 1.2 capabilities
- `wsdl_first_xml_wrapped` — Examine how the XML binding works with the doc-literal wrapped style
- `wsdl_first_xml_beans2` — Use the JAX-WS APIs with the XMLBeans data binding to run a simple client against a standalone server using SOAP over HTTP

- JAX-RS development with HTTPS, content negotiation, and Spring security
  - `jax_rs\basic_https` — Extend the JAX-RS sample on page 44 to implement communication using HTTPS
  - `jax_rs\content_negotiation` — Develop a RESTful service that implements content negotiation so that the same resource can be served using multiple representations
  - `jax_rs\spring_security` — Use Spring security to secure a RESTful service

FUSE Services Framework also includes samples that illustrate the following:

- JMS queue and publish-and-subscribe messaging
- JavaScript service development
- WS addressing, policy, and security
- Additional topics

See the FUSE Services Framework `samples` directory for the complete set of samples. Each sample includes a ReadMe file with instructions to run the sample.

Index