Red Hat JBoss Enterprise Application Platform 7.4

Developing Web Services Applications

Instructions for developing web services applications for Red Hat JBoss Enterprise Application Platform.

Last Updated: 2024-01-17

Red Hat JBoss Enterprise Application Platform 7.4 Developing Web Services Applications

Instructions for developing web services applications for Red Hat JBoss Enterprise Application Platform.

Legal Notice

Abstract

This guide provides information about how to develop web services applications with Red Hat JBoss Enterprise Application Platform.

Table of Contents

PROVIDING FEEDBACK ON JBOSS EAP DOCUMENTATION

To report an error or to improve our documentation, log in to your Red Hat Jira account and submit an issue. If you do not have a Red Hat Jira account, then you will be prompted to create an account.

Procedure

1. Click the following link to **create a ticket**.
2. Please include the **Document URL**, the **section number** and **describe the issue**.
3. Enter a brief description of the issue in the **Summary**.
4. Provide a detailed description of the issue or enhancement in the **Description**. Include a URL to where the issue occurs in the documentation.
5. Clicking **Submit** creates and routes the issue to the appropriate documentation team.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. INTRODUCTION TO WEB SERVICES

Web services provide a standard means of interoperating among different software applications. Each application can run on a variety of platforms and frameworks.

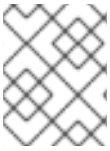
Web services facilitate internal, heterogeneous subsystem communication. The interoperability increases service reuse because functions do not need to be rewritten for various environments.

CHAPTER 2. DEVELOPING JAKARTA RESTFUL WEB SERVICES WEB SERVICES

Jakarta RESTful Web Services is the Jakarta EE API for RESTful web services. It provides support for building web services using representational state transfer, or "REST," using annotations. These annotations simplify the process of mapping Java objects to web resources.

RESTEasy is the Red Hat JBoss Enterprise Application Platform 7 implementation of Jakarta RESTful Web Services. JBoss EAP 7.3 and later are compliant with the Jakarta Enterprise Web Services 1.4 and the Jakarta RESTful Web Services 2.1 Jakarta EE specifications. They also provide additional features to the specification.

To get started with Jakarta RESTful Web Services, see the **helloworld-rs**, **jaxrs-client**, and **kitchensink** quickstarts that ship with Red Hat JBoss Enterprise Application Platform 7.



NOTE

JBoss EAP does not support the **resteasy-crypto**, **resteasy-yaml-provider**, and **jose-jwt** modules.

2.1. JAKARTA RESTFUL WEB SERVICES APPLICATION

When creating providers and web resources, you have the following options for declaring them:

- Simple subclassing of **javax.ws.rs.core.Application** without a **web.xml** file.
- Using a **web.xml** file.
- Subclassing **javax.ws.rs.core.Application** and providing a custom implementation.

2.1.1. Simple Subclassing javax.ws.rs.core.Application

You can use the **javax.ws.rs.core.Application** class to create a subclass that declares those providers and web resources. This class is provided by the RESTEasy libraries included with JBoss EAP.

To configure a resource or provider using **javax.ws.rs.core.Application**, simply create a class that extends it and add an **@ApplicationPath** annotation.

Example: Application Class

```
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/root-path")
public class MyApplication extends Application
{
}
```

2.1.2. Using web.xml

Alternatively, if you do not want to create a class that extends **javax.ws.rs.core.Application**, you can add the following to your **web.xml** file.

Example: web.xml

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <servlet>
    <servlet-name>javax.ws.rs.core.Application</servlet-name>
  </servlet>
  <servlet-mapping>
    <servlet-name>javax.ws.rs.core.Application</servlet-name>
    <url-pattern>/root-path/*</url-pattern>
  </servlet-mapping>
  ...
</web-app>
```

2.1.3. Subclassing javax.ws.rs.core.Application with a Custom Implementation

When subclassing **javax.ws.rs.core.Application** you can choose to provide a custom implementation for any of the existing methods. The **getClasses** and **getSingletons** methods return a collection of classes or singletons that must be included in the published Jakarta RESTful Web Services application.

- If either **getClasses** and **getSingletons** returns a non-empty collection, *only* those classes and singletons are published in the Jakarta RESTful Web Services application.
- If *both* **getClasses** and **getSingletons** return an empty collection, then *all* root resource classes and providers that are packaged in the web application are included in the Jakarta RESTful Web Services application. RESTEasy will then automatically discover those resources.

2.2. JAKARTA RESTFUL WEB SERVICES CLIENT

2.2.1. Jakarta RESTful Web Services Client API

Jakarta RESTful Web Services 2.0 introduced a new client API to send HTTP requests to remote RESTful web services. It is a *fluent* request-building API with 3 main classes:

- **Client**
- **WebTarget**
- **Response**

The **Client** interface is a builder of WebTarget instances. The **WebTarget** represents a distinct URL or URL template to build subresource WebTargets or invoke requests on.

There are two ways to create a client: the standard way, or using the **ResteasyClientBuilder** class. The advantage of using the **ResteasyClientBuilder** class is that it provides a few more helper methods to configure your client.

While the **ResteasyClientBuilder** class provides these helper methods, the class is specific to JBoss EAP APIs. If you want to migrate the application to a new server you must rebuild the application. The **ResteasyClientBuilder** class depends on RESTEasy and the class is not portable.

The standard way of creating a client adheres to both the Jakarta RESTful Web Services and Jakarta EE API specifications, and it is portable among Jakarta RESTful Web Services implementations.

NOTE

To ensure Jakarta RESTful Web Services applications remain portable, adhere to the Jakarta EE API specification and, if possible, use Jakarta EE API applications. Only use APIs specific to JBoss if a use case does not support using Jakarta EE APIs.

Following these guidelines can help reduce the number of issues that might occur when migrating an application to a different server or to a new Jakarta EE compatible JBoss implementation.

Using the Standard Way to Create a Client
The following example shows one of the standard ways to create a client:

```
Client client = ClientBuilder.newClient();
```

Alternatively, you can use another standard way to create a client as shown in the example below:

```
Client client = ClientBuilder.newBuilder().build();
WebTarget target = client.target("http://foo.com/resource");
Response response = target.request().get();
String value = response.readEntity(String.class);
response.close();  // You should close connections!
```

Using the ResteasyClientBuilder Class to Create a Client
The following example shows use of the **ResteasyClientBuilder** class to create a client:

```
ResteasyClient client = new ResteasyClientBuilder().build();
ResteasyWebTarget target = client.target("http://foo.com/resource");
```

With Jakarta RESTful Web Services 2.1, you can add two timeout methods to the **ClientBuilder** class. Timeout methods are specification-compliant methods, and you can use them instead of using the RESTEasy methods.

The following **ClientBuilder** specification-compliant methods replace certain deprecated RESTEasy methods:

- The **connectTimeout** method replaces the **establishConnectionTimeout** method.
  The **connectTimeout** method determines how long the client must wait when making a new server connection.

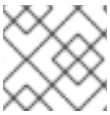
- The **readTimeout** method replaces the **socketTimeout** method.
  The **readTimeout** method determines how long the client must wait for a response from the server.

The following example shows specified values for the **connectTimeout** and **readTimeout** methods:

```
import javx.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;

Client client = ClientBuilder.newBuilder()
```

```
        .connectTimeout(100, TimeUnit.SECONDS)
        .readTimeout(2, TimeUnit.SECONDS)
        .build();
```

Note that **readTimeout** applies to an executed request on an already existing connection.



NOTE

Setting the value of the timeout parameter to zero causes the server to wait indefinitely.

RESTEasy automatically loads a set of default providers that includes all classes listed in the **META-INF/services/javax.ws.rs.ext.Providers** file. Additionally, you can manually register other providers, filters, and interceptors through the configuration object provided by the method call **Client.configuration()**. Configuration also lets you set configuration properties that might be needed.

Each **WebTarget** has a configuration instance that inherits the components and properties registered with the parent instance. This lets you set specific configuration options for each target resource, for example, the username and password.

Additional Resources

- For more information about the **ResteasyClientBuilder** class and its methods, see the Class ResteasyClientBuilder.

Using RESTEasy Client Classes

You must add the following dependency for RESTEasy client to your Maven **pom.xml** file:

```
<dependency>
 <groupId>org.jboss.resteasy</groupId>
 <artifactId>resteasy-client</artifactId>
 <version>VERSION_IN_EAP</version>
 </dependency>
```

See the **jaxrs-client** and **resteasy-jaxrs-client** quickstarts that ship with JBoss EAP for working examples that use the RESTEasy client classes.

Client-side Filters

The client side has two types of filters:

ClientRequestFilter

A **ClientRequestFilter** runs before an HTTP request is sent over the wire to the server. The **ClientRequestFilter** is also allowed to abort the request execution and provide a canned response without going over the wire to the server.

ClientResponseFilter

A **ClientResponseFilter** runs after a response is received from the server, but before the response body is unmarshalled. The **ClientResponseFilter** can modify the response object before it is handed to the application code. The following example illustrates these concepts:

```
// execute request filters
for (ClientRequestFilter filter : requestFilters) {
 filter.filter(requestContext);
 if (isAborted(requestContext)) {
   return requestContext.getAbortedResponseObject();
  }
```

```
}

// send request over the wire
response = sendRequest(request);

// execute response filters
for (ClientResponseFilter filter : responseFilters) {
 filter.filter(requestContext, responseContext);
}
```

Register Client-side Filters to the Client Request
The following example shows how to register the client-side filters to the client request:

```
client = ClientBuilder.newClient();
WebTarget base = client.target(generateURL("/") + "get");
base.register(ClientExceptionsCustomClientResponseFilter.class).request("text/plain").get();
```

Client-side Cache
RESTEasy has the ability to set up a client-side cache. This cache looks for cache-control headers sent back with a server response. If the cache-control headers specify that the client is allowed to cache the response, RESTEasy caches it within the local memory.

```
ResteasyWebTarget target = client.target(generateBaseUrl());
target.register(BrowserCacheFeature.class);
```

Chunked Encoding Support
RESTEasy provides the client API the ability to specify that requests should be sent in a *chunked* transfer mode. There are two ways of specifying the *chunked* transfer mode, as shown below.

- You can configure the **org.jboss.resteasy.client.jaxrs.ResteasyWebTarget** to send all the requests in chunked mode:

  ```
  ResteasyClient client = new ResteasyClientBuilder().build();
  ResteasyWebTarget target = client.target("http://localhost:8081/test");
  target.setChunked(b.booleanValue());
  Invocation.Builder request = target.request();
  ```

- Alternatively, you can configure a particular request to be sent in chunked mode:

  ```
  ResteasyClient client = new ResteasyClientBuilder().build();
  ResteasyWebTarget target = client.target("http://localhost:8081/test");
  ClientInvocationBuilder request = (ClientInvocationBuilder) target.request();
  request.setChunked(b);
  ```

  Unlike the **javax.ws.rs.client.Invocation.Builder** class, **org.jboss.resteasy.client.jaxrs.internal.ClientInvocationBuilder** is a RESTEasy class.

NOTE

The ability to send the requests in chunked mode depends on the underlying transport layer. In particular, it depends on the implementation of the **org.jboss.resteasy.client.jaxrs.ClientHttpEngine** class being used. Currently, only the default implementation **ApacheHttpClient43Engine** and the previous implementation **ApacheHttpClient4Engine** support the chunked mode. Both these are available in the **org.jboss.resteasy.client.jaxrs.engines** package. See section Implementing RESTEasy with HTTP Client for more information.

2.2.2. Implementing RESTEasy with HTTP Client

Network communication between the client and server is handled by default in RESTEasy. It uses the **HttpClient** from the Apache **HttpComponents** project. The interface between the RESTEasy client framework and the network is defined by the **ClientHttpEngine** interface.

RESTEasy ships with four implementations of this interface. The default implementation is **ApacheHttpClient43Engine**. This implementation uses Apache 4.3.

ApacheHttpClient4Engine is an implementation that uses the versions earlier than Apache 4.3. This class provides backward compatibility. RESTEasy automatically selects one of these two **ClientHttpEngine** implementations based on the detection of the Apache version. **InMemoryClientEngine** is an implementation that dispatches requests to a server in the same JVM, and **URLConnectionEngine** is an implementation that uses **java.net.HttpURLConnection**.

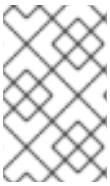
A client executor can be passed to a specific **ClientRequest**:

```
ResteasyClient client = new
ResteasyClientBuilder().httpEngine(engine).build();
```

RESTEasy and **HttpClient** make the default decisions to use the client framework without referencing **HttpClient**. However, for some applications it might be necessary to drill down into the **HttpClient** details. **ApacheHttpClient43Engine** and **ApacheHttpClient4Engine** can be supplied with an instance of **org.apache.http.client.HttpClient** and **org.apache.http.protocol.HttpContext**, which can carry additional configuration details into the **HttpClient** layer. For example, authentication can be configured as follows:

```
// Configure HttpClient to authenticate preemptively
// by prepopulating the authentication data cache.

// 1. Create AuthCache instance
AuthCache authCache = new BasicAuthCache();

// 2. Generate BASIC scheme object and add it to the local auth cache
AuthScheme basicAuth = new BasicScheme();
authCache.put(new HttpHost("sippycups.bluemonkeydiamond.com"), basicAuth);

// 3. Add AuthCache to the execution context
BasicHttpContext localContext = new BasicHttpContext();
localContext.setAttribute(ClientContext.AUTH_CACHE, authCache);

// 4. Create client executor and proxy
HttpClient httpClient = HttpClientBuilder.create().build();
ApacheHttpClient4Engine engine = new ApacheHttpClient4Engine(httpClient, localContext);
ResteasyClient client = new ResteasyClientBuilder().httpEngine(engine).build();
```

HttpContextProvider is an interface provided by RESTEasy, using which you can supply a custom **HttpContext** to the **ApacheHttpClient43Engine** and **ApacheHttpClient4Engine** implementations.



NOTE

It is important to understand the difference between *releasing* a connection and *closing* a connection. Releasing a connection makes it available for reuse. Closing a connection frees its resources and makes it unusable.

RESTEasy releases the connection without notification. The only counter example is the case in which the response is an instance of **InputStream**, which must be closed explicitly.

On the other hand, if the result of an invocation is an instance of **Response**, the **Response.close()** method must be used to release the connection.

```
WebTarget target = client.target("http://localhost:8081/customer/123");
Response response = target.request().get();
System.out.println(response.getStatus());
response.close();
```

You may execute this in a **try-finally** block. Releasing a connection makes it available for another use. It does not normally close the socket.

ApacheHttpClient4Engine.finalize() closes any open sockets, if it created the **HttpClient** that it has been using. It is not safe to rely on JDK to call **finalize()**. If an **HttpClient** is passed to the **ApacheHttpClient4Executor**, the user must close the connections, as shown below:

```
HttpClient httpClient = new HttpClientBuilder.create().build();
ApacheHttpClient4Engine executor = new ApacheHttpClient4Engine(httpClient);
...
httpClient.getConnectionManager().shutdown();
```

NOTE

If **ApacheHttpClient4Engine** has created its own instance of **HttpClient**, it is not necessary to wait for **finalize()** to close open sockets. The **ClientHttpEngine** interface has a **close()** method for this purpose.

Finally, if the **javax.ws.rs.client.Client** class has created the engine automatically, call **Client.close()**. This call cleans up any socket connections.

2.2.2.1. HTTP Redirect

The **ClientHttpEngine** implementations based on the Apache HttpClient support HTTP redirection. This feaure is disabled by default. You can enable this by setting the **setFollowRedirects** method to **true**, as shown below:

```
ApacheHttpClient43Engine engine = new ApacheHttpClient43Engine();
engine.setFollowRedirects(true);
Client client = new ResteasyClientBuilder().httpEngine(engine).build();
```

2.3. JAKARTA RESTFUL WEB SERVICES REQUEST PROCESSING

2.3.1. Asynchronous HTTP Request Processing

Asynchronous request processing allows you to process a single HTTP request using non-blocking input and output and, if required, in separate threads.

Consider an AJAX chat client in which you want to push and pull from both the client and the server. This scenario has the client blocking for a long time on the server's socket, waiting for a new message. In case of synchronous HTTP processing, where the server is blocking on incoming and outgoing input and output, you have one separate thread consumed per client connection. This model of request processing consumes a lot of memory and valuable thread resources.

Asynchronous processing separates the connection accepting and the request processing operations. It allocates two different threads: one to accept the client connection; the other to handle heavy, time-consuming operations. In this model, the container works as follows:

1. It dispatches a thread to accept a client connection, which is the acceptor.
2. Then it hands over the request to the processing thread, which is the worker.
3. Finally, it releases the acceptor thread.

The result is sent back to the client by the worker thread. Hence, the client's connection remains open, thereby improving the server's throughput and scalability.

2.3.1.1. Asynchronous NIO Request Processing

RESTEasy's default asynchronous engine implementation class is **ApacheHttpAsyncClient4Engine**. It is built on **HttpAsyncClient** from the Apache **HttpComponents**, which internally dispatches requests using a non-blocking IO model.

You can set the asynchronous engine as the active engine by calling the **useAsyncHttpEngine** method in the **ResteasyClientBuilder** class:

```
Client asyncClient = new ResteasyClientBuilder().useAsyncHttpEngine()
                    .build();
Future<Response> future = asyncClient
                        .target("http://locahost:8080/test").request()
                        .async().get();
Response res = future.get();
Assert.assertEquals(HttpResponseCodes.SC_OK, res.getStatus());
String entity = res.readEntity(String.class);
```

2.3.1.2. Server Asynchronous Response Processing

On the server side, asynchronous processing involves suspending the original request thread and initiating the request processing on a different thread, which releases the original server-side thread to accept other incoming requests.

2.3.1.2.1. AsyncResponse API

The Jakarta RESTful Web Services 2.1 specification defines asynchronous HTTP support using two classes: the **@Suspended** annotation and the **AsyncResponse** interface.

Injecting an **AsyncResponse** as a parameter to your Jakarta RESTful Web Services method prompts RESTEasy to detach the HTTP request and response from the thread being executed currently. This ensures that the current thread does not try to automatically process the response.

The **AsyncResponse** is the callback object. The act of calling one of the **resume()** methods causes a response to be sent back to the client and also terminates the HTTP request. The following is an example of asynchronous processing:

```
import javax.ws.rs.container.Suspended;
import javax.ws.rs.container.AsyncResponse;

@Path("/")
public class SimpleResource {
  @GET
  @Path("basic")
  @Produces("text/plain")
  public void getBasic(@Suspended final AsyncResponse response) throws Exception {
    Thread t = new Thread() {
      @Override
      public void run() {
        try {
          Response jaxrs = Response.ok("basic").type(MediaType.TEXT_PLAIN).build();
          response.resume(jaxrs);
        }
        catch (Exception e) {
          e.printStackTrace();
        }
      }
    };
    t.start();
  }
}
```

2.3.1.3. AsyncInvoker Client API

Similarly, on the client side, asynchronous processing prevents blocking the request thread since no time is spent waiting for a response from the server. For instance, a thread that issued a request may also update a user interface component. If that thread is blocked waiting for a response, the user's perceived performance of the application will suffer.

2.3.1.3.1. Using Future

In the code snippet below, the **get()** method is called on the **async()** method rather than the request. This changes the call mechanism from synchronous to asynchronous. Instead of responding synchronously, the **async()** method returns a **future** object. When you call the **get()** method, the call is blocked until the response is ready. The **future.get()** method will be returned whenever the response is ready.

```
import java.util.concurrent.Future;
import javax.ws.rs.client.Client;
...

@Test
public void AsyncGetTest() throws Exception {
    Client client = ClientBuilder.newClient();
```

```
    Future<String> future = client.target(generateURL("/test")).request().async().get(String.class);
    String entity = future.get();
    Assert.assertEquals("get", entity);
}
```

2.3.1.3.2. Using InvocationCallback

The **AsyncInvoker** interface allows you to register an object that will be called back when the asynchronous invocation is ready for processing. The **InvocationCallback** interface provides two methods: **completed()** and **failed()**. The **completed()** method is called whenever the processing is finished successfully and the response is received. Conversely, the **failed()** method is called whenever the request processing is not successful.

```
import javax.ws.rs.client.InvocationCallback;
...

@Test
public void AsyncCallbackGetTest() throws Exception {
    Client client = ClientBuilder.newClient();
 final CountDownLatch latch = new CountDownLatch(1);
       Future<Response> future = client.target(generateURL("/test")).request().async().get(new
InvocationCallback<Response>() {
         @Override
             public void completed(Response response) {
                String entity = response.readEntity(String.class);
                Assert.assertEquals("get", entity);
                latch.countDown();
             }

             @Override
             public void failed(Throwable error) {
             }
          });
          Response res = future.get();
          Assert.assertEquals(HttpResponseCodes.SC_OK, res.getStatus());
          Assert.assertTrue("Asynchronous invocation didn't use custom implemented Invocation
callback", latch.await(5, imeUnit.SECONDS));
}
```

2.3.2. Custom RESTEasy Annotations

With the addition of parameter names in the bytecode, you are no longer required to specify the parameter names in the following annotations: **@PathParam**, **@QueryParam**, **@FormParam**, **@CookieParam**, **@HeaderParam** and **@MatrixParam**. To do so, you must switch to the new annotations with the same name, in a different package, which have an optional value parameter. You can achieve this by following the steps below:

1. Import the **org.jboss.resteasy.annotations.jaxrs** package to replace annotations from the Jakarta RESTful Web Services specifications.

2. Configure your build system to record the method parameter names in the bytecode. Maven users can enable recording method parameter names in the bytecode by setting the **maven.compiler.parameters** to **true**:

```
<properties>
   <maven.compiler.parameters>true</maven.compiler.parameters>
</properties>
```

3. Remove the annotation value if the name matches the name of the annotated variable.

NOTE

You can omit the annotation name for annotated method parameters as well as annotated fields or the JavaBean properties.

Consider the following usage for an example:

```
import org.jboss.resteasy.annotations.jaxrs.*;

@Path("/library")
public class Library {

  @GET
  @Path("/book/{isbn}")
  public String getBook(@PathParam String isbn) {
    // search my database and get a string representation and return it
  }
}
```

If your annotated variable does not have the same name as the path parameter, you can specify the name as shown below:

```
import org.jboss.resteasy.annotations.jaxrs.*;

@Path("/library")
public class Library {

  @GET
  @Path("/book/{isbn}")
  public String getBook(@PathParam("isbn") String id) {
    // search my database and get a string representation and return it
  }
}
```

2.4. VIEWING RESTEASY ENDPOINTS

You can use the **read-resource** operation of the **jaxrs** subsystem to view structured output of each RESTEasy endpoint. An example of the management CLI command and the expected outcome is provided below.

```
/deployment=DEPLOYMENT_NAME/subsystem=jaxrs/rest-
resource=org.jboss.as.quickstarts.rshelloworld.HelloWorld:read-resource(include-runtime=true)
{
   "outcome" => "success",
   "result" => {
      "resource-class" => "org.jboss.as.quickstarts.rshelloworld.HelloWorld",
      "rest-resource-paths" => [
```

```
            {
                "resource-path" => "/hello/json",
                "consumes" => undefined,
                "produces" => [
                    "application/json",
                    "text/plain"
                ],
                "java-method" => "java.lang.String
org.jboss.as.quickstarts.rshelloworld.HelloWorld.getHelloWorldJSON()",
                "resource-methods" => [
                    "POST /wildfly-helloworld-rs/rest/hello/json",
                    "GET /wildfly-helloworld-rs/rest/hello/json"
                ]
            },
            {
                "resource-path" => "/hello/xml",
                "consumes" => undefined,
                "produces" => ["application/xml"],
                "java-method" => "java.lang.String
org.jboss.as.quickstarts.rshelloworld.HelloWorld.getHelloWorldXML(@QueryParam java.lang.String
name = 'LGAO')",
                "resource-methods" => ["GET /wildfly-helloworld-rs/rest/hello/xml"]
            }
        ],
        "sub-resource-locators" => [{
            "resource-class" => "org.jboss.as.quickstarts.rshelloworld.SubHelloWorld",
            "rest-resource-paths" => [
                {
                    "resource-path" => "/hello/subMessage/",
                    "consumes" => undefined,
                    "produces" => undefined,
                    "java-method" => "java.lang.String
org.jboss.as.quickstarts.rshelloworld.SubHelloWorld.helloInSub()",
                    "resource-methods" => ["GET /wildfly-helloworld-rs/rest/hello/subMessage/"]
                },
                {
                    "resource-path" => "/hello/subMessage/subPath",
                    "consumes" => undefined,
                    "produces" => undefined,
                    "java-method" => "java.lang.String
org.jboss.as.quickstarts.rshelloworld.SubHelloWorld.subPath()",
                    "resource-methods" => ["GET /wildfly-helloworld-rs/rest/hello/subMessage/subPath"]
                }
            ],
            "sub-resource-locators" => undefined
        }]
    }
}
```

In the example above, the output information is grouped by the **resource-class** and ordered as per the **resource-path**:

- **resource-path** is the address to access the endpoint.
- **resource-class** defines the class, where the endpoint is defined.

- **rest-resource-paths** includes the Java methods that define the resource path, HTTP method, consumes and produces of the endpoint.
- **java-method** specifies the name of the Java method and its parameters. It also contains information about the following Jakarta RESTful Web Services annotations, if defined: **@PathParam**, **@QueryParam**, **@HeaderParam**, **@MatrixParam**, **@CookieParam**, **@FormParam** and **@DefaultValue**.

Alternatively, you can use the **read-resource** operation without the **rest-resource** parameter defined and get the information about all the endpoints, as shown in the example below:

```
/deployment=DEPLOYMENT_NAME/subsystem=jaxrs:read-resource(include-
runtime=true,recursive=true)
```

2.5. VIEWING RESTEASY ENDPOINTS USING REGISTRYSTATSRESOURCE

You can get information about the RESTEasy endpoints of an application from the **RegistryStatsResource** resource.

Procedure

1. Register **RegistryStatsResource** by adding the following XML snippet in the deployment descriptor, **web.xml** file, of the application:

```
<context-param>
   <param-name>resteasy.resources</param-name>
   <param-value>org.jboss.resteasy.plugins.stats.RegistryStatsResource</param-value>
</context-param>
```

2. View the RESTEasy endpoints of the application:
   - Using the CLI:
     - To get the result in XML:

```
$ curl http://localhost:8080/{APPLICATION_PREFIX_URL}/resteasy/registry
```

     - To get the result in JSON:

```
$ curl http://localhost:8080/{APPLICATION_PREFIX_URL}/resteasy/registry -H
"Accept: application/json"
```

   - Using a web browser:

```
http://localhost:8080/{APPLICATION_PREFIX_URL}/resteasy/registry
```

2.6. URL-BASED NEGOTIATION

2.6.1. Mapping Extensions to Media Types

Some clients, such as browsers, cannot use the **Accept** and **Accept-Language** headers to negotiate the representation media type or language. RESTEasy can map file name suffixes to media types and languages to deal with this issue.

To map media types to file extensions using the **web.xml** file, you need to add a **resteasy.media.type.mappings** context param and the list of mappings as the **param-value**. The list is comma separated and uses colons (**:**) to delimit the file extension and media type.

Example web.xml Mapping File Extensions to Media Types

```
<context-param>
   <param-name>resteasy.media.type.mappings</param-name>
   <param-value>html : text/html, json : application/json, xml : application/xml</param-value>
</context-param>
```

In this example, the following URL variants for **http://localhost:8080/my-application/test** would be mapped:

- **http://localhost:8080/my-application/test.html**
- **http://localhost:8080/my-application/test.json**
- **http://localhost:8080/my-application/test.xml**

2.6.2. Mapping Extensions to Languages

Some clients, such as browsers, cannot use the **Accept** and **Accept-Language** headers to negotiate the representation media type or language. RESTEasy can map file name suffixes to media types and languages to deal with this issue. Follow these steps to map languages to file extensions, in the **web.xml** file.

To map media types to file extensions using the **web.xml** file, you need to add a **resteasy.language.mappings** context param and the list of mappings as the **param-value**. The list is comma separated and uses colons (**:**) to delimit the file extension and language type.

Example web.xml Mapping File Extensions to Language Types

```
<context-param>
   <param-name>resteasy.language.mappings</param-name>
   <param-value> en : en-US, es : es, fr : fr</param-name>
</context-param>
```

In this example, the following URL variants for **http://localhost:8080/my-application/test** would be mapped:

- **http://localhost:8080/my-application/test.en**
- **http://localhost:8080/my-application/test.es**
- **http://localhost:8080/my-application/test.fr**

2.7. CONTENT MARSHALLING AND PROVIDERS

2.7.1. Default Providers and Default Jakarta RESTful Web Services Content Marshalling

RESTEasy can automatically marshal and unmarshal a few different message bodies.

Table 2.1. Supported Media Types and Java Types

Media Types	Java Types
application/* +xml, **text/* +xml**, **application/* +json**, **application/* +fastinfoset**, **application/atom+***	Jakarta XML Binding annotated classes
application/* +xml, **text/* +xml**	org.w3c.dom.Document
* / *	java.lang.String
* / *	java.io.InputStream
text/plain	primitives, java.lang.String, or any type that has a String constructor, or static valueOf(String) method for input, toString() for output
* / *	javax.activation.DataSource
* / *	java.io.File
* / *	byte
application/x-www-form-urlencoded	javax.ws.rs.core.MultivaluedMap

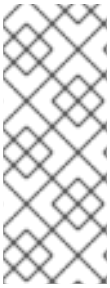
2.7.1.1. Text Media Types and Character Sets

According to the Jakarta RESTful Web Services specification, implementations must adhere to application-supplied character set metadata when writing responses. If a character set is not specified by the application or if the application specifies a character set that is not supported, then the implementations must use UTF-8 character set.

On the contrary, according to the HTTP specification, when no explicit charset parameter is provided by the sender, media subtypes of the **text** type are defined to have a default charset value of **ISO-8859-1** when received via HTTP. Data in character sets other than **ISO-8859-1** or its subsets must be labeled with an appropriate charset value.

In the absence of a character set specified by a resource or resource method, RESTEasy uses UTF-8 as the character set for text media types. In order to do so, RESTEasy adds an explicit charset parameter to the content-type response header.

To specify the original behavior, in which UTF-8 is used for text media types but the explicit charset parameter is not appended, set the context parameter **resteasy.add.charset** to **false**. The default value of this parameter is **true**.



NOTE

Text media types include:

- Media types with type **text** and any subtype.
- Media types with type **application** and subtype beginning with **xml**. This includes **application/xml-external-parsed-entity** and **application/xml-dtd**.

2.7.2. Content Marshalling with @Provider classes

The Jakarta RESTful Web Services specification allows you to plug in your own request/response body readers and writers. To do this, you annotate a class with **@Provider** and specify the **@Produces** types for a writer and **@Consumes** types for a reader. You must also implement a **MessageBodyReader/Writer** interface.

Client providers that are annotated using **@Provider** must be registered for every client instance for the Jakarta RESTful Web Services container runtime to process annotations. To avoid issues with undesired or duplicated client provider registrations, the system property **resteasy.client.providers.annotations.disabled** disables the default processing of client providers that are annotated with **@Provider**.

The RESTEasy **ServletContextLoader** automatically scans the **WEB-INF/lib** and classes directories for classes annotated with **@Provider**, or you can manually configure them in the **web.xml** file.

2.7.3. Providers Utility Class

javax.ws.rs.ext.Providers is a simple injectable interface that allows you to look up **MessageBodyReaders**, **Writers**, **ContextResolvers**, and **ExceptionMappers**. It is very useful for implementing multipart providers and content types that embed other random content types.

```
public interface Providers {
  <T> MessageBodyReader<T> getMessageBodyReader(Class<T> type, Type genericType,
Annotation annotations[], MediaType mediaType);
  <T> MessageBodyWriter<T> getMessageBodyWriter(Class<T> type, Type genericType, Annotation
annotations[], MediaType mediaType);
  <T extends="" throwable=""> ExceptionMapper<T> getExceptionMapper(Class<T> type);
  <T> ContextResolver<T> getContextResolver(Class<T> contextType, MediaType mediaType);
}
```

A **Providers** instance is injectable into **MessageBodyReader** or **Writers**:

```
@Provider
@Consumes("multipart/fixed")
public class MultipartProvider implements MessageBodyReader {

  private @Context Providers providers;
  ...
}
```

2.7.4. Configuring Document Marshalling

XML document parsers are subject to a form of attack known as the XXE (XML eXternal Entity) attack, in which expanding an external entity causes an unsafe file to be loaded. For example, the following document could cause the **/etc/passwd** file to be loaded.

```
<!--?xml version="1.0"?-->
<!DOCTYPE foo
[<!ENTITY xxe SYSTEM "file:///etc/passwd">]>
<search>
 <user>bill</user>
 <file>&xxe;<file>
</search>
```

By default, the RESTEasy built-in unmarshaller for **org.w3c.dom.Document** documents does not expand external entities. It replaces them with an empty string. You can configure it to replace external entities with values defined in the DTD. This is done by setting the **resteasy.document.expand.entity.references** context parameter to **true** in the **web.xml** file.

Example: Setting the resteasy.document.expand.entity.references Context Parameter

```
<context-param>
 <param-name>resteasy.document.expand.entity.references</param-name>
 <param-value>true</param-value>
</context-param>
```

Another way of dealing with the problem is by prohibiting DTDs, which RESTEasy does by default. This behavior can be changed by setting the **resteasy.document.secure.disableDTDs** context parameter to **false**.

Example: Setting the resteasy.document.secure.disableDTDs Context Parameter

```
<context-param>
 <param-name>resteasy.document.secure.disableDTDs</param-name>
 <param-value>false</param-value>
</context-param>
```

Documents are also subject to *Denial of Service Attacks* when buffers are overrun by large entities or too many attributes. For example, if a DTD defined the following entities, the expansion of **&foo6;** would result in 1,000,000 foos.

```
<!--ENTITY foo 'foo'-->
<!--ENTITY foo1 '&foo;&foo;&foo;&foo;&foo;&foo;&foo;&foo;&foo;&foo;'-->
<!--ENTITY foo2 '&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;'-->
<!--ENTITY foo3 '&foo2;&foo2;&foo2;&foo2;&foo2;&foo2;&foo2;&foo2;&foo2;&foo2;'-->
<!--ENTITY foo4 '&foo3;&foo3;&foo3;&foo3;&foo3;&foo3;&foo3;&foo3;&foo3;&foo3;'-->
<!--ENTITY foo5 '&foo4;&foo4;&foo4;&foo4;&foo4;&foo4;&foo4;&foo4;&foo4;&foo4;'-->
<!--ENTITY foo6 '&foo5;&foo5;&foo5;&foo5;&foo5;&foo5;&foo5;&foo5;&foo5;&foo5;'-->
```

By default, RESTEasy limits the number of expansions and the number of attributes per entity. The exact behavior depends on the underlying parser. The limit can be turned off by setting the **resteasy.document.secure.processing.feature** context parameter to **false**.

Example: Setting the resteasy.document.secure.processing.feature Context Parameter

```
<context-param>
```

```
 <param-name>resteasy.document.secure.processing.feature</param-name>
 <param-value>false</param-value>
</context-param>
```

2.7.5. Using MapProvider

You can use **MapProvider** to accept and return a map with Jakarta RESTful Web Services resources.

Example: Resource Accepting and Returning a Map

```
@Path("manipulateMap")
@POST
@Consumes("application/x-www-form-urlencoded")
@Produces("application/x-www-form-urlencoded")
public MultivaluedMap<String, String> manipulateMap(MultivaluedMap<String, String> map) {
  //do something
  return map;
}
```

You can also send and receive maps to Jakarta RESTful Web Services resources using the client.

Example: Client

```
MultivaluedMap<String, String> map = new MultivaluedHashMap<String, String>();

//add values to the map...

Response response = client.target(generateURL("/manipulateMap"))
                    .request(MediaType.APPLICATION_FORM_URLENCODED_TYPE)
                    .post(Entity.entity(map, MediaType.APPLICATION_FORM_URLENCODED_TYPE));

String data = response.readEntity(String.class);

//handle data...
```

2.7.6. Converting String Based Annotations to Objects

Jakarta RESTful Web Services **@*Param** annotations, including **@QueryParam**, **@MatrixParam**, **@HeaderParam**, **@PathParam**, and **@FormParam**, are represented as strings in a raw HTTP request. These types of injected parameters can be converted to objects if these objects have a **valueOf(String)** static method or a constructor that takes one **String** parameter.

If you have a class where the **valueOf()** method or the string constructor does not exist or is inappropriate for an HTTP request, Jakarta RESTful Web Services provides the **javax.ws.rs.ext.ParamConverterProvider** and **javax.ws.rs.ext.ParamConverter** to help convert the message parameter value to the corresponding custom Java type. **ParamConverterProvider** must be either programmatically registered in a Jakarta RESTful Web Services runtime or must be annotated with **@Provider** annotation to be automatically discovered by the Jakarta RESTful Web Services runtime during a provider scanning phase.

For example: The steps below demonstrate how to create a custom POJO object. The conversion from message parameter value such as **@QueryParam**, **@PathParam**, **@MatrixParam**, **@HeaderParam** into POJO object is done by implementation of **ParamConverter** and **ParamConverterProvider** interfaces.

1. Create the custom POJO class.

```
public class POJO {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

2. Create the custom POJO Converter class.

```
public class POJOConverter implements ParamConverter<POJO> {
  public POJO fromString(String str) {
    System.out.println("FROM STRNG: " + str);
    POJO pojo = new POJO();
    pojo.setName(str);
    return pojo;
  }

  public String toString(POJO value) {
    return value.getName();
  }
}
```

3. Create the custom POJO Converter Provider class.

```
public class POJOConverterProvider implements ParamConverterProvider {
  @Override
  public <T> ParamConverter<T> getConverter(Class<T> rawType, Type genericType,
Annotation[] annotations) {
    if (!POJO.class.equals(rawType)) return null;
    return (ParamConverter<T>)new POJOConverter();
  }
}
```

4. Create the custom MyResource class.

```
@Path("/")
public class MyResource {
  @Path("{pojo}")
  @PUT
  public void put(@QueryParam("pojo") POJO q, @PathParam("pojo") POJO pp,
@MatrixParam("pojo") POJO mp,
    @HeaderParam("pojo") POJO hp) {
    ...
  }
}
```

Extending the Functionality of the ParamConverter

In the Jakarta RESTful Web Services semantics, a **ParamConverter** converts a single string that represents an individual object. RESTEasy extends the semantics to allow a **ParamConverter** to parse the string representation of multiple objects and generate a **List<T>**, **Set<T>**, **SortedSet<T>**, **array**, or any other multi-valued data structure.

For an example, consider the resource:

```
@Path("queryParam")
public static class TestResource {
   @GET
   @Path("")
   public Response conversion(@QueryParam("q") List<String> list) {
      return Response.ok(stringify(list)).build();
   }
}

private static <T> String stringify(List<T> list) {
   StringBuffer sb = new StringBuffer();
   for (T s : list) {
      sb.append(s).append(',');
   }
   return sb.toString();
}
```

Calling **TestResource** as follows, using the standard notation:

```
@Test
public void testQueryParamStandard() throws Exception {
   ResteasyClient client = new ResteasyClientBuilder().build();
   Invocation.Builder request = client.target("http://localhost:8081/queryParam?
q=20161217&q=20161218&q=20161219").request();
   Response response = request.get();
   System.out.println("response: " + response.readEntity(String.class));
}
```

results in **response: 20161217,20161218,20161219,**.

If you want to use a comma-separated notation instead, you can add:

```
public static class MultiValuedParamConverterProvider implements ParamConverterProvider
   @SuppressWarnings("unchecked")
   @Override
   public <T> ParamConverter<T> getConverter(Class<T> rawType, Type genericType, Annotation[]
annotations) {
      if (List.class.isAssignableFrom(rawType)) {
         return (ParamConverter<T>) new MultiValuedParamConverter();
      }
      return null;
   }
}

public static class MultiValuedParamConverter implements ParamConverter<List<?>> {
   @Override
   public List<?> fromString(String param) {
      if (param == null || param.trim().isEmpty()) {
```

```
        return null;
      }
      return parse(param.split(","));
    }

    @Override
    public String toString(List<?> list) {
      if (list == null || list.isEmpty()) {
        return null;
      }
      return stringify(list);
    }

    private static List<String> parse(String[] params) {
      List<String> list = new ArrayList<String>();
      for (String param : params) {
        list.add(param);
      }
      return list;
    }
}
```

Now you can call **TestResource** as follows:

```
@Test
public void testQueryParamCustom() throws Exception {
  ResteasyClient client = new ResteasyClientBuilder().build();
  Invocation.Builder request = client.target("http://localhost:8081/queryParam?
q=20161217,20161218,20161219").request();
  Response response = request.get();
  System.out.println("response: " + response.readEntity(String.class));
}
```

and get **response: 20161217,20161218,20161219,**.

In this case, the **MultiValuedParamConverter.fromString()** function creates and returns an **ArrayList**, so that the **TestResource.conversion()** function can be rewritten:

```
@Path("queryParam")
public static class TestResource {

  @GET
  @Path("")
  public Response conversion(@QueryParam("q") ArrayList<String> list) {
    return Response.ok(stringify(list)).build();
  }
}
```

Alternatively, **MultiValuedParamConverter** can be rewritten to return a **LinkList** and the parameter list in **TestResource.conversion()** can be either a **List** or a **LinkedList**.

Finally, note that this extension works for arrays as well. For example,

```
public static class Foo {
   private String foo;
```

```
    public Foo(String foo) {
        this.foo = foo;
    }
    public String getFoo() {
        return foo;
    }
}

public static class FooArrayParamConverter implements ParamConverter < Foo[] > {
    @Override
    public Foo[] fromString(String value) {
        String[] ss = value.split(",");
        Foo[] fs = new Foo[ss.length];
        int i = 0;
        for (String s: ss) {
            fs[i++] = new Foo(s);
        }
        return fs;
    }

    @Override
    public String toString(Foo[] values) {
        StringBuffer sb = new StringBuffer();
        for (int i = 0; i < values.length; i++) {
            sb.append(values[i].getFoo()).append(",");
        }
        if (sb.length() > 0) {
            sb.deleteCharAt(sb.length() - 1);
        }
        return sb.toString();
    }
}

@Provider
public static class FooArrayParamConverterProvider implements ParamConverterProvider {
    @SuppressWarnings("unchecked")
    @Override
    public < T > ParamConverter < T > getConverter(Class < T > rawType, Type genericType,
Annotation[] annotations) {
        if (rawType.equals(Foo[].class));
        return (ParamConverter < T > ) new FooArrayParamConverter();
    }
}

@Path("")
public static class ParamConverterResource {

    @GET
    @Path("test")
    public Response test(@QueryParam("foos") Foo[] foos) {
        return Response.ok(new FooArrayParamConverter().toString(foos)).build();
    }
}
```

java.util.Optional Parameter Types

RESTEasy offers several additional **java.util.Optional** parameter types. These parameter types act as wrapper object types. They allow users to input optional typed parameters, and eliminate all null checks by using methods like **Optional.orElse()**.

```
@Path("/double")
@GET
public String optDouble(@QueryParam("value") OptionalDouble value) {
    return Double.toString(value.orElse(4242.0));
}
```

The example above demonstrates that the **OptionalDouble** can be used as a parameter type. If a value is not provided in **@QueryParam**, then the default value will be returned. Optional parameters are supported for the following parameter types:

- **@QueryParam**
- **@MatrixParam**
- **@FormParam**
- **@HeaderParam**
- **@CookieParam**

2.7.7. Serializable Provider

Deserializing Java objects from untrusted sources is not safe. Therefore, **org.jboss.resteasy.plugins.providers.SerializableProvider** is disabled by default. It is not recommended to use this provider.

2.7.8. JSON Provider

2.7.8.1. JsonFilter Support in RESTEasy Jackson2

JsonFilter facilitates dynamic filtering by allowing you to annotate a class with **@JsonFilter**. The following example defines mapping from the **nameFilter** class to the filter instances, and then filtering out bean properties when serializing the instances to JSON format.

```
@JsonFilter(value="nameFilter")
public class Jackson2Product {
    protected String name;
    protected int id;
    public Jackson2Product() {
    }
    public Jackson2Product(final int id, final String name) {
        this.id = id;
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getId() {
```

```
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
}
```

@JsonFilter annotates the resource class to filter out the property that should not be serialized in the JSON response. To map the filter ID and instance, you must create another Jackson class and add the ID and filter instance map to it, as shown in the example below.

```
public class ObjectFilterModifier extends ObjectWriterModifier {
 public ObjectFilterModifier() {
 }
 @Override
 public ObjectWriter modify(EndpointConfigBase<?> endpoint,
  MultivaluedMap<String, Object> httpHeaders, Object valueToWrite,
  ObjectWriter w, JsonGenerator jg) throws IOException {

  FilterProvider filterProvider = new SimpleFilterProvider().addFilter(
   "nameFilter",
   SimpleBeanPropertyFilter.filterOutAllExcept("name"));
  return w.with(filterProvider);
 }
}
```

In the example above, the method **modify()** takes care of filtering all properties except the **name** property before writing the response. For this to work, RESTEasy must know about this mapping information. You can set the mapping information either in a **WriterInterceptor** or a servlet filter, as shown in the examples below.

Example: Setting ObjectFilterModifier Using WriterInterceptor

```
@Provider
public class JsonFilterWriteInterceptor implements WriterInterceptor{

 private ObjectFilterModifier modifier = new ObjectFilterModifier();
 @Override
 public void aroundWriteTo(WriterInterceptorContext context)
  throws IOException, WebApplicationException {
     //set a threadlocal modifier
          ObjectWriterInjector.set(modifier);
             context.proceed();
 }

}
```

Example: Setting ObjectFilterModifier Using Servlet Filter

```
public class ObjectWriterModifierFilter implements Filter {
 private static ObjectFilterModifier modifier = new ObjectFilterModifier();

 @Override
 public void init(FilterConfig filterConfig) throws ServletException {
 }
```

```
@Override
public void doFilter(ServletRequest request, ServletResponse response,
 FilterChain chain) throws IOException, ServletException {
    ObjectWriterInjector.set(modifier);
    chain.doFilter(request, response);
}

@Override
public void destroy() {
}

}
```

Now, RESTEasy can get the **ObjectFilterModifier** from the **ThreadLocal** variable and configure it to modify **ObjectWriter** before writing the response.

2.7.8.2. JSON serialization of time and duration objects

NOTE

Using RESTEasy to serialize time and duration objects, such as **LocalDateTime**, **LocalDate** or **Duration** classes without configuring the Jackson2 provider results in an error.

RESTEasy supports serialization of time and duration objects with the Jackson2 provider. The following example shows how to configure a Jackson2 provider to activate serialization:

Example: Configure a Jackson2 provider for serialization

```
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.ext.ContextResolver;
import javax.ws.rs.ext.Provider;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.json.JsonMapper;

@Provider
@Produces(MediaType.APPLICATION_JSON)
public class JacksonDatatypeJacksonProducer implements ContextResolver<ObjectMapper> {

   private final ObjectMapper json;

   public JacksonDatatypeJacksonProducer() throws Exception {
      this.json = JsonMapper.builder()
            .findAndAddModules()
            .build();
   }

   @Override
   public ObjectMapper getContext(Class<?> objectType) {
```

```
        return json;
    }
}
```

2.7.8.3. JSON Binding

RESTEasy supports both Jakarta JSON Binding and Jakarta JSON Processing. In accordance with the specification, entity providers for Jakarta JSON Binding take precedence over the ones for Jakarta JSON Processing for all types of entities except **JsonValue** and its sub-types.

The **JsonBindingProvider** property from **resteasy-json-binding-provider** module provides support for Jakarta JSON Binding. To satisfy Jakarta RESTful Web Services 2.1 requirements, the **JsonBindingProvider** provider takes precedence over the other providers for dealing with JSON payloads, in particular the Jackson payload.

For the same input, the JSON outputs from Jackson and Jakarta JSON Binding reference implementation can vary. Consequently, in order to retain backward compatibility, you can set the **resteasy.preferJacksonOverJsonB** context property to **true** and disable the **JsonBindingProvider** configuration for the current deployment.

JBoss EAP supports specifying the default value for the **resteasy.preferJacksonOverJsonB** context property by setting a system property with the same name. If no value is set for the context and system properties, it scans Jakarta RESTful Web Services deployments for Jackson annotations and sets the property to **true** if any of these annotations is found.

2.7.9. Jakarta XML Binding Providers

2.7.9.1. Jakarta XML Binding and XML Provider

RESTEasy provides Jakarta XML Binding provider support for XML.

@XmlHeader and @Stylesheet
RESTEasy provides setting an XML header using the **@org.jboss.resteasy.annotations.providers.jaxb.XmlHeader** annotation.

Example: Using the @XmlHeader Annotation

```
@XmlRootElement
public static class Thing {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}

@Path("/test")
public static class TestService {

  @GET
  @Path("/header")
```

```
    @Produces("application/xml")
    @XmlHeader("<?xml-stylesheet type='text/xsl' href='${baseuri}foo.xsl' ?>")
    public Thing get() {
      Thing thing = new Thing();
      thing.setName("bill");
      return thing;
    }
}
```

The **@XmlHeader** ensures that the XML output has an XML-stylesheet header.

RESTEasy has a convenient annotation for stylesheet headers.

Example: Using the @Stylesheet Annotation

```
@XmlRootElement
public static class Thing {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}

@Path("/test")
public static class TestService {

  @GET
  @Path("/stylesheet")
  @Produces("application/xml")
  @Stylesheet(type="text/css", href="${basepath}foo.xsl")
  @Junk
  public Thing getStyle() {
    Thing thing = new Thing();
    thing.setName("bill");
    return thing;
  }
}
```

2.7.9.2. Jakarta XML Binding and JSON Provider

RESTEasy allows you to marshal Jakarta XML Binding annotated POJOs to and from JSON using the JSON provider. This provider wraps the Jackson JSON library to accomplish this task. It has a Java Beans based model and APIs similar to Jakarta XML Binding.

While Jackson already includes Jakarta RESTful Web Services integration, it was expanded by RESTEasy. To include it in your project, you need to update the Maven dependencies.

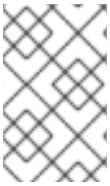
Maven Dependencies for Jackson

```
<dependency>
```

```
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jackson2-provider</artifactId>
    <version>${version.org.jboss.resteasy}</version>
    <scope>provided</scope>
</dependency>
```

NOTE

The default JSON provider for RESTEasy is Jackson2. Previous versions of JBoss EAP included the Jackson1 JSON provider. For more details on migrating your existing applications from the Jackson1 provider, see the JBoss EAP *Migration Guide*. If you still want to use the Jackson1 provider, you have to explicitly update the Maven dependencies to obtain it.



NOTE

The default JSON provider for RESTEasy in previous versions of JBoss EAP was Jettison, but is now deprecated in JBoss EAP 7. For more details, see the JBoss EAP *Migration Guide*.

Example JSON Provider

```
@XmlRootElement
public static class Thing {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}

@Path("/test")
public static class TestService {
  @GET
  @Path("/thing")
  @Produces("application/json")
  public Thing get() {
    Thing thing = new Thing();
    thing.setName("the thing");
    return thing;
  }
}
```

2.7.9.2.1. Jackson Module Support for Java 8

This section provides the Maven dependencies and shows how to register the Jackson modules needed to support Java 8 features, when the core Jackson modules do not require Java 8 runtime environment. These Jackson modules include:

- Java 8 data types

- Java 8 date/time

Add the following Maven dependencies:

```
<dependency>
    <groupId>com.fasterxml.jackson.datatype</groupId>
    <artifactId>jackson-datatype-jdk8</artifactId>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.datatype</groupId>
    <artifactId>jackson-datatype-jsr310</artifactId>
</dependency>
```

You can find and register all the modules using **findAndRegisterModules()** or **ObjectMapper.registerModule()**, as shown in the examples below:

```
ObjectMapper mapper = new ObjectMapper();
mapper.findAndRegisterModules();
```

```
ObjectMapper mapper = new ObjectMapper()
  .registerModule(new ParameterNamesModule())
  .registerModule(new Jdk8Module())
  .registerModule(new JavaTimeModule());
```

Example: Duration Data Type

```
@GET
@Path("/duration")
@Produces(MediaType.APPLICATION_JSON)
public Duration getDuration() {
    return Duration.ofSeconds(5, 6);
}
```

Example: Optional Data Type

```
@GET
@Path("/optional/{nullParam}")
@Produces(MediaType.APPLICATION_JSON)
public Optional<String> getOptional(@PathParam("nullParam") boolean nullParameter) {
    return nullParameter ? Optional.<String>empty() : Optional.of("info@example.com");
}
```

You must use the custom implementation of the **ContextResolver** in order to use these Jackson modules in RESTEasy.

```
@Provider
@Produces(MediaType.APPLICATION_JSON)
public class JacksonDatatypeJacksonProducer implements ContextResolver<ObjectMapper> {
    private final ObjectMapper json;
    public JacksonDatatypeJacksonProducer() throws Exception {
        this.json = new ObjectMapper()
                .findAndRegisterModules()
                .configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false)
```

```
                .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
    }
    @Override
    public ObjectMapper getContext(Class<?> objectType) {
        return json;
    }
}
```

2.7.9.2.2. Switching the Default Jackson Provider

JBoss EAP 7 includes Jackson 2.6.x or greater and **resteasy-jackson2-provider** is now the default Jackson provider.

To switch to the default **resteasy-jackson-provider** that was included in the previous release of JBoss EAP, exclude the new provider and add a dependency for the previous provider in the **jboss-deployment-structure.xml** application deployment descriptor file.

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-deployment-structure>
  <deployment>
    <exclusions>
      <module name="org.jboss.resteasy.resteasy-jackson2-provider"/>
    </exclusions>
    <dependencies>
      <module name="org.jboss.resteasy.resteasy-jackson-provider" services="import"/>
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

2.7.10. Creating Jakarta XML Binding Decorators

RESTEasy's Jakarta XML Binding providers have a pluggable way to decorate Marshaller and Unmarshaller instances. You can create an annotation that can trigger either a Marshaller or Unmarshaller instance, which can be used to decorate methods.

Create a Jakarta XML Binding Decorator with RESTEasy

1. Create the Processor class.

    a. Create a class that implements **DecoratorProcessor<Target, Annotation>**. The target is either the Jakarta XML Binding Marshaller or Unmarshaller class. The annotation is created in step two.

    b. Annotate the class with **@DecorateTypes**, and declare the MIME types the decorator should decorate.

    c. Set properties or values within the decorate function.

    Example: Processor Class

```
import org.jboss.resteasy.core.interception.DecoratorProcessor;
import org.jboss.resteasy.annotations.DecorateTypes;
import javax.xml.bind.Marshaller;
import javax.xml.bind.PropertyException;
import javax.ws.rs.core.MediaType;
```

```
import javax.ws.rs.Produces;
import java.lang.annotation.Annotation;

@DecorateTypes({"text/*+xml", "application/*+xml"})
public class PrettyProcessor implements DecoratorProcessor<Marshaller, Pretty> {
   public Marshaller decorate(Marshaller target, Pretty annotation,
     Class type, Annotation[] annotations, MediaType mediaType) {
   target.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
   }
}
```

2. Create the annotation.

   a. Create a custom interface that is annotated with the **@Decorator** annotation.

   b. Declare the processor and target for the **@Decorator** annotation. The processor is created in step one. The target is either the Jakarta XML Binding **Marshaller** or **Unmarshaller** class.

   Example: Custom Interface with @Decorator Annotation

```
import org.jboss.resteasy.annotations.Decorator;

@Target({ElementType.TYPE, ElementType.METHOD, ElementType.PARAMETER,
ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Decorator(processor = PrettyProcessor.class, target = Marshaller.class)
public @interface Pretty {}
```

3. Add the annotation created in step two to a function so that either the input or output is decorated when it is marshaled.

You have now created a Jakarta XML Binding decorator, which can be applied within a Jakarta RESTful Web Services web service.

2.7.11. Multipart Providers in Jakarta RESTful Web Services

The multipart MIME format is used to pass lists of content bodies embedded in one message. One example of a multipart MIME format is the **multipart/form-data** MIME type. This is often found in web application HTML form documents and is generally used to upload files. The **form-data** format in this MIME type is the same as other multipart formats, except that each inlined piece of content has a name associated with it.

RESTEasy allows for the **multipart/form-data** and **multipart/*** MIME types. RESTEasy also provides a custom API for reading and writing multipart types as well as marshalling arbitrary **List** (for any multipart type) and **Map** (multipart/form-data only) objects.

IMPORTANT

There are a lot of frameworks doing multipart parsing automatically with the help of filters and interceptors, such as **org.jboss.seam.web.MultipartFilter** in Seam or **org.springframework.web.multipart.MultipartResolver** in Spring. However, the incoming multipart request stream can be parsed only once. RESTEasy users working with multipart should make sure that nothing parses the stream before RESTEasy gets it.

2.7.11.1. Input with Multipart Data

When writing a Jakarta RESTful Web Services service, RESTEasy provides the **org.jboss.resteasy.plugins.providers.multipart.MultipartInput** interface to allow you to read in any multipart MIME type.

```
package org.jboss.resteasy.plugins.providers.multipart;

public interface MultipartInput {

   List<InputPart> getParts();
   String getPreamble();

   // You must call close to delete any temporary files created
   // Otherwise they will be deleted on garbage collection or on JVM exit
   void close();
}

public interface InputPart {

   MultivaluedMap<String, String> getHeaders();
   String getBodyAsString();
   <T> T getBody(Class<T> type, Type genericType) throws IOException;
   <T> T getBody(org.jboss.resteasy.util.GenericType<T> type) throws IOException;
   MediaType getMediaType();
   boolean isContentTypeFromMessage();
}
```

MultipartInput is a simple interface that allows you to get access to each part of the multipart message. Each part is represented by an **InputPart** interface, and each part has a set of headers associated with it. You can unmarshal the part by calling one of the **getBody()** methods. The **genericType** parameter can be **null**, but the **type** parameter must be set. RESTEasy will find a **MessageBodyReader** based on the media type of the part as well as the type information you pass in.

2.7.11.1.1. Input with multipart/mixed

Example: Unmarshalling Parts

```
@Path("/multipart")
public class MyService {

   @PUT
   @Consumes("multipart/mixed")
   public void put(MultipartInput input) {
      List<Customer> customers = new ArrayList...;
      for (InputPart part : input.getParts()) {
         Customer cust = part.getBody(Customer.class, null);
         customers.add(cust);
      }
      input.close();
   }
}
```

NOTE

The above example assumes the **Customer** class is annotated with Jakarta XML Binding.

Sometimes you may want to unmarshal a body part that is sensitive to generic type metadata. In this case you can use the **org.jboss.resteasy.util.GenericType** class.

Example: Unmarshalling a Type Sensitive to Generic Type Metadata

```
@Path("/multipart")
public class MyService {

   @PUT
   @Consumes("multipart/mixed")
   public void put(MultipartInput input) {
      for (InputPart part : input.getParts()) {
         List<Customer> cust = part.getBody(new GenericType<List<Customer>>() {});
      }
      input.close();
   }
}
```

Use of **GenericType** is required because it is the only way to obtain generic type information at runtime.

2.7.11.1.2. Input with multipart/mixed and java.util.List

If the body parts are uniform, you do not have to manually unmarshal each and every part. You can just provide a **java.util.List** as your input parameter. It must have the type it is unmarshalling with the generic parameter of the **List** type declaration.

Example: Unmarshalling a List of Customers

```
@Path("/multipart")
public class MyService {

   @PUT
   @Consumes("multipart/mixed")
   public void put(List<Customer> customers) {
      ...
   }
}
```

NOTE

The above example assumes the **Customer** class is annotated with Jakarta XML Binding.

2.7.11.1.3. Input with multipart/form-data

When writing a Jakarta RESTful Web Services service, RESTEasy provides an interface that allows you to read in **multipart/form-data** MIME type. **multipart/form-data** is often found in web application HTML form documents and is generally used to upload files. The **form-data** format is the same as other multipart formats, except that each inlined piece of content has a name associated with it. The interface used for form-data input is **org.jboss.resteasy.plugins.providers.multipart.MultipartFormDataInput**.

Example: MultipartFormDataInput Interface

```
public interface MultipartFormDataInput extends MultipartInput {

    @Deprecated
    Map<String, InputPart> getFormData();
    Map<String, List<InputPart>> getFormDataMap();
    <T> T getFormDataPart(String key, Class<T> rawType, Type genericType) throws IOException;
    <T> T getFormDataPart(String key, GenericType<T> type) throws IOException;
}
```

It works in much the same way as **MultipartInput** described earlier.

2.7.11.1.4. java.util.Map with multipart/form-data

With form-data, if the body parts are uniform, you do not have to manually unmarshal each and every part. You can just provide a **java.util.Map** as your input parameter. It must have the type it is unmarshalling with the generic parameter of the **List** type declaration.

Example: Unmarshalling a Map of Customer objects

```
@Path("/multipart")
public class MyService {

    @PUT
    @Consumes("multipart/form-data")
    public void put(Map<String, Customer> customers) {
        ...
    }
}
```

NOTE

The above example assumes the **Customer** class is annotated with Jakarta XML Binding.

2.7.11.1.5. Input with multipart/related

When writing a Jakarta RESTful Web Services service, RESTEasy provides an interface that allows you to read in **multipart/related** MIME type. A **multipart/related** is used to indicate that message parts should not be considered individually but rather as parts of an aggregate whole and is defined by RFC 2387.

One example usage for **multipart/related** is to send a web page complete with images in a single message. Every **multipart/related** message has a root/start part that references the other parts of the message. The parts are identified by their **Content-ID** headers. The interface used for related input is **org.jboss.resteasy.plugins.providers.multipart.MultipartRelatedInput**.

Example: MultipartRelatedInput Interface

```
public interface MultipartRelatedInput extends MultipartInput {

    String getType();
    String getStart();
    String getStartInfo();
```

```
    InputPart getRootPart();
    Map<String, InputPart> getRelatedMap();
}
```

It works in much the same way as **MultipartInput**.

2.7.11.2. Output with Multipart Data

RESTEasy provides a simple API to output multipart data.

```
package org.jboss.resteasy.plugins.providers.multipart;

public class MultipartOutput {

    public OutputPart addPart(Object entity, MediaType mediaType)
    public OutputPart addPart(Object entity, GenericType type, MediaType mediaType)
    public OutputPart addPart(Object entity, Class type, Type genericType, MediaType mediaType)
    public List<OutputPart> getParts()
    public String getBoundary()
    public void setBoundary(String boundary)
}

public class OutputPart {

    public MultivaluedMap<String, Object> getHeaders()
    public Object getEntity()
    public Class getType()
    public Type getGenericType()
    public MediaType getMediaType()
}
```

To output multipart data, you need to create a **MultipartOutput** object and call the **addPart()** method. RESTEasy will automatically find a **MessageBodyWriter** to marshal your entity objects. Similar to **MultipartInput**, sometimes you might have marshalling that is sensitive to generic type metadata. In that case, use the **GenericType**. Usually, passing in an object and its **MediaType** should be enough.

Example: Returning a multipart/mixed Format

```
@Path("/multipart")
public class MyService {

    @GET
    @Produces("multipart/mixed")
    public MultipartOutput get() {

        MultipartOutput output = new MultipartOutput();
        output.addPart(new Customer("bill"), MediaType.APPLICATION_XML_TYPE);
        output.addPart(new Customer("monica"), MediaType.APPLICATION_XML_TYPE);
        return output;
    }
}
```

NOTE

The above example assumes the **Customer** class is annotated with Jakarta XML Binding.

2.7.11.2.1. Multipart Output with java.util.List

If the body parts are uniform, you do not have to manually marshal each and every part or even use a **MultipartOutput** object. You can provide a **java.util.List** which must have the generic type it is marshalling with the generic parameter of the **List** type declaration. You must also annotate the method with the **@PartType** annotation to specify the media type of each part.

Example: Returning a List of Customer Objects

```
@Path("/multipart")
public class MyService {

   @GET
   @Produces("multipart/mixed")
   @PartType("application/xml")
   public List<Customer> get(){
      ...
   }
}
```

NOTE

The above example assumes the **Customer** class is annotated with Jakarta XML Binding.

2.7.11.2.2. Output with multipart/form-data

RESTEasy provides a simple API to output **multipart/form-data**.

```
package org.jboss.resteasy.plugins.providers.multipart;

public class MultipartFormDataOutput extends MultipartOutput {

   public OutputPart addFormData(String key, Object entity, MediaType mediaType)
   public OutputPart addFormData(String key, Object entity, GenericType type, MediaType
mediaType)
   public OutputPart addFormData(String key, Object entity, Class type, Type genericType,
MediaType mediaType)
   public Map<String, OutputPart> getFormData()
}
```

To output **multipart/form-data**, you must create a **MultipartFormDataOutput** object and call the **addFormData()** method. RESTEasy will automatically find a **MessageBodyWriter** to marshal your entity objects. Similar to **MultipartInput**, sometimes you might have marshalling that is sensitive to generic type metadata. In that case, use the **GenericType**. Usually, passing in an object and its **MediaType** should be enough.

Example: Returning multipart/form-data Format

```
@Path("/form")
public class MyService {
```

```
    @GET
    @Produces("multipart/form-data")
    public MultipartFormDataOutput get() {

        MultipartFormDataOutput output = new MultipartFormDataOutput();
        output.addPart("bill", new Customer("bill"), MediaType.APPLICATION_XML_TYPE);
        output.addPart("monica", new Customer("monica"), MediaType.APPLICATION_XML_TYPE);
        return output;
    }
}
```

NOTE

The above example assumes the **Customer** class is annotated with Jakarta XML Binding.

2.7.11.2.3. Multipart FormData Output with java.util.Map

If the body parts are uniform, you do not have to manually marshal every part or use a **MultipartFormDataOutput** object. You can just provide a **java.util.Map** which must have the generic type it is marshalling with the generic parameter of the **Map** type declaration. You must also annotate the method with the **@PartType** annotation to specify the media type of each part.

Example: Returning a Map of Customer Objects

```
@Path("/multipart")
public class MyService {

    @GET
    @Produces("multipart/form-data")
    @PartType("application/xml")
    public Map<String, Customer> get() {
        ...
    }
}
```

NOTE

The above example assumes the **Customer** class is annotated with Jakarta XML Binding.

2.7.11.2.4. Output with multipart/related

RESTEasy provides a simple API to output **multipart/related**.

```
package org.jboss.resteasy.plugins.providers.multipart;

public class MultipartRelatedOutput extends MultipartOutput {

    public OutputPart getRootPart()
    public OutputPart addPart(Object entity, MediaType mediaType,
        String contentId, String contentTransferEncoding)
    public String getStartInfo()
    public void setStartInfo(String startInfo)
}
```

To output **multipart/related**, you must create a **MultipartRelatedOutput** object and call the **addPart()** method. The first added part is used as the root part of the **multipart/related** message, and RESTEasy automatically finds a **MessageBodyWriter** to marshal your entity objects. Similar to **MultipartInput**, sometimes you might have marshalling that is sensitive to generic type metadata. In that case, use the **GenericType**. Usually, passing in an object and its **MediaType** should be enough.

Example: Returning multipart/related Format Sending Two Images

```
@Path("/related")
public class MyService {

   @GET
   @Produces("multipart/related")
   public MultipartRelatedOutput get() {

      MultipartRelatedOutput output = new MultipartRelatedOutput();
      output.setStartInfo("text/html");

      Map<String, String> mediaTypeParameters = new LinkedHashMap<String, String>();
      mediaTypeParameters.put("charset", "UTF-8");
      mediaTypeParameters.put("type", "text/html");
      output.addPart(
         "<html><body>\n"
         + "This is me: <img src='cid:http://example.org/me.png' />\n"
         + "<br />This is you: <img src='cid:http://example.org/you.png' />\n"
         + "</body></html>",
         new MediaType("text", "html", mediaTypeParameters),
         "<mymessage.xml@example.org>", "8bit");
      output.addPart("// binary octets for me png",
         new MediaType("image", "png"), "<http://example.org/me.png>",
         "binary");
      output.addPart("// binary octets for you png", new MediaType(
         "image", "png"),
         "<http://example.org/you.png>", "binary");
      client.putRelated(output);
      return output;
   }
}
```

NOTE

The above example assumes the **Customer** class is annotated with Jakarta XML Binding.

2.7.11.3. Mapping Multipart Forms to POJOs

If you have an exact knowledge of your multipart/form-data packets, you can map them to and from a POJO class. This is accomplished using the **org.jboss.resteasy.annotations.providers.multipart.MultipartForm** annotation (**@MultipartForm**) and the Jakarta RESTful Web Services **@FormParam** annotation. To do so, you need to define a POJO with at least a default constructor and annotate its fields and/or properties with **@FormParams**. These **@FormParams** must also be annotated with **org.jboss.resteasy.annotations.providers.multipart.PartType** (**@PartType**) if you are creating output.

Example: Mapping Multipart Forms to a POJO

```
public class CustomerProblemForm {

   @FormParam("customer")
   @PartType("application/xml")
   private Customer customer;

   @FormParam("problem")
   @PartType("text/plain")
   private String problem;

   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer cust) { this.customer = cust; }
   public String getProblem() { return problem; }
   public void setProblem(String problem) { this.problem = problem; }
}
```

After defining your POJO class you can then use it to represent **multipart/form-data**.

Example: Submit CustomerProblemForm

```
@Path("portal")
public interface CustomerPortal {

   @Path("issues/{id}")
   @Consumes("multipart/form-data")
   @PUT
   public void putProblem(@MultipartForm CustomerProblemForm,
                          @PathParam("id") int id);
}

// Somewhere using it:
{
   CustomerPortal portal = ProxyFactory.create(CustomerPortal.class, "http://example.com");
   CustomerProblemForm form = new CustomerProblemForm();
   form.setCustomer(...);
   form.setProblem(...);

   portal.putProblem(form, 333);
}
```

The **@MultipartForm** annotation tells RESTEasy that the object has **@FormParam** and that it should be marshaled from that. You can also use the same object to receive multipart data.

Example: Receive CustomerProblemForm

```
@Path("portal")
public class CustomerPortalServer {

   @Path("issues/{id})
   @Consumes("multipart/form-data")
   @PUT
   public void putIssue(@MultipartForm CustomerProblemForm,
```

```
                    @PathParam("id") int id) {
        ... write to database...
    }
}
```

2.7.11.4. XML-binary Optimized Packaging (XOP)

If you have a Jakarta XML Binding annotated POJO that also holds some binary content, you may choose to send it in such a way that the binary does not need to be encoded in any way such as base64 or hex. This is accomplished using XOP and results in faster transport while still using the convenient POJO.

RESTEasy allows for XOP messages packaged as **multipart/related**.

To configure XOP, you first need a Jakarta XML Binding annotated POJO.

Example: Jakarta XML Binding POJO

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public static class Xop {

    private Customer bill;
    private Customer monica;

    @XmlMimeType(MediaType.APPLICATION_OCTET_STREAM)
    private byte[] myBinary;

    @XmlMimeType(MediaType.APPLICATION_OCTET_STREAM)
    private DataHandler myDataHandler;

    // methods, other fields ...
}
```

NOTE

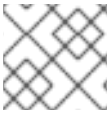
@XmlMimeType tells Jakarta XML Binding the mime type of the binary content. This is not required to do XOP packaging but it is recommended to be set if you know the exact type.

In the above POJO **myBinary** and **myDataHandler** will be processed as binary attachments while the whole XOP object will be sent as XML. In place of the binaries, only their references will be generated. **javax.activation.DataHandler** is the most general supported type. If you need a **java.io.InputStream** or a **javax.activation.DataSource**, you need to use the **DataHandler**. **java.awt.Image** and **javax.xml.transform.SourceSome** are available as well.

Example: Client Sending Binary Content with XOP

```
// our client interface:
@Path("mime")
public static interface MultipartClient {
    @Path("xop")
    @PUT
    @Consumes(MultipartConstants.MULTIPART_RELATED)
```

```
    public void putXop(@XopWithMultipartRelated Xop bean);
}

// Somewhere using it:
{
    MultipartClient client = ProxyFactory.create(MultipartClient.class,
        "http://www.example.org");
    Xop xop = new Xop(new Customer("bill"), new Customer("monica"),
        "Hello Xop World!".getBytes("UTF-8"),
        new DataHandler(new ByteArrayDataSource("Hello Xop World!".getBytes("UTF-8"),
        MediaType.APPLICATION_OCTET_STREAM)));
    client.putXop(xop);
}
```



NOTE

The above example assumes the **Customer** class is annotated with Jakarta XML Binding.

The **@Consumes(MultipartConstants.MULTIPART_RELATED)** is used to tell RESTEasy that you want to send **multipart/related** packages, which is the container format that holds the XOP message. **@XopWithMultipartRelated** is used to tell RESTEasy that you want to make XOP messages.

Example: RESTEasy Server for Receiving XOP

```
@Path("/mime")
public class XopService {
    @PUT
    @Path("xop")
    @Consumes(MultipartConstants.MULTIPART_RELATED)
    public void putXopWithMultipartRelated(@XopWithMultipartRelated Xop xop) {
        // do very important things here
    }
}
```

@Consumes(MultipartConstants.MULTIPART_RELATED) is used to tell RESTEasy that you want to read **multipart/related** packages. **@XopWithMultipartRelated** is used to tell RESTEasy that you want to read XOP messages. You can configure a RESTEasy server to produce XOP values in a similar way by adding a **@Produces** annotation and returning the appropriate type.

2.7.11.5. Overwriting the Default Fallback Content Type for Multipart Messages

By default, if no **Content-Type** header is present in a part, **text/plain; charset=us-ascii** is used as a fallback. This is defined by the MIME RFC. However some web clients, such as many browsers, may send **Content-Type** headers for the file parts, but not for all fields in a **multipart/form-data** request. This can cause character encoding and unmarshalling errors on the server side. The **PreProcessInterceptor** infrastructure of RESTEasy can be used to correct this issue. You can use it to define another, non-RFC compliant fallback value, dynamically per request.

Example: Setting * / *; charset=UTF-8 as the Default Fallback

```
import org.jboss.resteasy.plugins.providers.multipart.InputPart;

@Provider
@ServerInterceptor
```

```
public class ContentTypeSetterPreProcessorInterceptor implements PreProcessInterceptor {

   public ServerResponse preProcess(HttpRequest request, ResourceMethod method)
         throws Failure, WebApplicationException {
      request.setAttribute(InputPart.DEFAULT_CONTENT_TYPE_PROPERTY, "*/*; charset=UTF-
8");
      return null;
   }
}
```

2.7.11.6. Overwriting the Content Type for Multipart Messages

Using an interceptor and the **InputPart.DEFAULT_CONTENT_TYPE_PROPERTY attribute** allows you to set a default **Content-Type**. You can also override the **Content-Type** in any input part by calling **org.jboss.resteasy.plugins.providers.multipart.InputPart.setMediaType()**.

Example: Overriding the Content-Type

```
@POST
@Path("query")
@Consumes(MediaType.MULTIPART_FORM_DATA)
@Produces(MediaType.TEXT_PLAIN)
public Response setMediaType(MultipartInput input) throws IOException {

   List<InputPart> parts = input.getParts();
   InputPart part = parts.get(0);
   part.setMediaType(MediaType.valueOf("application/foo+xml"));
   String s = part.getBody(String.class, null);
   ...
}
```

2.7.11.7. Overwriting the Default Fallback charset for Multipart Messages

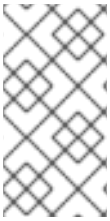
In some cases, part of a multipart message may have a **Content-Type** header with no **charset** parameter. If the **InputPart.DEFAULT_CONTENT_TYPE_PROPERTY** property is set and the value has a **charset** parameter, that value will be appended to an existing **Content-Type** header that has no **charset** parameter.

You can also specify a default **charset** using the constant **InputPart.DEFAULT_CHARSET_PROPERTY** (**resteasy.provider.multipart.inputpart.defaultCharset**).

Example: Specifying a Default charset

```
import org.jboss.resteasy.plugins.providers.multipart.InputPart;

@Provider
@ServerInterceptor
public class ContentTypeSetterPreProcessorInterceptor implements PreProcessInterceptor {

   public ServerResponse preProcess(HttpRequest request, ResourceMethod method)
         throws Failure, WebApplicationException {
      request.setAttribute(InputPart.DEFAULT_CHARSET_PROPERTY, "UTF-8");
```

```
        return null;
    }
}
```



NOTE

If both **InputPart.DEFAULT_CONTENT_TYPE_PROPERTY** and **InputPart.DEFAULT_CHARSET_PROPERTY** are set, then the value of **InputPart.DEFAULT_CHARSET_PROPERTY** will override any charset in the value of **InputPart.DEFAULT_CONTENT_TYPE_PROPERTY**.

2.7.11.8. Send Multipart Entity with RESTEasy Client

In addition to configuring multipart providers, you can also configure the RESTEasy client to send multipart data.

Using RESTEasy Client Classes
To use RESTEasy client classes in your application, you must add the Maven dependencies to your project's POM file.

Example: Maven Dependencies

```
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-client</artifactId>
  <version>${version.org.jboss.resteasy}</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-multipart-provider</artifactId>
  <version>${version.org.jboss.resteasy}</version>
  <scope>provided</scope>
</dependency>
```

Sending Multipart Data Using the RESTEasy Client
To send multipart data, you must first configure a RESTEasy Client and construct a **org.jboss.resteasy.plugins.providers.multipart.MultipartFormDataOutput** object to contain your multipart data. You can then use the client to send that **MultipartFormDataOutput** object as a **javax.ws.rs.core.GenericEntity**.

Example: RESTEasy Client

```
ResteasyClient client = new ResteasyClientBuilder().build();
ResteasyWebTarget target = client.target("http://foo.com/resource");

MultipartFormDataOutput formOutputData = new MultipartFormDataOutput();
formOutputData.addFormData("part1", "this is part 1", MediaType.TEXT_PLAIN);
formOutputData.addFormData("part2", "this is part 2", MediaType.TEXT_PLAIN);

GenericEntity<MultipartFormDataOutput> data = new GenericEntity<MultipartFormDataOutput>
(formOutputData) { };

Response response = target.request().put(Entity.entity(data,
```

```
MediaType.MULTIPART_FORM_DATA_TYPE));

response.close();
```

2.7.12. RESTEasy Atom Support

The RESTEasy Atom API and Provider is a simple object model that RESTEasy defines to represent Atom. The main classes for the API are in the **org.jboss.resteasy.plugins.providers.atom** package. RESTEasy uses Jakarta XML Binding to marshal and unmarshal the API. The provider is Jakarta XML Binding based, and is not limited to sending Atom objects using XML. All Jakarta XML Binding providers that RESTEasy has can be reused by the Atom API and provider, including JSON.

```
import org.jboss.resteasy.plugins.providers.atom.Content;
import org.jboss.resteasy.plugins.providers.atom.Entry;
import org.jboss.resteasy.plugins.providers.atom.Feed;
import org.jboss.resteasy.plugins.providers.atom.Link;
import org.jboss.resteasy.plugins.providers.atom.Person;

@Path("atom")
public class MyAtomService {

   @GET
   @Path("feed")
   @Produces("application/atom+xml")
   public Feed getFeed() throws URISyntaxException {
      Feed feed = new Feed();
      feed.setId(new URI("http://example.com/42"));
      feed.setTitle("My Feed");
      feed.setUpdated(new Date());
      Link link = new Link();
      link.setHref(new URI("http://localhost"));
      link.setRel("edit");
      feed.getLinks().add(link);
      feed.getAuthors().add(new Person("John Brown"));
      Entry entry = new Entry();
      entry.setTitle("Hello World");
      Content content = new Content();
      content.setType(MediaType.TEXT_HTML_TYPE);
      content.setText("Nothing much");
      entry.setContent(content);
      feed.getEntries().add(entry);
      return feed;
   }
}
```

2.7.12.1. Using Jakarta XML Binding with Atom Provider

The **org.jboss.resteasy.plugins.providers.atom.Content** class allows you to unmarshal and marshal Jakarta XML Binding annotated objects that are the body of the content.

Example: Entry with a Customer

```
@XmlRootElement(namespace = "http://jboss.org/Customer")
@XmlAccessorType(XmlAccessType.FIELD)
```

```
public class Customer {
  @XmlElement
  private String name;

  public Customer() {
  }

  public Customer(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}

@Path("atom")
public static class AtomServer {
  @GET
  @Path("entry")
  @Produces("application/atom+xml")
  public Entry getEntry() {
    Entry entry = new Entry();
    entry.setTitle("Hello World");
    Content content = new Content();
    content.setJAXBObject(new Customer("bill"));
    entry.setContent(content);
    return entry;
  }
}
```

The **Content.setJAXBObject()** method lets you specify the content object you send to Jakarta XML Binding to marshal appropriately. If you are using a different base format other than XML, that is **application/atom+json**, the attached Jakarta XML Binding object is marshalled in the same format. If you have an Atom document as input, you can also extract Jakarta XML Binding objects from **Content** using the **Content.getJAXBObject(Class clazz)** method.

Example: Atom Document Extracting a Customer Object

```
@Path("atom")
public static class AtomServer {
  @PUT
  @Path("entry")
  @Produces("application/atom+xml")
  public void putCustomer(Entry entry) {
    Content content = entry.getContent();
    Customer cust = content.getJAXBObject(Customer.class);
  }
}
```

2.7.13. YAML Provider

WARNING

The **resteasy-yaml-provider** module is not supported. Its use is not recommended due to a security issue in the **SnakeYAML** library used by RESTEasy for unmarshalling.

RESTEasy comes with built in support for YAML using the **SnakeYAML** library.

In releases prior to JBoss EAP 7.1, the YAML provider setting was enabled by default and you only needed to configure the Maven dependencies for YAML to use it in your application. Starting with JBoss EAP 7.1, the YAML provider is disabled by default and must be explicitly enabled in the application.

Enable the YAML Provider
To enable the YAML provider in your application, follow these steps:

1. Create or update a file named **javax.ws.rs.ext.Providers**.
2. Add the following content to the file.

```
org.jboss.resteasy.plugins.providers.YamlProvider
```

3. Place the file in the **META-INF/services/** folder of your WAR or JAR file.

YAML Provider Maven Dependencies
To use the YAML provider in your application, you must add the **snakeyaml** JAR dependencies to the project POM file of your application.

Example: Maven Dependencies for YAML

```
<dependency>
   <groupId>org.jboss.resteasy</groupId>
   <artifactId>resteasy-yaml-provider</artifactId>
   <version>${version.org.jboss.resteasy}</version>
   <scope>provided</scope>
</dependency>

<dependency>
   <groupId>org.yaml</groupId>
   <artifactId>snakeyaml</artifactId>
   <version>${version.org.yaml.snakeyaml}</version>
</dependency>
```

YAML Provider Code Example
The YAML provider recognizes three mime types:

- text/x-yaml
- text/yaml
- application/x-yaml

The following is an example of how to use YAML in a resource method.

Example: Resource Producing YAML

```
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("/yaml")
public class YamlResource {

  @GET
  @Produces("text/x-yaml")
  public MyObject getMyObject() {
    return createMyObject();
  }
...
}
```

2.8. USING JAKARTA JSON PROCESSING

Jakarta JSON Processing is defined in the Jakarta JSON Processing 1.1 specification.

Jakarta JSON Processing defines an API to process JSON. JBoss EAP has support for **javax.json.JsonObject**, **javax.json.JsonArray**, and **javax.json.JsonStructure** as request or response entities.

NOTE

Jakarta JSON Processing is different from JSON with Padding (JSONP).

NOTE

Jakarta JSON Processing will not conflict with Jackson if they are on the same classpath.

To create a **JsonObject**, use the **JsonObjectBuilder** by calling **Json.createObjectBuilder()** and building the JSON object.

Example: Create javax.json.JsonObject

```
JsonObject obj = Json.createObjectBuilder().add("name", "Bill").build();
```

Example: Corresponding JSON for javax.json.JsonObject

```
{
  "name":"Bill"
}
```

To create a **JsonArray**, use the **JsonArrayBuilder** by calling **Json.createArrayBuilder()** and building the JSON array.

Example: Create javax.json.JsonArray

```
JsonArray array =
  Json.createArrayBuilder()
    .add(Json.createObjectBuilder().add("name", "Bill").build())
    .add(Json.createObjectBuilder().add("name", "Monica").build()).build();
```

Example: Corresponding JSON for javax.json.JsonArray

```
[
  {
  "name":"Bill"
  },
  {
  "name":"Monica"
  }
]
```

JsonStructure is a parent class of **JsonObject** and **JsonArray**.

Example: Create javax.json.JsonStructure

```
JsonObject obj = Json.createObjectBuilder().add("name", "Bill").build();

JsonArray array =
  Json.createArrayBuilder()
    .add(Json.createObjectBuilder().add("name", "Bill").build())
    .add(Json.createObjectBuilder().add("name", "Monica").build()).build();

JsonStructure sObj = (JsonStructure) obj;
JsonStructure sArray = (JsonStructure) array;
```

You can use **JsonObject**, **JsonArray**, and **JsonStructure** directly in Jakarta RESTful Web Services resources.

Example: Jakarta RESTful Web Services Resources with Jakarta JSON Processing

```
@Path("object")
@POST
@Produces("application/json")
@Consumes("application/json")
public JsonObject object(JsonObject obj) {
  // do something
  return obj;
 }

@Path("array")
@POST
@Produces("application/json")
@Consumes("application/json")
public JsonArray array(JsonArray array) {
  // do something
  return array;
}
```

```
@Path("structure")
@POST
@Produces("application/json")
@Consumes("application/json")
public JsonStructure structure(JsonStructure structure) {
  // do something
  return structure;
}
```

You can also use Jakarta JSON Processing from a client to send JSON.

Example: Client Using Jakarta JSON Processing

```
WebTarget target = client.target(...);
JsonObject obj = Json.createObjectBuilder().add("name", "Bill").build();
JsonObject newObj = target.request().post(Entity.json(obj), JsonObject.class);
```

2.9. RESTEASY/JAKARTA ENTERPRISE BEANS INTEGRATION

To integrate RESTEasy with Jakarta Enterprise Beans, add Jakarta RESTful Web Services annotations to the Jakarta Enterprise Beans classes that you want to expose as Jakarta RESTful Web Services endpoints. You can also apply the annotations on the bean's business interface. There are two ways to activate the beans as endpoints:

- Using the **web.xml** file.
- Using **javax.ws.rs.core.Application**.

To make an Jakarta Enterprise Beans function as a Jakarta RESTful Web Services resource, annotate a stateless session bean's **@Remote** or **@Local** interface with Jakarta RESTful Web Services annotations:

```
@Local
@Path("/Library")
public interface Library {
  @GET
  @Path("/books/{isbn}")
  public String getBook(@PathParam("isbn") String isbn);
}
@Stateless
public class LibraryBean implements Library {
...
}
```

NOTE

Note that the **Library** interface is referenced by fully qualified name, whereas **LibraryBean** is referenced only by the simple class name.

Then, manually register the Jakarta Enterprise Beans with RESTEasy using the **resteasy.jndi.resources** context parameter in the RESTEasy **web.xml** file:

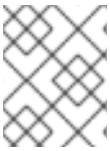
```
<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <context-param>
    <param-name>resteasy.jndi.resources</param-name>
    <param-value>java:module/LibraryBean!org.app.Library</param-value>
  </context-param>
  <listener>
    <listener-class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-class>
  </listener>
  <servlet>
    <servlet-name>Resteasy</servlet-name>
    <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Resteasy</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

You can also specify multiple Java Naming and Directory Interface names of Jakarta Enterprise Beans, separated by commas, for the **resteasy.jndi.resources** context parameter.

An alternate Jakarta EE-standard way to activate Jakarta Enterprise Beans as RESTEasy endpoints is by using **javax.ws.rs.core.Application**. This is achieved by including the Jakarta Enterprise Beans implementation class into the set returned by the application's **getClasses()** method. This approach does not need anything to be specified in the **web.xml** file.

See the **kitchensink**, **helloworld-html5**, and **managed-executor-service** quickstarts that ship with JBoss EAP for working examples that demonstrate RESTEasy integration with Jakarta Enterprise Beanss.

2.10. SPRING INTEGRATION



NOTE

Your application must have an existing Jakarta XML Web Services service and client configuration.

RESTEasy integrates with Spring 4.2.x.

Maven users must use the **resteasy-spring** artifact. Alternatively, the JAR is available as a module in JBoss EAP.

RESTEasy comes with its own Spring **ContextLoaderListener** that registers a RESTEasy specific **BeanPostProcessor** that processes Jakarta RESTful Web Services annotations when a bean is created by a **BeanFactory**. This means that RESTEasy automatically scans for **@Provider** and Jakarta RESTful Web Services resource annotations on your bean class and registers them as Jakarta RESTful Web Services resources.

Add the following to your **web.xml** file to enable the RESTEasy/Spring integration functionality:

```
<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <listener>
```

```
    <listener-class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-class>
  </listener>
  <listener>
    <listener-class>org.jboss.resteasy.plugins.spring.SpringContextLoaderListener</listener-class>
  </listener>
  <listener>
    <listener-class>org.springframework.web.context.request.RequestContextListener</listener-
class>
  </listener>
  <servlet>
    <servlet-name>Resteasy</servlet-name>
    <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Resteasy</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

The **SpringContextLoaderListener** must be declared after **ResteasyBootstrap** as it uses **ServletContext** attributes initialized by it.

See the **spring-resteasy** quickstart that ships with JBoss EAP for a working example of a web application that demonstrates RESTEasy integration with Spring.

2.11. JAKARTA CONTEXTS AND DEPENDENCY INJECTION INTEGRATION

Integration between RESTEasy and Jakarta Contexts and Dependency Injection is provided by the **resteasy-cdi** module.

Both the Jakarta RESTful Web Services and Jakarta Contexts and Dependency Injection specifications introduce their own component models. Every class placed in a Jakarta Contexts and Dependency Injection archive, which fulfills a set of basic constraints, is implicitly a Jakarta Contexts and Dependency Injection bean. Explicit declaration of your Java class with **@Path** or **@Provider** is required for it to become a Jakarta RESTful Web Services component. Without the integration code, annotating a class suitable for being a Jakarta Contexts and Dependency Injection bean with Jakarta RESTful Web Services annotations gives a faulty result and the Jakarta RESTful Web Services component is not managed by the Jakarta Contexts and Dependency Injection. The **resteasy-cdi** module is a bridge that allows RESTEasy to work with class instances obtained from the Jakarta Contexts and Dependency Injection container.

During a web service invocation, the **resteasy-cdi** module asks the Jakarta Contexts and Dependency Injection container for the managed instance of a Jakarta RESTful Web Services component. Then, this instance is passed to RESTEasy. If a managed instance is not available for some reason, such as the class being placed in a JAR file that is not a bean deployment archive, RESTEasy falls back to instantiating the class itself.

As a result, Jakarta Contexts and Dependency Injection services like injection, lifecycle management, events, decoration, and interceptor bindings can be used in Jakarta RESTful Web Services components.

2.11.1. Default Scope

A Jakarta Contexts and Dependency Injection bean that does not explicitly define a scope is **@Dependent** scoped by default. This pseudo-scope means that the bean adapts to the lifecycle of the

bean that it is injected into. Normal scopes, including request, session, and application, are more suitable for Jakarta RESTful Web Services components as they designate the component's lifecycle boundaries explicitly. Therefore, the **resteasy-cdi** module alters the default scoping in the following way:

- If a Jakarta RESTful Web Services root resource does not define a scope explicitly, it is bound to the request scope.
- If a Jakarta RESTful Web Services provider or **javax.ws.rs.Application** subclass does not define a scope explicitly, it is bound to the application scope.

WARNING

Since the scope of all beans that do not declare a scope is modified by the **resteasy-cdi** module, this affects session beans as well. As a result, a conflict occurs if the scope of a stateless session bean or singleton is changed automatically as the specification prohibits these components to be **@RequestScoped**. Therefore, you need to explicitly define a scope when using stateless session beans or singletons. This requirement is likely to be removed in future releases.

The **resteasy-cdi** module is bundled with JBoss EAP. Therefore, there is no need to download the module separately or add any additional configuration. See the **kitchensink** quickstart that ships with JBoss EAP for a working example of using Jakarta Contexts and Dependency Injection beans with a Jakarta RESTful Web Services resource.

2.12. RESTEASY FILTERS AND INTERCEPTORS

Jakarta RESTful Web Services has two different concepts for interceptions: filters and interceptors. Filters are mainly used to modify or process incoming and outgoing request headers or response headers. They execute before and after request and response processing.

2.12.1. Server-side Filters

On the server side, you have two different types of filters: **ContainerRequestFilters** and **ContainerResponseFilters**. **ContainerRequestFilters** run before your Jakarta RESTful Web Services resource method is invoked. **ContainerResponseFilters** run after your Jakarta RESTful Web Services resource method is invoked.

In addition, there are two types of **ContainerRequestFilters**: pre-matching and post-matching. Pre-matching **ContainerRequestFilters** are designated with the **@PreMatching** annotation and execute before the Jakarta RESTful Web Services resource method is matched with the incoming HTTP request. Post-matching **ContainerRequestFilters** are designated with the **@PostMatching** annotation and execute after the Jakarta RESTful Web Services resource method is matched with the incoming HTTP request.

Pre-matching filters often are used to modify request attributes to change how it matches to a specific resource method, for example to strip **.xml** and add an **Accept** header. **ContainerRequestFilters** can abort the request by calling **ContainerRequestContext.abortWith(Response)**. For example, a filter might want to abort if it implements a custom authentication protocol.

After the resource class method is executed, Jakarta RESTful Web Services runs all **ContainerResponseFilters**. These filters allow you to modify the outgoing response before it is marshalled and sent to the client.

Example: Request Filter

```
public class RoleBasedSecurityFilter implements ContainerRequestFilter {
  protected String[] rolesAllowed;
  protected boolean denyAll;
  protected boolean permitAll;

  public RoleBasedSecurityFilter(String[] rolesAllowed, boolean denyAll, boolean permitAll) {
    this.rolesAllowed = rolesAllowed;
    this.denyAll = denyAll;
    this.permitAll = permitAll;
  }

  @Override
  public void filter(ContainerRequestContext requestContext) throws IOException  {
    if (denyAll) {
      requestContext.abortWith(Response.status(403).entity("Access forbidden: role not
allowed").build());
      return;
    }
    if (permitAll) return;
    if (rolesAllowed != null) {
      SecurityContext context = ResteasyProviderFactory.getContextData(SecurityContext.class);
      if (context != null) {
        for (String role : rolesAllowed) {
          if (context.isUserInRole(role)) return;
        }
        requestContext.abortWith(Response.status(403).entity("Access forbidden: role not
allowed").build());
        return;
      }
    }
    return;
  }
}
```

Example: Response Filter

```
public class CacheControlFilter implements ContainerResponseFilter {
  private int maxAge;

  public CacheControlFilter(int maxAge) {
    this.maxAge = maxAge;
  }

  public void filter(ContainerRequestContext req, ContainerResponseContext res)
        throws IOException {
    if (req.getMethod().equals("GET")) {
      CacheControl cc = new CacheControl();
      cc.setMaxAge(this.maxAge);
      res.getHeaders().add("Cache-Control", cc);
```

```
    }
  }
}
```

2.12.2. Client-side Filters

More information on client-side filters can be found in the Jakarta RESTful Web Services Client API section of this guide.

2.12.3. RESTEasy Interceptors

2.12.3.1. Intercept Jakarta RESTful Web Services Invocations

RESTEasy can intercept Jakarta RESTful Web Services invocations and route them through listener-like objects called interceptors.

While filters modify request or response headers, interceptors deal with message bodies. Interceptors are executed in the same call stack as their corresponding reader or writer. **ReaderInterceptors** wrap around the execution of **MessageBodyReaders**. **WriterInterceptors** wrap around the execution of **MessageBodyWriters**. They can be used to implement a specific content-encoding. They can be used to generate digital signatures or to post or pre-process a Java object model before or after it is marshalled.

ReaderInterceptors and **WriterInterceptors** can be used on either the server or client side. They are annotated with **@Provider**, as well as either **@ServerInterceptor** or **@ClientInterceptor** so that RESTEasy knows whether or not to add them to the interceptor list.

These interceptors wrap around the invocation of **MessageBodyReader.readFrom()** or **MessageBodyWriter.writeTo()**. They can be used to wrap the **Output** or **Input** streams.

Example: Interceptor

```
@Provider
public class BookReaderInterceptor implements ReaderInterceptor {
   @Inject private Logger log;
   @Override
   @ReaderInterceptorBinding
   public Object aroundReadFrom(ReaderInterceptorContext context) throws IOException,
WebApplicationException {
      log.info("*** Intercepting call in BookReaderInterceptor.aroundReadFrom()");
      VisitList.add(this);
      Object result = context.proceed();
      log.info("*** Back from intercepting call in BookReaderInterceptor.aroundReadFrom()"); return
result;
   }
}
```

The interceptors and the **MessageBodyReader** or **Writer** are invoked in one big Java call stack. **ReaderInterceptorContext.proceed()** or **WriterInterceptorContext.proceed()** are called in order to go to the next interceptor or, if there are no more interceptors to invoke, the **readFrom()** or **writeTo()** method of the **MessageBodyReader** or **MessageBodyWriter**. This wrapping allows objects to be modified before they get to the **Reader** or **Writer**, and then cleaned up after **proceed()** returns.

The example below is a server-side interceptor that adds a header value to the response.

```
@Provider
public class BookWriterInterceptor implements WriterInterceptor {
  @Inject private Logger log;

  @Override
  @WriterInterceptorBinding
  public void aroundWriteTo(WriterInterceptorContext context) throws IOException,
WebApplicationException {
    log.info("*** Intercepting call in BookWriterInterceptor.aroundWriteTo()");
    VisitList.add(this);
    context.proceed();
    log.info("*** Back from intercepting call in BookWriterInterceptor.aroundWriteTo()");
  }
}
```

2.12.3.2. Registering an Interceptor

To register a RESTEasy Jakarta RESTful Web Services interceptor in an application, list it in the **web.xml** file under the **resteasy.providers** parameter in the **context-param** element, or return it as a class or as an object in the **Application.getClasses()** or **Application.getSingletons()** method.

```
<context-param>
   <param-name>resteasy.providers</param-name>
   <param-value>my.app.CustomInterceptor</paramvalue>
</context-param>
```

```
package org.jboss.resteasy.example;

import javax.ws.rs.core.Application;
import java.util.HashSet;
import java.util.Set;

public class MyApp extends Application {

  public java.util.Set<java.lang.Class<?>> getClasses() {
    Set<Class<?>> resources = new HashSet<Class<?>>();
    resources.add(MyResource.class);
    resources.add(MyProvider.class);
    return resources;
  }
}
```

```
package org.jboss.resteasy.example;

import javax.ws.rs.core.Application;
import java.util.HashSet;
import java.util.Set;

public class MyApp extends Application {

   protected Set<Object> singletons = new HashSet<Object>();

   public MyApp() {
      singletons.add(new MyResource());
```

```
        singletons.add(new MyProvider());
    }

    @Override
    public Set<Object> getSingletons() {
        return singletons;
    }
}
```

2.12.4. GZIP Compression and Decompression

RESTEasy supports GZIP compression and decompression. To support GZIP decompression, the client framework or a Jakarta RESTful Web Services service automatically decompresses a message body with a **Content-Encoding** of **gzip**, and it can automatically set the **Accept-Encoding** header to **gzip, deflate** so that you do not have to set this header manually. To support GZIP compression, RESTEasy compresses the outgoing message if the client framework is sending a request or if the server is sending a response with the **Content-Encoding** header set to **gzip**. You can use the **@org.jboss.resteasy.annotation.GZIP** annotation to set the **Content-Encoding** header.

The following example tags the outgoing message body **order** to be gzip compressed.

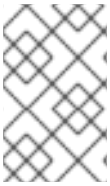
Example: GZIP Compression

```
@Path("/")
public interface MyProxy {

   @Consumes("application/xml")
   @PUT
   public void put(@GZIP Order order);
}
```

Example: GZIP Compression Tagging Server Responses

```
@Path("/")
public class MyService {

   @GET
   @Produces("application/xml")
   @GZIP
   public String getData() {...}
}
```

2.12.4.1. Configuring GZIP Compression and Decompression



NOTE

RESTEasy disables GZIP compression and decompression by default in order to prevent decompression of an entity that might be huge in size but has been compressed by an attacker and sent to the server.

There are three interceptors that are relevant to GZIP compression and decompression:

- **org.jboss.resteasy.plugins.interceptors.GZIPDecodingInterceptor**: If the **Content-Encoding** header is present and has the value **gzip**, **GZIPDecodingInterceptor** installs an **InputStream** that decompresses the message body.
- **org.jboss.resteasy.plugins.interceptors.GZIPEncodingInterceptor**: If the **Content-Encoding** header is present and has the value **gzip**, **GZIPEncodingInterceptor** installs an **OutputStream** that compresses the message body.
- **org.jboss.resteasy.plugins.interceptors.AcceptEncodingGZIPFilter**: If the **Accept-Encoding** header does not exist, **AcceptEncodingGZIPFilter** adds **Accept-Encoding** header with the value **gzip, deflate**. If the **Accept-Encoding** header exists but does not contain **gzip**, **AcceptEncodingGZIPFilter** interceptor appends the value **, gzip**.

NOTE

Enabling GZIP compression or decompression does not depend on the presence of the **AcceptEncodingGZIPFilter** interceptor.

Enabling GZIP decompression sets an upper limit on the number of bytes the **GZIPDecodingInterceptor** can extract from the compressed message body. The default limit is **10,000,000**.

2.12.4.2. Server-side GZIP Configuration

You can enable the interceptors by including their class names in the **javax.ws.rs.ext.Providers** file on the class path. The upper limit on deflated files is set using the web application context parameter **resteasy.gzip.max.input**. If this limit is exceeded on the server side, **GZIPDecodingInterceptor** returns a response with status **413 - Request Entity Too Large** and a message specifying the upper limit.

2.12.4.2.1. Client-side GZIP Configuration

You can enable the GZIP interceptors by registering them with, for example, a **Client** or **WebTarget**. For example:

```
Client client = new ResteasyClientBuilder() // Activate gzip compression on client:
    .register(AcceptEncodingGZIPFilter.class)
    .register(GZIPDecodingInterceptor.class)
    .register(GZIPEncodingInterceptor.class)
    .build();
```

You can configure the upper limit on deflated files by creating an instance of **GZIPDecodingInterceptor** with a specific value:

```
Client client = new ResteasyClientBuilder() // Activate gzip compression on client:
    .register(AcceptEncodingGZIPFilter.class)
    .register(new GZIPDecodingInterceptor(256))
    .register(GZIPEncodingInterceptor.class)
    .build();
```

If the upper limit is exceeded on the client side, **GZIPDecodingInterceptor** throws a **ProcessingException** with a message specifying the upper limit.

2.12.5. Per-Resource Method Filters and Interceptors

Sometimes you want a filter or interceptor to only run for a specific resource method. You can do this in two different ways:

- Implement the **DynamicFeature** interface.
- Use the **@NameBinding** annotation.

Implement the DynamicFeature Interface

The **DynamicFeature** interface includes a callback method, **configure(ResourceInfo resourceInfo, FeatureContext context)**, which is invoked for each and every deployed Jakarta RESTful Web Services method. The **ResourceInfo** parameter contains information about the current Jakarta RESTful Web Services method being deployed. **FeatureContext** is an extension of the **Configurable** interface. You can use the **register()** method of this parameter to bind the filters and interceptors that you want to assign to this method.

Example: Using the DynamicFeature Interface

```
@Provider
public class AnimalTypeFeature implements DynamicFeature {
   @Override
   public void configure(ResourceInfo info, FeatureContext context) {
      if (info.getResourceMethod().getAnnotation(GET.class) != null)
         AnimalFilter filter = new AnimalFilter();
         context.register(filter);
      }
   }
}
```

In the example above, the provider that you register using **AnimalTypeFeature** must implement one of the interfaces. This example registers the provider **AnimalFilter** that must implement one of the following interfaces: **ContainerRequestFilter**, **ContainerResponseFilter**, **ReaderInterceptor**, **WriterInterceptor**, or **Feature**. In this case **AnimalFilter** will be applied to all resource methods annotated with GET annotation. See DynamicFeature Documentation for details.

Use the @NameBinding Annotation

@NameBinding works a lot like Jakarta Contexts and Dependency Injection interceptors. You annotate a custom annotation with **@NameBinding** and then apply that custom annotation to your filter and resource method.

Example: Using @NameBinding

```
@NameBinding
public @interface DoIt {}

@DoIt
public class MyFilter implements ContainerRequestFilter {...}

@Path("/root")
public class MyResource {

   @GET
   @DoIt
   public String get() {...}
}
```

See NameBinding Documentation for details.

2.12.6. Ordering

Ordering is accomplished by using the **@Priority** annotation on your filter or interceptor class.

2.12.7. Exception Handling with Filters and Interceptors

Exceptions associated with filters or interceptors can occur on either the client side or the server side. On the client side, there are two types of exceptions you will have to handle: **javax.ws.rs.client.ProcessingException** and **javax.ws.rs.client.ResponseProcessingException**. A **javax.ws.rs.client.ProcessingException** will be thrown on the client side if there was an error before a request is sent to the server. A **javax.ws.rs.client.ResponseProcessingException** will be thrown on the client side if there was an error in processing the response received by the client from the server.

On the server side, exceptions thrown by filters or interceptors are handled in the same way as other exceptions thrown from Jakarta RESTful Web Services methods, which tries to find an **ExceptionMapper** for the exception being thrown. More details on how exceptions are handled in Jakarta RESTful Web Services methods can be found in the Exception Handling section.

2.13. LOGGING RESTEASY PROVIDERS AND INTERCEPTORS

RESTEasy logs the used providers and interceptors in the **DEBUG** level of logging. You can use the following management CLI commands to enable all the log levels related to RESTEasy:

```
/subsystem=logging/console-handler=CONSOLE:write-attribute(name=level,value=ALL)

/subsystem=logging/logger=org.jboss.resteasy:add(level=ALL)

/subsystem=logging/logger=javax.xml.bind:add(level=ALL)

/subsystem=logging/logger=com.fasterxml.jackson:add(level=ALL)
```

2.14. EXCEPTION HANDLING

2.14.1. Creating an Exception Mapper

Exception mappers are custom components provided by applications that catch thrown exceptions and write specific HTTP responses.

When you create an exception mapper, you create a class that is annotated with the **@Provider** annotation and implements the **ExceptionMapper** interface.

An example exception mapper is provided below:

```
@Provider
public class EJBExceptionMapper implements ExceptionMapper<javax.ejb.EJBException> {
  public Response toResponse(EJBException exception) {
    return Response.status(500).build();
  }
}
```

To register an exception mapper, list it in the **web.xml** file, under the **resteasy.providers context-param**, or register it programmatically through the **ResteasyProviderFactory** class.

2.14.2. Managing Internally Thrown Exceptions

Table 2.2. Exception List

Exception	HTTP Code	Description
BadRequestException	400	Bad Request. The request was not formatted correctly, or there was a problem processing the request input.
UnauthorizedException	401	Unauthorized. Security exception thrown if you are using RESTEasy's annotation-based role-based security.
InternalServerErrorException	500	Internal Server Error.
MethodNotAllowedException	405	There is no Jakarta RESTful Web Services method for the resource to handle the invoked HTTP operation.
NotAcceptableException	406	There is no Jakarta RESTful Web Services method that can produce the media types listed in the Accept header.
NotFoundException	404	There is no Jakarta RESTful Web Services method that serves the request path/resource.
ReaderException	400	All exceptions thrown from MessageBodyReaders are wrapped within this exception. If there is no ExceptionMapper for the wrapped exception, or if the exception is not a WebApplicationException, then by default, RESTEasy returns a 400 code.

Exception	HTTP Code	Description
WriterException	500	All exceptions thrown from MessageBodyWriters are wrapped within this exception. If there is no ExceptionMapper for the wrapped exception, or if the exception is not a WebApplicationException, then by default, RESTEasy returns a 400 code.
JAXBUnmarshalException	400	The Jakarta XML Binding providers (XML and Jackson) throw this exception on reads which might wrap JAXBExceptions. This class extends ReaderException.
JAXBMarshalException	500	The Jakarta XML Binding providers (XML and Jackson) throw this exception on writes which might wrap JAXBExceptions. This class extends WriterException.
ApplicationException	N/A	Wraps all exceptions thrown from application code, and it functions in the same way as InvocationTargetException. If there is an ExceptionMapper for wrapped exception, then that is used to handle the request.
Failure	N/A	Internal RESTEasy error. Not logged.
LoggableFailure	N/A	Internal RESTEasy error. Logged.
DefaultOptionsMethodException	N/A	If the user invokes **HTTP OPTIONS** and no Jakarta RESTful Web Services method for it, RESTEasy provides a default behavior by throwing this exception.
UnrecognizedPropertyExceptionHandler	400	RESTEasy Jackson provider throws this exception when JSON data is determined to be invalid.

2.15. SECURING JAKARTA RESTFUL WEB SERVICES WEB SERVICES

RESTEasy supports the **@RolesAllowed**, **@PermitAll**, and **@DenyAll** annotations on Jakarta RESTful Web Services methods. However, you must enable role-based security in order for these annotations to be recognized.

2.15.1. Enable Role-Based Security

Follow these steps to configure the **web.xml** file to enable role-based security.

WARNING

Do not activate role-based security if the application uses Jakarta Enterprise Beans. The Jakarta Enterprise Beans container will provide the functionality, instead of RESTEasy.

Enable Role-Based Security for a RESTEasy Jakarta RESTful Web Services Web Service

1. Open the **web.xml** file for the application in a text editor.
2. Add the following **<context-param>** to the file, within the **<web-app>** tags.

```
<context-param>
  <param-name>resteasy.role.based.security</param-name>
  <param-value>true</param-value>
</context-param>
```

3. Declare all roles used within the RESTEasy Jakarta RESTful Web Services WAR file, using the **<security-role>** tags.

```
<security-role>
  <role-name>ROLE_NAME</role-name>
</security-role>
<security-role>
  <role-name>ROLE_NAME</role-name>
</security-role>
```

4. Authorize access to all URLs handled by the Jakarta RESTful Web Services runtime for all roles.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Resteasy</web-resource-name>
    <url-pattern>/PATH</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>ROLE_NAME</role-name>
    <role-name>ROLE_NAME</role-name>
  </auth-constraint>
</security-constraint>
```

5. Define the appropriate login configuration for this application.

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>jaxrs</realm-name>
</login-config>
```

Role-based security has been enabled within the application, with a set of defined roles.

Example: Role-Based Security Configuration

```
<web-app>

  <context-param>
    <param-name>resteasy.role.based.security</param-name>
    <param-value>true</param-value>
  </context-param>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Resteasy</web-resource-name>
      <url-pattern>/security</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>admin</role-name>
      <role-name>user</role-name>
    </auth-constraint>
  </security-constraint>

  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>jaxrs</realm-name>
  </login-config>

  <security-role>
    <role-name>admin</role-name>
  </security-role>
  <security-role>
    <role-name>user</role-name>
  </security-role>

</web-app>
```

2.15.2. Securing Jakarta RESTful Web Services Web Services Using Annotations

To secure Jakarta RESTful Web Services web services using an annotation, complete the following steps.

1. Enable role-based security .

2. Add security annotations to the Jakarta RESTful Web Services web service. RESTEasy supports the following annotations:

   @RolesAllowed

   Defines which roles can access the method. All roles should be defined in the **web.xml** file.

@PermitAll

Allows all roles defined in the **web.xml** file to access the method.

@DenyAll

Denies all access to the method.

Below is an example that uses the **@RolesAllowed** annotation to specify that the **admin** role can access the web service.

```
@RolesAllowed("admin")
@Path("/test")
public class TestService {
  ...
}
```

2.15.3. Setting Programmatic Security

Jakarta RESTful Web Services includes a programmatic API for gathering security information about a secured request. The **javax.ws.rs.core.SecurityContext** interface has a method for determining the identity of the user making the secured HTTP invocation. It also has a method that allows you to check whether or not the current user belongs to a certain role:

```
public interface SecurityContext {

  public Principal getUserPrincipal();
  public boolean isUserInRole(String role);
  public boolean isSecure();
  public String getAuthenticationScheme();
}
```

You can access a **SecurityContext** instance by injecting it into a field, setter method, or resource method parameter using the **@Context** annotation.

```
@Path("test")
public class SecurityContextResource {
   @Context
   SecurityContext securityContext;

   @GET
   @Produces("text/plain")
   public String get() {
      if (!securityContext.isUserInRole("admin")) {
         throw new
WebApplicationException(Response.serverError().status(HttpResponseCodes.SC_UNAUTHORIZED)

                .entity("User " + securityContext.getUserPrincipal().getName() + " is not
authorized").build());
      }
      return "Good user " + securityContext.getUserPrincipal().getName();
   }
}
```

2.16. RESTEASY ASYNCHRONOUS JOB SERVICE

The RESTEasy Asynchronous Job Service is designed to add asynchronous behavior to the HTTP protocol. While HTTP is a synchronous protocol, it is aware of asynchronous invocations. The HTTP 1.1 response code **202 Accepted** means that the server has received and accepted the response for processing, but the processing has not yet been completed. The Asynchronous Job Service builds around this.

2.16.1. Enabling the Asynchronous Job Service

Enable the asynchronous job service in the **web.xml** file:

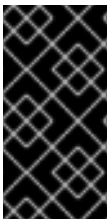
```
<context-param>
   <param-name>resteasy.async.job.service.enabled</param-name>
   <param-value>true</param-value>
</context-param>
```

2.16.2. Configuring Asynchronous Jobs

This section covers examples of the query parameters for asynchronous jobs with RESTEasy.

WARNING

Role based security does not work with the Asynchronous Job Service as it cannot be implemented portably. If the Asynchronous Job Service is used, application security must be done through XML declarations in the **web.xml** file instead.



IMPORTANT

While GET, DELETE, and PUT methods can be invoked asynchronously, this breaks the HTTP 1.1 contract of these methods. While these invocations might not change the state of the resource if invoked more than once, they do change the state of the server as new job entries with each invocation.

The **asynch** query parameter is used to run invocations in the background. A **202 Accepted** response is returned, as well as a location header with a URL pointing to where the response of the background method is located.

```
POST http://example.com/myservice?asynch=true
```

The example above returns a **202 Accepted** response. It also returns a location header with a URL pointing to where the response of the background method is located. An example of the location header is shown below:

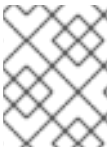
```
HTTP/1.1 202 Accepted
Location: http://example.com/asynch/jobs/3332334
```

The URI will take the form of:

```
/asynch/jobs/{job-id}?wait={milliseconds}|nowait=true
```

GET, POST and DELETE operations can be performed on this URL.

- GET returns the Jakarta RESTful Web Services resource method invoked as a response if the job was completed. If the job has not been completed, the GET returns a **202 Accepted** response code. Invoking GET does not remove the job, so it can be called multiple times.
- POST does a read of the job response and removes the job if it has been completed.
- DELETE is called to manually clean up the job queue.



NOTE

When the job queue is full, it evicts the earliest job from memory automatically without needing to call DELETE.

The GET and POST operations allow for the maximum wait time to be defined, using the **wait** and **nowait** query parameters. If the **wait** parameter is not specified, the operation will default to **nowait=true**, and will not wait at all if the job is not complete. The **wait** parameter is defined in milliseconds.

```
POST http://example.com/asynch/jobs/122?wait=3000
```

RESTEasy supports fire and forget jobs, using the **oneway** query parameter.

```
POST http://example.com/myservice?oneway=true
```

The example above returns a **202 Accepted** response, but no job is created.

NOTE

The configuration parameters for the Asynchronous Job Service can be found in the RESTEasy Asynchronous Job Service Configuration Parameters section in the appendix.

2.17. RESTEASY JAVASCRIPT API

2.17.1. About the RESTEasy JavaScript API

RESTEasy can generate a JavaScript API that uses AJAX calls to invoke Jakarta RESTful Web Services operations. Each Jakarta RESTful Web Services resource class will generate a JavaScript object of the same name as the declaring class or interface. The JavaScript object contains each Jakarta RESTful Web Services method as properties.

```
@Path("foo")
public class Foo {

  @Path("{id}")
  @GET
  public String get(@QueryParam("order") String order, @HeaderParam("X-Foo") String header,
    @MatrixParam("colour") String colour, @CookieParam("Foo-Cookie") String cookie) {
  }

  @POST
```

```
  public void post(String text) {
  }
}
```

The following JavaScript code uses the Jakarta RESTful Web Services API that was generated in the previous example.

```
var text = Foo.get({order: 'desc', 'X-Foo': 'hello', colour: 'blue', 'Foo-Cookie': 123987235444});
Foo.post({$entity: text});
```

Each JavaScript API method takes an optional object as single parameter where each property is a cookie, header, path, query or form parameter as identified by its name, or the API parameter properties. For details about the API parameter properties see RESTEasy Javascript API Parameters appendix.

2.17.1.1. Enable the RESTEasy JavaScript API Servlet

The RESTEasy JavaScript API is disabled by default. Follow these steps to enable it by updating the **web.xml** file.

1. Open the **web.xml** file of the application in a text editor.
2. Add the following configuration to the file, inside the **web-app** tags:

```
<servlet>
   <servlet-name>RESTEasy JSAPI</servlet-name>
   <servlet-class>org.jboss.resteasy.jsapi.JSAPIServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>RESTEasy JSAPI</servlet-name>
   <url-pattern>/</url-pattern>
</servlet-mapping>
```

2.17.1.2. Build AJAX Queries

The RESTEasy JavaScript API can be used to manually construct requests. The following are some examples of this behavior.

Example: REST Object Used to Override RESTEasy JavaScript API Client Behavior

```
// Change the base URL used by the API:
REST.apiURL = "http://api.service.com";

// log everything in a div element
REST.log = function(text) {
  jQuery("#log-div").append(text);
};
```

The **REST** object contains the following read-write properties:

- **apiURL**: Set by default to the Jakarta RESTful Web Services root URL. Used by every JavaScript client API functions when constructing the requests.

- **log**: Set to **function(string)** in order to receive RESTEasy client API logs. This is useful if you want to debug your client API and place the logs where you can see them.

Example: Class Using REST.Request() Method to Build Custom Requests

```
var r = new REST.Request();
r.setURI("http://api.service.com/orders/23/json");
r.setMethod("PUT");
r.setContentType("application/json");
r.setEntity({id: "23"});
r.addMatrixParameter("JSESSIONID", "12309812378123");
r.execute(function(status, request, entity) {
  log("Response is " + status);
});
```

2.18. RESTEASY SPI TO MODIFY RESOURCE METADATA

JBoss EAP provides a RESTEasy service provider interface (SPI) to modify resource class metadata, which is created using **ResourceBuilder**. When processing Jakarta RESTful Web Services deployments, RESTEasy uses **ResourceBuilder** to create metadata for each Jakarta RESTful Web Services resource. Such metadata is defined using the metadata SPI in package **org.jboss.resteasy.spi.metadata**, in particular the **ResourceClass** interface:

```
package org.jboss.resteasy.spi.metadata;

public interface ResourceClass
{
  String getPath();

  Class<?> getClazz();

  ResourceConstructor getConstructor();

  FieldParameter[] getFields();

  SetterParameter[] getSetters();

  ResourceMethod[] getResourceMethods();

  ResourceLocator[] getResourceLocators();
}
```

RESTEasy allows customizing the metadata generation by providing implementations of the **ResourceClassProcessor** interface. The following example illustrates the usage of this SPI:

```
package org.jboss.resteasy.test.core.spi.resource;

import org.jboss.logging.Logger;
import org.jboss.resteasy.spi.metadata.ResourceClass;
import org.jboss.resteasy.spi.metadata.ResourceClassProcessor;

import javax.ws.rs.ext.Provider;

@Provider
```

```
public class ResourceClassProcessorImplementation implements ResourceClassProcessor {

    protected static final Logger logger =
Logger.getLogger(ResourceClassProcessorImplementation.class.getName());
@Override
    public ResourceClass process(ResourceClass clazz) {
        logger.info(String.format("ResourceClassProcessorImplementation#process method called on
class %s",
                        clazz.getClazz().getSimpleName()));
        String clazzName = clazz.getClazz().getSimpleName();
        if (clazzName.startsWith("ResourceClassProcessorEndPoint")
                || clazzName.equals("ResourceClassProcessorProxy")
                || clazzName.equals("ResourceClassProcessorProxyEndPoint")) {
            return new ResourceClassProcessorClass(clazz);
        }
        return clazz;
    }
```

The new processors, which are stored using the **ResteasyProviderFactory** class, are resolved as regular Jakarta RESTful Web Services annotated providers. They allow wrapping resource metadata classes with custom versions that you can use for various advanced scenarios, such as:

- Adding additional resource method or locators to the resource.
- Modifying the HTTP methods.
- Modifying the **@Produces** or the **@Consumes** media types.

2.19. MICROPROFILE REST CLIENT

IMPORTANT

MicroProfile REST client is provided as Technology Preview only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs), might not be functionally complete, and Red Hat does not recommend to use them for production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

See Technology Preview Features Support Scope on the Red Hat Customer Portal for information about the support scope for Technology Preview features.

JBoss EAP 7.4 supports the MicroProfile REST client 1.4.x that builds on Jakarta RESTful Web Services 2.1 client APIs to provide a type-safe approach to invoke RESTful services over HTTP. The MicroProfile Type Safe REST clients are defined as Java interfaces. With the MicroProfile REST clients, you can write client applications with executable code.

The MicroProfile REST client enables:

- An intuitive syntax
- Programmatic registration of providers
- Declarative registration of providers
- Declarative specification of headers

- Propagation of headers on the server
- **ResponseExceptionMapper**
- Jakarta Contexts and Dependency Injection integration

2.19.1. Intuitive Syntax

The MicroProfile REST client enables a version of distributed object communication, which is also implemented in CORBA, Java Remote Method Invocation (RMI), the JBoss Remoting Project, and RESTEasy. For example, consider the resource:

```
@Path("resource")
public class TestResource {
  @Path("test")
  @GET
  String test() {
    return "test";
  }
 }
```

The Jakarta RESTful Web Services native way of accessing the **TestResource** class is:

```
Client client = ClientBuilder.newClient();
String response = client.target("http://localhost:8081/test").request().get(String.class);
```

However, Microprofile REST client supports a more intuitive syntax by directly calling the **test()** method:

```
@Path("resource")
public interface TestResourceIntf {
   @Path("test")
   @GET
   public String test();
}

TestResourceIntf service = RestClientBuilder.newBuilder()
                           .baseUrl(http://localhost:8081/))
                           .build(TestResourceIntf.class);
String s = service.test();
```

In the example above, making calls on the **TestResource** class becomes much easier with the **TestResourceIntf** class, as illustrated by the call **service.test()**.

The following example is a more elaborate version of the **TestResourceIntf** class:

```
@Path("resource")
public interface TestResourceIntf2 {
  @Path("test/{path}")mes("text/plain")
  @Produces("text/html")
  @POST
  public String test(@PathParam("path") String path, @QueryParam("query") String query, String
entity);
}
```

Calling the **service.test("p", "q", "e")** method results in an HTTP message that looks like:

```
POST /resource/test/p/?query=q HTTP/1.1
Accept: text/html
Content-Type: text/plain
Content-Length: 1

e
```

2.19.2. Programmatic Registration of Providers

With the MicroProfile REST client, you can also configure the client environment by registering providers. For example:

```
TestResourceIntf service = RestClientBuilder.newBuilder()
                           .baseUrl(http://localhost:8081/))
                           .register(MyClientResponseFilter.class)
                           .register(MyMessageBodyReader.class)
                           .build(TestResourceIntf.class);
```

2.19.3. Declarative Registration of Providers

You can also register providers declaratively by adding the **org.eclipse.microprofile.rest.client.annotation.RegisterProvider** annotation to the target interface as shown below:

```
@Path("resource")
@RegisterProvider(MyClientResponseFilter.class)
@RegisterProvider(MyMessageBodyReader.class)
public interface TestResourceIntf2 {
  @Path("test/{path}")
  @Consumes("text/plain")
  @Produces("text/html")
  @POST
  public String test(@PathParam("path") String path, @QueryParam("query") String query, String
entity);
}
```

Declaring the **MyClientResponseFilter** class and the **MyMessageBodyReader** class with annotations eliminates the need to call the **RestClientBuilder.register()** method.

2.19.4. Declarative Specification of Headers

You can specify a header for an HTTP request in the following ways:

- By annotating one of the resource method parameters.
- By declaratively using the **org.eclipse.microprofile.rest.client.annotation.ClientHeaderParam** annotation.

The following example illustrates setting a header by annotating one of the resource method parameters with the annotation **@HeaderValue**:

```
@POST
@Produces(MediaType.TEXT_PLAIN)
@Consumes(MediaType.TEXT_PLAIN)
String contentLang(@HeaderParam(HttpHeaders.CONTENT_LANGUAGE) String contentLanguage,
String subject);
```

The following example illustrates setting a header using the **org.eclipse.microprofile.rest.client.annotation.ClientHeaderParam** annotation:

```
@POST
@Produces(MediaType.TEXT_PLAIN)
@Consumes(MediaType.TEXT_PLAIN)
@ClientHeaderParam(name=HttpHeaders.CONTENT_LANGUAGE, value="{getLanguage}")
String contentLang(String subject);

default String getLanguage() {
  return ...;
}
```

2.19.5. Propagation of Headers on the Server

An instance of **org.eclipse.microprofile.rest.client.ext.ClientHeadersFactory**, if activated, can do a bulk transfer of incoming headers to an outgoing request. The default instance **org.eclipse.microprofile.rest.client.ext.DefaultClientHeadersFactoryImpl** returns a map consisting of those incoming headers that are listed in the comma-separated configuration property **org.eclipse.microprofile.rest.client.propagateHeaders**. The following are the rules for instantiating the **ClientHeadersFactory** interface:

- A **ClientHeadersFactory** instance invoked in the context of a Jakarta RESTful Web Services request must support injection of fields and methods annotated with **@Context**.
- A **ClientHeadersFactory** instance that is managed by Jakarta Contexts and Dependency Injection must use the appropriate Jakarta Contexts and Dependency Injection-managed instance. It must also support the **@Inject** injection.

The **org.eclipse.microprofile.rest.client.ext.ClientHeadersFactory** interface is defined as follows:

```
public interface ClientHeadersFactory {

/**
 * Updates the HTTP headers to send to the remote service. Note that providers
 * on the outbound processing chain could further update the headers.
 *
 * @param incomingHeaders - the map of headers from the inbound Jakarta RESTful Web Services
request. This will
 * be an empty map if the associated client interface is not part of a Jakarta RESTful Web Services
request.
 * @param clientOutgoingHeaders - the read-only map of header parameters specified on the
 * client interface.
 * @return a map of HTTP headers to merge with the clientOutgoingHeaders to be sent to
 * the remote service.
 */
MultivaluedMap<String, String> update(MultivaluedMap<String, String> incomingHeaders,
                                      MultivaluedMap<String, String> clientOutgoingHeaders);
}
```

For more information about the **ClientHeadersFactory** interface, see ClientHeadersFactory Javadoc.

2.19.6. ResponseExceptionMapper

The **org.eclipse.microprofile.rest.client.ext.ResponseExceptionMapper** class is the client-side inverse of the **javax.ws.rs.ext.ExceptionMapper** class defined in Jakarta RESTful Web Services. That is, where the **ExceptionMapper.toResponse()** method turns an **Exception** class thrown during the server-side processing into a **Response** class, the **ResponseExceptionMapper.toThrowable()** method turns a Response class received on the client-side with an HTTP error status into an **Exception** class.

You can register the **ResponseExceptionMapper** class either programmatically or declaratively. In the absence of a registered **ResponseExceptionMapper** class, a default **ResponseExceptionMapper** class maps any response with status **>= 400** to a **WebApplicationException** class.

2.19.7. Jakarta Contexts and Dependency Injection Integration

In MicroProfile REST client, you must annotate any interface that is managed as a Jakarta Contexts and Dependency Injection bean with the **@RegisterRestClient** class. For example:

```
@Path("resource")
@RegisterProvider(MyClientResponseFilter.class)
public static class TestResourceImpl {
    @Inject TestDataBase db;

    @Path("test/{path}")
    @Consumes("text/plain")
    @Produces("text/html")
    @POST
    public String test(@PathParam("path") String path, @QueryParam("query")
    String query, String entity) {
      return db.getByName(query);
    }
  }
  @Path("database")
  @RegisterRestClient
  public interface TestDataBase {

    @Path("")
    @POST
    public String getByName(String name);
  }
```

Here, the MicroProfile REST client implementation creates a client for a **TestDataBase** class service, allowing easy access by the **TestResourceImpl** class. However, it does not include the information about the path to the **TestDataBase** class implementation. This information can be supplied by the optional **@Reg- isterProvider** parameter **baseUri**:

```
@Path("database")
@RegisterRestClient(baseUri="https://localhost:8080/webapp")
public interface TestDataBase {
  @Path("")
  @POST
  public String getByName(String name);
}
```

This indicates that you can access the implementation of **TestDataBase** at https://localhost:8080/webapp. You can also supply the information externally with the following system variable:

```
<fully qualified name of TestDataBase>/mp-rest/url=<URL>
```

For example, the following indicates that you can access an implementation of the **com.bluemonkeydiamond.TestDatabase** class at https://localhost:8080/webapp:

```
com.bluemonkeydiamond.TestDatabase/mp-rest/url=https://localhost:8080/webapp
```

2.20. SUPPORT FOR THE COMPLETIONSTAGE TYPE

The Jakarta RESTful Web Services 2.1 specification supports declaring asynchronous resource methods by returning a **CompletionStage** instead of using the **@Suspended** annotation.

Whenever a resource method returns a **CompletionStage** that it subscribed to, the request is suspended. The request is resumed only when the **CompletionStage** type is:

- Resolved to a value, which is then treated as the return value for the method.
- Treated as an error case, and the exception is processed as if it were thrown by the resource method.

The following is an example of asynchronous processing using **CompletionStage**:

```
public class SimpleResource
{

  @GET
  @Path("basic")
  @Produces("text/plain")
  public CompletionStage<Response> getBasic() throws Exception
  {
    final CompletableFuture<Response> response = new CompletableFuture<>();
    Thread t = new Thread()
    {
      @Override
      public void run()
      {
        try
        {
          Response jaxrs = Response.ok("basic").type(MediaType.TEXT_PLAIN).build();
          response.complete(jaxrs);
        }
        catch (Exception e)
        {
          response.completeExceptionally(e);
        }
      }
    };
    t.start();
```

```
        return response;
    }
}
```

2.21. EXTENDING RESTEASY SUPPORT FOR ASYNCHRONOUS REQUEST PROCESSING AND REACTIVE RETURN TYPES

IMPORTANT

Extending RESTEasy support is provided as Technology Preview only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs), might not be functionally complete, and Red Hat does not recommend to use them for production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

See Technology Preview Features Support Scope on the Red Hat Customer Portal for information about the support scope for Technology Preview features.

2.21.1. Pluggable Reactive Types

Jakarta RESTful Web Services 2.1 is extensible to support various reactive libraries. RESTEasy's optional module **resteasy-rxjava2** supports the following reactive types:

- **io.reactivex.Single**: Similar to **CompletionStage**, because it holds one potential value at the most.
- **io.reactivex.Flowable**: Implements **io.reactivex.Publisher**.
- **io.reactivex.Observable**: Similar to **Flowable**, except that it does not support backpressure, the ability of a subscriber to control the load it receives from a producer by calling **Subscription.request()**.

If you import **resteasy-rxjava2**, you can return these reactive types from your resource methods on the server side and receive them on the client side.

The **resteasy-rxjava2** module supports the following three classes to access Singles, Observables, and Flowables respectively on the client side:

- **org.jboss.resteasy.rxjava2.SingleRxInvoker**
- **org.jboss.resteasy.rxjava2.FlowableRxInvoker**
- **org.jboss.resteasy.rxjava2.ObservableRxInvoker**

2.21.2. Extensions for Additional Reactive Classes

RESTEasy implements a framework that supports extensions for additional reactive classes. On the server side, when a resource method returns a **CompletionStage** type, RESTEasy subscribes to it using the **org.jboss.resteasy.core.AsyncResponseConsumer.CompletionStageResponseConsumer** class. When the **CompletionStage** completes, it calls **CompletionStageResponseConsumer.accept()**, which sends the result back to the client.

Support for **CompletionStage** is built in to RESTEasy. We can extend that support to a class like **Single**

by providing a mechanism for transforming a **Single** into a **CompletionStage**. In the **resteasy-rxjava2** module, **org.jboss.resteasy.rxjava2.SingleProvider**, which implements the **org.jboss.resteasy.spi.AsyncResponseProvider<Single<?>>** interface provides this mechanism:

```
public interface AsyncResponseProvider<T> {
  public CompletionStage toCompletionStage(T asyncResponse);
}
```

Given the **SingleProvider** class, RESTEasy can take a **Single**, transform it into a **CompletionStage** and then use **CompletionStageResponseConsumer** to handle the eventual value of the **Single**. Similarly, when a resource method returns a streaming reactive class like **Flowable**, RESTEasy subscribes to it, receives a stream of data elements, and sends them to the client. **AsyncResponseConsumer** has several supporting classes, each of which implements a different mode of streaming.

For example, **AsyncResponseConsumer.AsyncGeneralStreamingSseResponseConsumer** handles general streaming and SSE streaming. Subscription is done by calling **org.reactivestreams.Publisher.subscribe()**, so it needs a mechanism to turn a **Flowable** into a **Publisher**, for example. That is, an implementation of **org.jboss.resteasy.spi.AsyncStreamProvider<Flowable>** is called for, which defines **AsyncStreamProvider** as shown in the following example:

```
public interface AsyncStreamProvider<T> {
  public Publisher toAsyncStream(T asyncResponse);
}
```

In the **resteasy-rxjava2** module, **org.jboss.resteasy.FlowableProvider** provides that mechanism for **Flowable**.

That means, on the server side, you can add support for other reactive types by declaring a **@Provider** annotation for the **AsyncStreamProvider** interface for streams or the **AsyncResponseProvider** interface for single values. Both these interfaces have a single method to convert the new reactive type into a **Publisher** or a **CompletionStage** for streams or for single values respectively.

On the client side, the Jakarta RESTful Web Services 2.1 imposes two requirements for support of the reactive classes:

- Support for **CompletionStage** as an implementation of the **javax.ws.rs.client.CompletionStageRxInvoker** interface.
- Extensibility by supporting the registration of providers that implement:

```
public interface RxInvokerProvider<T extends RxInvoker> {
   public boolean isProviderFor(Class<T> clazz);
   public T getRxInvoker(SyncInvoker syncInvoker, ExecutorService executorService);
}
```

Once an RxInvokerProvider is registered, you can request an RxInvoker by calling the **javax.ws.rs.client.Invocation.Builder** method:

```
public <T extends RxInvoker> T rx(Class<T> clazz);
```

You can use the RxInvoker for making an invocation that returns the appropriate reactive class. For example:

```
FlowableRxInvoker invoker =
client.target(generateURL("/get/string")).request().rx(FlowableRxInvoker.class);
Flowable<String> flowable = (Flowable<String>) invoker.get();
```

RESTEasy provides partial support for implementing **RxInvokers**. For example, **SingleProvider**, mentioned above, also implements **org.jboss.resteasy.spi.AsyncClientResponseProvider<Single<?>>**, where **AsyncClientResponseProvider** is defined as the following:

```
public interface AsyncClientResponseProvider<T> {
  public T fromCompletionStage(CompletionStage<?> completionStage);
}
```

2.21.3. Reactive Clients API

RESTEasy defines a new type of invoker named **RxInvoker**, and a default implementation of this type named **CompletionStageRxInvoker**. **CompletionStageRxInvoker** implements Java 8's interface **CompletionStage**. This interface declares a large number of methods dedicated to managing asynchronous computations.

2.21.4. Asynchronous Filters

If you must suspend execution of your filter until a certain resource is available, you can convert it into an asynchronous filter. Turning a request asynchronous does not require any change to your resource method declaration or the additional filter declaration.

To turn a filter's execution asynchronous, you must cast:

- The **ContainerRequestContext** into **SuspendableContainerRequestContext** for pre and post request filters.
- The **ContainerResponseContext** into a **SuspendableContainerResponseContext** for response filters.

These context objects can turn the current filter's execution into asynchronous by calling the **suspend()** method. Once asynchronous, the filter chain is suspended, and resumes only after one of the following methods is called on the context object:

- **abortWith(Response)**: Terminate the filter chain, return the given Response to the client. This applies only to ContainerRequestFilter.
- **resume()**: Resume execution of the filter chain by calling the next filter.
- **resume(Throwable)**: Abort execution of the filter chain by throwing the given exception. This behaves as if the filter were synchronous and threw the given exception.

2.21.5. Proxies

Proxies are a RESTEasy extension that supports an intuitive programming style, which replaces generic Jakarta RESTful Web Services invoker calls with application-specific interface calls. The proxy framework is extended to include both **CompletionStage** and the RxJava2 types **Single**, **Observable**, and **Flowable**. The two following examples illustrate how RESTEasy proxies work:

Example 1:

```
@Path("")
public interface RxCompletionStageResource {

  @GET
  @Path("get/string")
  @Produces(MediaType.TEXT_PLAIN)
  public CompletionStage<String> getString();
}

@Path("")
public class RxCompletionStageResourceImpl {

  @GET
  @Path("get/string")
  @Produces(MediaType.TEXT_PLAIN)
  public CompletionStage<String> getString() { .... }
}

public class RxCompletionStageProxyTest {

  private static ResteasyClient client;
  private static RxCompletionStageResource proxy;

  static {
    client = new ResteasyClientBuilder().build();
    proxy = client.target(generateURL("/")).proxy(RxCompletionStageResource.class);
  }

  @Test
  public void testGet() throws Exception {
    CompletionStage<String> completionStage = proxy.getString();
    Assert.assertEquals("x", completionStage.toCompletableFuture().get());
  }
}
```

Example 2:

```
public interface Rx2FlowableResource {

  @GET
  @Path("get/string")
  @Produces(MediaType.TEXT_PLAIN)
  @Stream
  public Flowable<String> getFlowable();
}

@Path("")
public class Rx2FlowableResourceImpl {

  @GET
  @Path("get/string")
  @Produces(MediaType.TEXT_PLAIN)
  @Stream
  public Flowable<String> getFlowable() { ... }
}
```

```
public class Rx2FlowableProxyTest {

  private static ResteasyClient client;
  private static Rx2FlowableResource proxy;

  static {
    client = new ResteasyClientBuilder().build();
    proxy = client.target(generateURL("/")).proxy(Rx2FlowableResource.class);
  }

  @Test
  public void testGet() throws Exception {
    Flowable<String> flowable = proxy.getFlowable();
    flowable.subscribe(
      (String o) -> stringList.add(o),
      (Throwable t) -> errors.incrementAndGet(),
      () -> latch.countDown());
    boolean waitResult = latch.await(30, TimeUnit.SECONDS);
    Assert.assertTrue("Waiting for event to be delivered has timed out.", waitResult);
    Assert.assertEquals(0, errors.get());
    Assert.assertEquals(xStringList, stringList);
  }
}
```

CHAPTER 3. DEVELOPING JAKARTA XML WEB SERVICES

Jakarta XML Web Services defines the mapping between WSDL and Java, as well as the classes to be used for accessing web services and publishing them. JBossWS implements Jakarta XML Web Services 2.3, which users can reference for any vendor-agnostic web service usage need.

3.1. USING JAKARTA XML WEB SERVICES TOOLS

The following Jakarta XML Web Services command-line tools are included with the JBoss EAP distribution. These tools can be used in a variety of ways for server and client-side development.

Table 3.1. Jakarta XML Web Services Command-Line Tools

Command	Description
wsprovide	Generates Jakarta XML Web Services portable artifacts, and provides the abstract contract. Used for bottom-up development.
wsconsume	Consumes the abstract contract (WSDL and Schema files), and produces artifacts for both a server and client. Used for top-down and client development.

See Jakarta XML Web Services Tools for more details on the usage of these tools.

3.1.1. Server-side Development Strategies

When developing a web service endpoint on the server side, you have the option of starting from Java code, known as *bottom-up development*, or from the WSDL that defines your service, known as *top-down development*. If this is a new service, meaning that there is no existing contract, then the bottom-up approach is the fastest route; you only need to add a few annotations to your classes to get a service up and running. However, if you are developing a service with a contract already defined, it is far simpler to use the top-down approach, since the tool can generate the annotated code for you.

Bottom-up use cases:

- Exposing an already existing Jakarta Enterprise Beans 3 bean as a web service.
- Providing a new service, and you want the contract to be generated for you.

Top-down use cases:

- Replacing the implementation of an existing web service, and you can not break compatibility with older clients.
- Exposing a service that conforms to a contract specified by a third party, for example, a vendor that calls you back using an already defined protocol.
- Creating a service that adheres to the XML Schema and WSDL you developed by hand up front.

Bottom-Up Strategy Using wsprovide
The bottom-up strategy involves developing the Java code for your service, and then annotating it using Jakarta XML Web Services annotations. These annotations can be used to customize the contract that is generated for your service. For example, you can change the operation name to map to anything you like. However, all of the annotations have sensible defaults, so only the **@WebService** annotation is required.

This can be as simple as creating a single class:

```
package echo;

@javax.jws.WebService
public class Echo {

  public String echo(String input) {
    return input;
  }
}
```

A deployment can be built using this class, and it is the only Java code needed to deploy on JBossWS. The WSDL, and all other Java artifacts called *wrapper classes* will be generated for you at deploy time.

The primary purpose of the **wsprovide** tool is to generate portable Jakarta XML Web Services artifacts. Additionally, it can be used to provide the WSDL file for your service. This can be obtained by invoking **wsprovide** using the **-w** option:

```
$ javac -d . Echo.java
$ EAP_HOME/bin/wsprovide.sh --classpath=. -w echo.Echo
```

Inspecting the WSDL reveals a service named **EchoService**:

```
<wsdl:service name="EchoService">
  <wsdl:port name="EchoPort" binding="tns:EchoServiceSoapBinding">
    <soap:address location="http://localhost:9090/EchoPort"/>
  </wsdl:port>
</wsdl:service>
```

As expected, this service defines an operation, **echo**:

```
<wsdl:portType name="Echo">
  <wsdl:operation name="echo">
    <wsdl:input name="echo" message="tns:echo">
    </wsdl:input>
    <wsdl:output name="echoResponse" message="tns:echoResponse">
    </wsdl:output>
  </wsdl:operation>
</wsdl:portType>
```

When deploying you do not need to run this tool. You only need it for generating portable artifacts or the abstract contract for your service.

A POJO endpoint for the deployment can be created in a simple **web.xml** file:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-
app_2_4.xsd"
  version="2.4">

  <servlet>
    <servlet-name>Echo</servlet-name>
```

```
    <servlet-class>echo.Echo</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Echo</servlet-name>
    <url-pattern>/Echo</url-pattern>
  </servlet-mapping>
</web-app>
```

The **web.xml** and the single Java class can now be used to create a WAR:

```
$ mkdir -p WEB-INF/classes
$ cp -rp echo WEB-INF/classes/
$ cp web.xml WEB-INF
$ jar cvf echo.war WEB-INF
added manifest
adding: WEB-INF/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/echo/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/echo/Echo.class(in = 340) (out= 247)(deflated 27%)
adding: WEB-INF/web.xml(in = 576) (out= 271)(deflated 52%)
```

The WAR can then be deployed to JBoss EAP. This will internally invoke **wsprovide**, which will generate the WSDL. If the deployment was successful, and you are using the default settings, it should be available in the management console.

NOTE

For a portable Jakarta XML Web Services deployment, the wrapper classes generated earlier could be added to the deployment.

Top-Down Strategy Using wsconsume

The top-down development strategy begins with the abstract contract for the service, which includes the WSDL file and zero or more schema files. The **wsconsume** tool is then used to consume this contract, and produce annotated Java classes, and optionally sources, that define it.

NOTE

wsconsume might have problems with symlinks on Unix systems.

Using the WSDL file from the bottom-up example, a new Java implementation that adheres to this service can be generated. The **-k** option is passed to **wsconsume** to preserve the Java source files that are generated, instead of providing just Java classes:

```
$ EAP_HOME/bin/wsconsume.sh -k EchoService.wsdl
```

The following table shows the purpose of each generated file:

Table 3.2. Generated Files

File	Purpose
Echo.java	Service Endpoint Interface

File	Purpose
EchoResponse.java	Wrapper bean for response message
EchoService.java	Used only by Jakarta XML Web Services clients
Echo_Type.java	Wrapper bean for request message
ObjectFactory.java	Jakarta XML Binding XML Registry
package-info.java	Holder for Jakarta XML Binding package annotations

Examining the service endpoint interface reveals annotations that are more explicit than in the class written by hand in the bottom-up example, however, these evaluate to the same contract.

```
@WebService(targetNamespace = "http://echo/", name = "Echo")
@XmlSeeAlso({ObjectFactory.class})
public interface Echo {

    @WebMethod
    @RequestWrapper(localName = "echo", targetNamespace = "http://echo/", className =
"echo.Echo_Type")
    @ResponseWrapper(localName = "echoResponse", targetNamespace = "http://echo/", className
= "echo.EchoResponse")
    @WebResult(name = "return", targetNamespace = "")
    public java.lang.String echo(
        @WebParam(name = "arg0", targetNamespace = "")
        java.lang.String arg0
    );
}
```

The only missing piece, other than for packaging, is the implementation class, which can now be written using the above interface.

```
package echo;

@javax.jws.WebService(endpointInterface="echo.Echo")
public class EchoImpl implements Echo {
  public String echo(String arg0) {
    return arg0;
  }
}
```

3.1.2. Client-side Development Strategies

Before going in to detail on the client side, it is important to understand the decoupling concept that is central to web services. Web services are not the best fit for internal RPC, even though they can be used in this way. There are much better technologies for this, such as CORBA and RMI. Web services were designed specifically for interoperable coarse-grained correspondence. There is no expectation or

guarantee that any party participating in a web service interaction will be at any particular location, running on any particular operating system, or written in any particular programming language. So because of this, it is important to clearly separate client and server implementations. The only thing they should have in common is the abstract contract definition. If, for whatever reason, your software does not adhere to this principal, then you should not be using web services. For the above reasons, the recommended methodology for developing a client is to follow the top-down approach, even if the client is running on the same server.

Top-Down Strategy Using wsconsume
This section repeats the process of the server-side top-down section, however, it uses a deployed WSDL. This is to retrieve the correct value for **soap:address**, shown below, which is computed at deploy time. This value can be edited manually in the WSDL if necessary, but you must take care to provide the correct path.

Example: soap:address in a Deployed WSDL

```
<wsdl:service name="EchoService">
  <wsdl:port name="EchoPort" binding="tns:EchoServiceSoapBinding">
    <soap:address location="http://localhost.localdomain:8080/echo/Echo"/>
  </wsdl:port>
</wsdl:service>
```

Use **wsconsume** to generate Java classes for the deployed WSDL.

```
$ EAP_HOME/bin/wsconsume.sh -k http://localhost:8080/echo/Echo?wsdl
```

Notice how the **EchoService.java** class stores the location from which the WSDL was obtained.

```
@WebServiceClient(name = "EchoService",
                  wsdlLocation = "http://localhost:8080/echo/Echo?wsdl",
                  targetNamespace = "http://echo/")
public class EchoService extends Service {

    public final static URL WSDL_LOCATION;

    public final static QName SERVICE = new QName("http://echo/", "EchoService");
    public final static QName EchoPort = new QName("http://echo/", "EchoPort");

    ...

    @WebEndpoint(name = "EchoPort")
    public Echo getEchoPort() {
        return super.getPort(EchoPort, Echo.class);
    }

    @WebEndpoint(name = "EchoPort")
    public Echo getEchoPort(WebServiceFeature... features) {
        return super.getPort(EchoPort, Echo.class, features);
    }
}
```

As you can see, this generated class extends the main client entry point in Jakarta XML Web Services, **javax.xml.ws.Service**. While you can use **Service** directly, this is far simpler since it provides the configuration information for you. Note the **getEchoPort()** method, which returns an instance of our

service endpoint interface. Any web service operation can then be called by just invoking a method on the returned interface.

IMPORTANT

Do not refer to a remote WSDL URL in a production application. This causes network I/O every time you instantiate the **Service** object. Instead, use the tool on a saved local copy, or use the URL version of the constructor to provide a new WSDL location.

Write and compile the client:

```
import echo.*;

public class EchoClient {

  public static void main(String args[]) {

    if (args.length != 1) {
        System.err.println("usage: EchoClient <message>");
        System.exit(1);
    }

    EchoService service = new EchoService();
    Echo echo = service.getEchoPort();
    System.out.println("Server said: " + echo.echo(args0));
  }
}
```

You can change the endpoint address of your operation at runtime, by setting the **ENDPOINT_ADDRESS_PROPERTY** as shown below:

```
EchoService service = new EchoService();
Echo echo = service.getEchoPort();

/* Set NEW Endpoint Location */
String endpointURL = "http://NEW_ENDPOINT_URL";
BindingProvider bp = (BindingProvider)echo;
bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);

System.out.println("Server said: " + echo.echo(args0));
```

3.2. JAKARTA XML WEB SERVICES WEB SERVICE ENDPOINTS

3.2.1. About Jakarta XML Web Services Web Service Endpoints

A Jakarta XML Web Services web service endpoint is the server component of a web service. Clients and other web services communicate with it over the HTTP protocol using an XML language called Simple Object Access Protocol (SOAP). The endpoint itself is deployed into the JBoss EAP container.

WSDL descriptors can be created in one of the following two ways:

- Writing WSDL descriptors manually.

- Using Jakarta XML Web Services annotations that create the WSDL descriptors automatically. This is the most common method for creating WSDL descriptors.

An endpoint implementation bean is annotated with Jakarta XML Web Services annotations and deployed to the server. The server automatically generates and publishes the abstract contract in WSDL format for client consumption. All marshalling and unmarshalling is delegated to the Jakarta XML Binding service.

The endpoint itself might be a Plain Old Java Object (POJO) or a Jakarta EE web application. You can also expose endpoints using a Jakarta Enterprise Beans 3 stateless session bean. It is packaged into a web archive (WAR) file. The specification for packaging the endpoint is defined in the Jakarta Web Services Metadata Specification 2.1.

Example: POJO Endpoint

```
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class JSEBean {
    @WebMethod
    public String echo(String input) {
        ...
    }
}
```

Example: Web Services Endpoint

```
<web-app ...>
  <servlet>
    <servlet-name>TestService</servlet-name>
    <servlet-class>org.jboss.quickstarts.ws.jaxws.samples.jsr181pojo.JSEBean01</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>TestService</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

The following Jakarta Enterprise Beans 3 stateless session bean exposes the same method on the remote interface as well as an endpoint operation.

```
@Stateless
@Remote(EJB3RemoteInterface.class)

@WebService

@SOAPBinding(style = SOAPBinding.Style.RPC)
public class EJB3Bean implements EJB3RemoteInterface {
    @WebMethod
    public String echo(String input) {
        ...
    }
}
```

Service Endpoint Interface

Jakarta XML Web Services services typically implement a Java service endpoint interface (SEI), which might be mapped from a WSDL port type, either directly or using annotations. This SEI provides a high-level abstraction that hides the details between Java objects and their XML representations.

Endpoint Provider Interface

In some cases, Jakarta XML Web Services services need the ability to operate at the XML message level. The endpoint **Provider** interface provides this functionality to the web services that implement it.

Consuming and Accessing the Endpoint

After you deploy your web service, you can consume the WSDL to create the component stubs which will be the basis for your application. Your application can then access the endpoint to do its work.

3.2.2. Developing and Deploying Jakarta XML Web Services Web Service Endpoint

A Jakarta XML Web Services service endpoint is a server-side component that responds to requests from Jakarta XML Web Services clients and publishes the WSDL definition for itself.

See the following quickstarts that ship with JBoss EAP for working examples of how to develop Jakarta XML Web Services endpoint applications.

- jaxws-addressing
- jaxws-ejb
- jaxws-pojo
- jaxws-retail
- wsat-simple
- wsba-coordinator-completion-simple
- wsba-participant-completion-simple

Development Requirements

A web service must fulfill the requirements of the Jakarta XML Web Services API and Jakarta Web Services Metadata Specification 2.1 specification.

- It contains a **javax.jws.WebService** annotation.
- All method parameters and return types are compatible with Jakarta XML Binding 2.3 specification.

The following is an example of a web service implementation that meets these requirements.

Example: Web Service Implementation

```
package org.jboss.quickstarts.ws.jaxws.samples.retail.profile;

import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;

@Stateless
```

```
@WebService(
    name = "ProfileMgmt",
    targetNamespace = "http://org.jboss.ws/samples/retail/profile",
    serviceName = "ProfileMgmtService")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
public class ProfileMgmtBean {
    @WebMethod
    public DiscountResponse getCustomerDiscount(DiscountRequest request) {
        DiscountResponse dResponse = new DiscountResponse();
        dResponse.setCustomer(request.getCustomer());
        dResponse.setDiscount(10.00);
        return dResponse;
    }
}
```

The following is an example of the **DiscountRequest** class that is used by the **ProfileMgmtBean** bean in the previous example. The annotations are included for verbosity. Typically, the Jakarta XML Binding defaults are reasonable and do not need to be specified.

Example: DiscountRequest Class

```
package org.jboss.test.ws.jaxws.samples.retail.profile;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

import org.jboss.test.ws.jaxws.samples.retail.Customer;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(
  name = "discountRequest",
  namespace="http://org.jboss.ws/samples/retail/profile",
  propOrder = { "customer" }
)
public class DiscountRequest {

    protected Customer customer;

    public DiscountRequest() {
    }

    public DiscountRequest(Customer customer) {
      this.customer = customer;
    }

    public Customer getCustomer() {
      return customer;
    }

    public void setCustomer(Customer value) {
      this.customer = value;
    }

}
```

Packaging Your Deployment

The implementation class is wrapped in a JAR deployment. Any metadata required for deployment is taken from the annotations on the implementation class and the service endpoint interface. You can deploy the JAR using the management CLI or the management console, and the HTTP endpoint is created automatically.

The following listing shows an example of the structure for a JAR deployment of a Jakarta Enterprise Beans web service.

```
$ jar -tf jaxws-samples-retail.jar
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.class
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.class
org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtBean.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.class
org/jboss/test/ws/jaxws/samples/retail/profile/package-info.class
```

3.3. JAKARTA XML WEB SERVICES WEB SERVICE CLIENTS

3.3.1. Consume and Access a Jakarta XML Web Services Web Service

After creating a web service endpoint, either manually or using Jakarta XML Web Services annotations, you can access its WSDL. This WSDL can be used to create the basic client application that will communicate with the web service. The process of generating Java code from the published WSDL is called consuming the web service. This happens in the following phases:

1. Create the client artifacts.
2. Construct a service stub.

Create the Client Artifacts

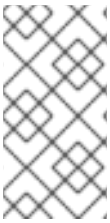
Before you can create client artifacts, you need to create your WSDL contract. The following WSDL contract is used for the examples presented in the rest of this section.

The examples below rely on having this WSDL contract in the **ProfileMgmtService.wsdl** file.

```
<definitions
   name='ProfileMgmtService'
   targetNamespace='http://org.jboss.ws/samples/retail/profile'
   xmlns='http://schemas.xmlsoap.org/wsdl/'
   xmlns:ns1='http://org.jboss.ws/samples/retail'
   xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
   xmlns:tns='http://org.jboss.ws/samples/retail/profile'
   xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

  <types>

    <xs:schema targetNamespace='http://org.jboss.ws/samples/retail'
            version='1.0' xmlns:xs='http://www.w3.org/2001/XMLSchema'>
      <xs:complexType name='customer'>
        <xs:sequence>
          <xs:element minOccurs='0' name='creditCardDetails' type='xs:string'/>
          <xs:element minOccurs='0' name='firstName' type='xs:string'/>
```

```
            <xs:element minOccurs='0' name='lastName' type='xs:string'/>
          </xs:sequence>
        </xs:complexType>
      </xs:schema>

      <xs:schema
          targetNamespace='http://org.jboss.ws/samples/retail/profile'
          version='1.0'
          xmlns:ns1='http://org.jboss.ws/samples/retail'
          xmlns:tns='http://org.jboss.ws/samples/retail/profile'
          xmlns:xs='http://www.w3.org/2001/XMLSchema'>

        <xs:import namespace='http://org.jboss.ws/samples/retail'/>
        <xs:element name='getCustomerDiscount'
                    nillable='true' type='tns:discountRequest'/>
        <xs:element name='getCustomerDiscountResponse'
                    nillable='true' type='tns:discountResponse'/>
        <xs:complexType name='discountRequest'>
          <xs:sequence>
            <xs:element minOccurs='0' name='customer' type='ns1:customer'/>

          </xs:sequence>
        </xs:complexType>
        <xs:complexType name='discountResponse'>
          <xs:sequence>
            <xs:element minOccurs='0' name='customer' type='ns1:customer'/>
            <xs:element name='discount' type='xs:double'/>
          </xs:sequence>
        </xs:complexType>
      </xs:schema>

    </types>

    <message name='ProfileMgmt_getCustomerDiscount'>
      <part element='tns:getCustomerDiscount' name='getCustomerDiscount'/>
    </message>
    <message name='ProfileMgmt_getCustomerDiscountResponse'>
      <part element='tns:getCustomerDiscountResponse'
            name='getCustomerDiscountResponse'/>
    </message>
    <portType name='ProfileMgmt'>
      <operation name='getCustomerDiscount'
                 parameterOrder='getCustomerDiscount'>

        <input message='tns:ProfileMgmt_getCustomerDiscount'/>
        <output message='tns:ProfileMgmt_getCustomerDiscountResponse'/>
      </operation>
    </portType>
    <binding name='ProfileMgmtBinding' type='tns:ProfileMgmt'>
      <soap:binding style='document'
                    transport='http://schemas.xmlsoap.org/soap/http'/>
      <operation name='getCustomerDiscount'>
        <soap:operation soapAction=''/>
        <input>

          <soap:body use='literal'/>
```

```
        </input>
        <output>
          <soap:body use='literal'/>
        </output>
      </operation>
    </binding>
    <service name='ProfileMgmtService'>
      <port binding='tns:ProfileMgmtBinding' name='ProfileMgmtPort'>

        <!-- service address will be rewritten to actual one when WSDL is requested from running server
-->
        <soap:address location='http://SERVER:PORT/jaxws-retail/ProfileMgmtBean'/>
      </port>
    </service>
</definitions>
```



NOTE

If you use Jakarta XML Web Services annotations to create your web service endpoint, the WSDL contract is generated automatically, and you only need its URL. You can find this URL by navigating to **Runtime**, selecting the applicable server, selecting **Webservices**, then choosing the endpoint.

The **wsconsume.sh** or **wsconsume.bat** tool is used to consume the abstract contract (WSDL) and produce annotated Java classes and optional sources that define it. The tool is located in the ***EAP_HOME*/bin/** directory.

```
$ ./wsconsume.sh --help
WSConsumeTask is a cmd line tool that generates portable JAX-WS artifacts from a WSDL file.

usage: org.jboss.ws.tools.cmd.WSConsume [options] <wsdl-url>

options:
   -h, --help                Show this help message
   -b, --binding=<file>      One or more JAX-WS or Java XML Binding files
   -k, --keep                Keep/Generate Java source
   -c  --catalog=<file>      Oasis XML Catalog file for entity resolution
   -p  --package=<name>      The target package for generated source
   -w  --wsdlLocation=<loc>  Value to use for @WebService.wsdlLocation
   -o, --output=<directory>  The directory to put generated artifacts
   -s, --source=<directory>  The directory to put Java source
   -t, --target=<2.0|2.1|2.2>  The JAX-WS target
   -q, --quiet               Be somewhat more quiet
   -v, --verbose             Show full exception stack traces
   -l, --load-consumer       Load the consumer and exit (debug utility)
   -e, --extension           Enable SOAP 1.2 binding extension
   -a, --additionalHeaders   Enable processing of implicit SOAP headers
   -n, --nocompile           Do not compile generated sources
```

The following command generates the source **.java** files listed in the output, from the **ProfileMgmtService.wsdl** file. The sources use the directory structure of the package, which is specified with the **-p** switch.

```
[user@host bin]$ wsconsume.sh -k -p org.jboss.test.ws.jaxws.samples.retail.profile
```

```
ProfileMgmtService.wsdl
output/org/jboss/test/ws/jaxws/samples/retail/profile/Customer.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/package-info.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/Customer.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/package-info.class
```

Both **.java** source files and compiled **.class** files are generated into the **output**/ directory within the directory where you run the command.

Table 3.3. Descriptions of Artifacts Created by**wsconsume.sh**

File	Description
ProfileMgmt.java	Service endpoint interface.
Customer.java	Custom data type.
Discount.java	Custom data types.
ObjectFactory.java	Jakarta XML Binding XML registry.
package-info.java	Jakarta XML Binding package annotations.
ProfileMgmtService.java	Service factory.

The **wsconsume** command generates all custom data types (Jakarta XML Binding annotated classes), the service endpoint interface, and a service factory class. These artifacts are used to build web service client implementations.

Construct a Service Stub

Web service clients use service stubs to abstract the details of a remote web service invocation. To a client application, a web service invocation looks like an invocation of any other business component. In this case the service endpoint interface acts as the business interface, and a service factory class is not used to construct it as a service stub.

The following example first creates a service factory using the WSDL location and the service name. Next, it uses the service endpoint interface created by **wsconsume** to build the service stub. Finally, the stub can be used just as any other business interface would be.

You can find the WSDL URL for your endpoint in the JBoss EAP management console. You can find this URL by navigating to **Runtime**, selecting the applicable server, selecting **Webservices**, then choosing the endpoint.

```
import javax.xml.ws.Service;
[...]
Service service = Service.create(
new URL("http://example.org/service?wsdl"),
new QName("MyService")
);
ProfileMgmt profileMgmt = service.getPort(ProfileMgmt.class);

// Use the service stub in your application
```

3.3.2. Develop a Jakarta XML Web Services Client Application

The client communicates with, and requests work from, the Jakarta XML Web Services endpoint, which is deployed in the Java Enterprise Edition 7 container. For detailed information about the classes, methods, and other implementation details mentioned below, see the relevant sections of the Javadocs bundle included with JBoss EAP.

Overview
A **Service** is an abstraction which represents a WSDL service. A WSDL service is a collection of related ports, each of which includes a port type bound to a particular protocol and a particular endpoint address.

Usually, the Service is generated when the rest of the component stubs are generated from an existing WSDL contract. The WSDL contract is available via the WSDL URL of the deployed endpoint, or can be created from the endpoint source using the **wsprovide** tool in the ***EAP_HOME*/bin/** directory.

This type of usage is referred to as the static use case. In this case, you create instances of the **Service** class which is created as one of the component stubs.

You can also create the service manually, using the **Service.create** method. This is referred to as the dynamic use case.

Usage
Static Use Case
The static use case for a Jakarta XML Web Services client assumes that you already have a WSDL contract. This might be generated by an external tool or generated by using the correct Jakarta XML Web Services annotations when you create your Jakarta XML Web Services endpoint.

To generate your component stubs, you use the **wsconsume** tool included in ***EAP_HOME*/bin**. The tool takes the WSDL URL or file as a parameter, and generates multiple files, structured in a directory tree. The source and class files representing your **Service** are named **_Service.java** and **_Service.class**, respectively.

The generated implementation class has two public constructors, one with no arguments and one with two arguments. The two arguments represent the WSDL location (a **java.net.URL**) and the service name (a **javax.xml.namespace.QName**) respectively.

The no-argument constructor is the one used most often. In this case the WSDL location and service name are those found in the WSDL. These are set implicitly from the **@WebServiceClient** annotation that decorates the generated class.

```
@WebServiceClient(name="StockQuoteService", targetNamespace="http://example.com/stocks",
wsdlLocation="http://example.com/stocks.wsdl")
public class StockQuoteService extends javax.xml.ws.Service
{
```

```
    public StockQuoteService() {
      super(new URL("http://example.com/stocks.wsdl"), new QName("http://example.com/stocks",
"StockQuoteService"));
    }

    public StockQuoteService(String wsdlLocation, QName serviceName) {
      super(wsdlLocation, serviceName);
    }
    ...
}
```

For details about how to obtain a port from the service and how to invoke an operation on the port, see Dynamic Proxy. For details about how to work with the XML payload directly or with the XML representation of the entire SOAP message, see Dispatch.

Dynamic Use Case

In the dynamic case, no stubs are generated automatically. Instead, a web service client uses the **Service.create** method to create **Service** instances. The following code fragment illustrates this process.

```
URL wsdlLocation = new URL("http://example.org/my.wsdl");
QName serviceName = new QName("http://example.org/sample", "MyService");
Service service = Service.create(wsdlLocation, serviceName);
```

Handler Resolver

Jakarta XML Web Services provides a flexible plug-in framework for message processing modules, known as handlers. These handlers extend the capabilities of a Jakarta XML Web Services runtime system. A **Service** instance provides access to a **HandlerResolver** via a pair of **getHandlerResolver** and **setHandlerResolver** methods that can configure a set of handlers on a per-service, per-port or per-protocol binding basis.

When a **Service** instance creates a proxy or a **Dispatch** instance, the handler resolver currently registered with the service creates the required handler chain. Subsequent changes to the handler resolver configured for a **Service** instance do not affect the handlers on previously created proxies or **Dispatch** instances.

Executor

Service instances can be configured with a **java.util.concurrent.Executor**. The **Executor** invokes any asynchronous callbacks requested by the application. The **setExecutor** and **getExecutor** methods of **Service** can modify and retrieve the **Executor** configured for a service.

Dynamic Proxy

A dynamic proxy is an instance of a client proxy using one of the **getPort** methods provided in the **Service**. The **portName** specifies the name of the WSDL port the service uses. The **serviceEndpointInterface** specifies the service endpoint interface supported by the created dynamic proxy instance.

```
public <T> T getPort(QName portName, Class<T> serviceEndpointInterface)
public <T> T getPort(Class<T> serviceEndpointInterface)
```

The Service Endpoint Interface is usually generated using the **wsconsume** tool, which parses the WSDL and creates Java classes from it.

A typed method, which returns a port, is also provided. These methods also return dynamic proxies that implement the SEI. See the following example.

-

```
@WebServiceClient(name = "TestEndpointService", targetNamespace = "http://org.jboss.ws/wsref",
  wsdlLocation = "http://localhost.localdomain:8080/jaxws-samples-webserviceref?wsdl")

public class TestEndpointService extends Service {
   ...

   public TestEndpointService(URL wsdlLocation, QName serviceName) {
      super(wsdlLocation, serviceName);
   }

   @WebEndpoint(name = "TestEndpointPort")
   public TestEndpoint getTestEndpointPort() {
      return (TestEndpoint)super.getPort(TESTENDPOINTPORT, TestEndpoint.class);
   }
}
```

@WebServiceRef

The **@WebServiceRef** annotation declares a reference to a web service. It follows the resource pattern shown by the **javax.annotation.Resource** annotation defined in JSR 250. The Jakarta EE equivalent for these annotations is in the Jakarta Annotations 1.3 specification.

- You can use it to define a reference whose type is a generated **Service** class. In this case, the type and value element each refer to the generated **Service** class type. Moreover, if the reference type can be inferred by the field or method declaration the annotation is applied to, the type and value elements might, but are not required to, have the default value of **Object.class**. If the type cannot be inferred, then at least the type element must be present with a non-default value.

- You can use it to define a reference whose type is an SEI. In this case, the type element might (but is not required to) be present with its default value if the type of the reference can be inferred from the annotated field or method declaration. However, the value element must always be present and refer to a generated service class type, which is a subtype of **javax.xml.ws.Service**. The **wsdlLocation** element, if present, overrides the WSDL location information specified in the **@WebService** annotation of the referenced generated service class.

```
public class EJB3Client implements EJB3Remote
{
  @WebServiceRef
  public TestEndpointService service4;

  @WebServiceRef
  public TestEndpoint port3;
 }
```

Dispatch

XML web services use XML messages for communication between the endpoint, which is deployed in the Jakarta EE container, and any clients. The XML messages use an XML language called Simple Object Access Protocol (SOAP). The Jakarta XML Web Services API provides the mechanisms for the endpoint and clients to each be able to send and receive SOAP messages. Marshalling is the process of converting a Java Object into a SOAP XML message. Unmarshalling is the process of converting the SOAP XML message back into a Java Object.

In some cases, you need access to the raw SOAP messages themselves, rather than the result of the conversion. The **Dispatch** class provides this functionality. **Dispatch** operates in one of two usage modes, which are identified by one of the following constants.

- **javax.xml.ws.Service.Mode.MESSAGE** - This mode directs client applications to work directly with protocol-specific message structures. When used with a SOAP protocol binding, a client application works directly with a SOAP message.
- **javax.xml.ws.Service.Mode.PAYLOAD** - This mode causes the client to work with the payload itself. For instance, if it is used with a SOAP protocol binding, a client application would work with the contents of the SOAP body rather than the entire SOAP message.

Dispatch is a low-level API which requires clients to structure messages or payloads as XML, with strict adherence to the standards of the individual protocol and a detailed knowledge of message or payload structure. **Dispatch** is a generic class which supports input and output of messages or message payloads of any type.

```
Service service = Service.create(wsdlURL, serviceName);
Dispatch dispatch = service.createDispatch(portName, StreamSource.class, Mode.PAYLOAD);

String payload = "<ns1:ping xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
dispatch.invokeOneWay(new StreamSource(new StringReader(payload)));

payload = "<ns1:feedback xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
Source retObj = (Source)dispatch.invoke(new StreamSource(new StringReader(payload)));
```

Asynchronous Invocations

The **BindingProvider** interface represents a component that provides a protocol binding which clients can use. It is implemented by proxies and is extended by the **Dispatch** interface.

BindingProvider instances might provide asynchronous operation capabilities. Asynchronous operation invocations are decoupled from the **BindingProvider** instance at invocation time. The response context is not updated when the operation completes. Instead, a separate response context is made available using the **Response** interface.

```
public void testInvokeAsync() throws Exception {
  URL wsdlURL = new URL("http://" + getServerHost() + ":8080/jaxws-samples-asynchronous?
wsdl");
  QName serviceName = new QName(targetNS, "TestEndpointService");
  Service service = Service.create(wsdlURL, serviceName);
  TestEndpoint port = service.getPort(TestEndpoint.class);
  Response response = port.echoAsync("Async");
  // access future
  String retStr = (String) response.get();
  assertEquals("Async", retStr);
}
```

@Oneway Invocations

The **@Oneway** annotation indicates that the given web method takes an input message but returns no output message. Usually, a **@Oneway** method returns the thread of control to the calling application before the business method is executed.

```
@WebService (name="PingEndpoint")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class PingEndpointImpl {
  private static String feedback;

  @WebMethod
  @Oneway
```

```
    public void ping() {
      log.info("ping");
      feedback = "ok";
    }

    @WebMethod
    public String feedback() {
      log.info("feedback");
      return feedback;
    }
}
```

Timeout Configuration
Two different properties control the timeout behavior of the HTTP connection and the timeout of a client which is waiting to receive a message. The first is **javax.xml.ws.client.connectionTimeout** and the second is **javax.xml.ws.client.receiveTimeout**. Each is expressed in milliseconds, and the correct syntax is shown below.

```
public void testConfigureTimeout() throws Exception {
  //Set timeout until a connection is established
  ((BindingProvider)port).getRequestContext().put("javax.xml.ws.client.connectionTimeout", "6000");

  //Set timeout until the response is received
  ((BindingProvider) port).getRequestContext().put("javax.xml.ws.client.receiveTimeout", "1000");

  port.echo("testTimeout");
}
```

3.4. CONFIGURING THE WEB SERVICES SUBSYSTEM

JBossWS components handle the processing of web service endpoints and are provided to JBoss EAP through the **webservices** subsystem. The subsystem supports the configuration of published endpoint addresses and endpoint handler chains.

A default **webservices** subsystem is provided in the server's domain and standalone configuration files. It contains several predefined endpoint and client configurations.

```
<subsystem xmlns="urn:jboss:domain:webservices:2.0">
  <wsdl-host>${jboss.bind.address:127.0.0.1}</wsdl-host>
  <endpoint-config name="Standard-Endpoint-Config"/>
  <endpoint-config name="Recording-Endpoint-Config">
    <pre-handler-chain name="recording-handlers" protocol-bindings="##SOAP11_HTTP
##SOAP11_HTTP_MTOM ##SOAP12_HTTP ##SOAP12_HTTP_MTOM">
      <handler name="RecordingHandler"
class="org.jboss.ws.common.invocation.RecordingServerHandler"/>
    </pre-handler-chain>
  </endpoint-config>
  <client-config name="Standard-Client-Config"/>
</subsystem>
```

3.4.1. Endpoint Configurations

JBossWS enables extra setup configuration data to be predefined and associated with an endpoint implementation. Predefined endpoint configurations can be used for Jakarta XML Web Services client

and Jakarta XML Web Services endpoint setup. Endpoint configurations can include Jakarta XML Web Services handlers and key/value properties declarations. This feature provides a convenient way to add handlers to web service endpoints and to set key/value properties that control JBossWS and Apache CXF internals.

The **webservices** subsystem allows you to define named sets of endpoint configuration data. Each endpoint configuration must have a unique name within the subsystem. The **org.jboss.ws.api.annotation.EndpointConfig** annotation can then be used to assign an endpoint configuration to a Jakarta XML Web Services implementation in a deployed application. See Assigning a Configuration for more information on assigning endpoint configurations.

There are two predefined endpoint configurations in the default JBoss EAP configuration:

- **Standard-Endpoint-Config** is the endpoint configuration used for any endpoint that does not have an explicitly-assigned endpoint configuration.
- **Recording-Endpoint-Config** is an example of custom endpoint configuration that includes a recording handler.

Add an Endpoint Configuration
You can add a new endpoint configuration using the management CLI.

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config:add
```

Configure an Endpoint Configuration
You can add key/value property declarations for the endpoint configuration using the management CLI.

```
/subsystem=webservices/endpoint-config=Standard-Endpoint-
Config/property=PROPERTY_NAME:add(value=PROPERTY_VALUE)
```

You can also configure handler chains and handlers for these endpoint configurations.

Remove an Endpoint Configuration
You can remove a endpoint configuration using the management CLI.

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config:remove
```

3.4.2. Handler Chains

Each endpoint configuration can be associated with **PRE** or **POST** handler chains. Each handler chain may include Jakarta XML Web Services-compliant handlers to perform additional processing on messages. For outbound messages, **PRE** handler chain handlers are executed before any handler attached to the endpoints using standard Jakarta XML Web Services means, such as the **@HandlerChain** annotation. **POST** handler chain handlers are executed after usual endpoint handlers. For inbound messages, the opposite applies.

Server Outbound Messages

```
Endpoint --> PRE Handlers --> Endpoint Handlers --> POST Handlers --> ... --> Client
```

Server Inbound Messages

```
Client --> ... --> POST Handlers --> Endpoint Handlers --> PRE Handlers --> Endpoint
```

Add a Handler Chain
You can add a **POST** handler chain to an endpoint configuration using the following management CLI command.

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-post-handler-chain:add
```

You can add a **PRE** handler chain to an endpoint configuration using the following management CLI command.

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config/pre-handler-chain=my-pre-handler-chain:add
```

Configure a Handler Chain
Use the **protocol-bindings** attribute to set which protocols trigger the handler chain to start.

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-post-handler-chain:write-attribute(name=protocol-bindings,value=##SOAP11_HTTP)
```

See the handlers section for information on configuring handlers for a handler chain.

Remove a Handler Chain
You can remove a handler chain using the management CLI.

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-post-handler-chain:remove
```

3.4.3. Handlers

A Jakarta XML Web Services handler is added to a handler chain and specifies the fully-qualified name of the handler class. When the endpoint is deployed, an instance of that class is created for each referencing deployment. Either the deployment class loader or the class loader for the **org.jboss.as.webservices.server.integration** module must be able to load the handler class.

See the Handler Javadocs for a listing of the available handlers.

Add a Handler
You can add a handler to a handler chain using the following management CLI command. You must provide the class name of the handler.

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-post-handler-chain/handler=my-handler:add(class="com.arjuna.webservices11.wsarj.handler.InstanceIdentifierInHandler")
```

Configure a Handler
You can update the class for a handler using the management CLI.

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-post-handler-chain/handler=my-handler:add(class="org.jboss.ws.common.invocation.RecordingServerHandler")
```

Remove a Handler
You can remove a handler using the management CLI.

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-post-handler-chain/handler=my-handler:
```

3.4.4. Published Endpoint Addresses

The rewriting of the **<soap:address>** element of endpoints published in WSDL contracts is supported. This feature is useful for controlling the server address that is advertised to clients for each endpoint.

The following table lists the attributes that can be configured for this feature.

Name	Description
modify-wsdl-address	This boolean enables and disables the address rewrite functionality. When **modify-wsdl-address** is set to **true** and the content of **<soap:address>** is a valid URL, JBossWS rewrites the URL using the values of **wsdl-host** and **wsdl-port** or **wsdl-secure-port**. When **modify-wsdl-address** is set to **false** and the content of **<soap:address>** is a valid URL, JBossWS does not rewrite the URL. The **<soap:address>** URL is used. When the content of **<soap:address>** is not a valid URL, JBossWS rewrites it no matter what the setting of **modify-wsdl-address**. If **modify-wsdl-address** is set to **true** and **wsdl-host** is not defined or explicitly set to **jbossws.undefined.host**, the content of **<soap:address>** URL is used. JBossWS uses the requester's host when rewriting the **<soap:address>**. When **modify-wsdl-address** is not defined JBossWS uses a default value of **true**.
wsdl-host	The host name or IP address to be used for rewriting **<soap:address>**. If **wsdl-host** is set to **jbossws.undefined.host**, JBossWS uses the requester's host when rewriting the **<soap:address>**. When **wsdl-host** is not defined JBossWS uses a default value of **jbossws.undefined.host**.
wsdl-path-rewrite-rule	This string defines a SED substitution command, for example **s/regexp/replacement/g**, that JBossWS executes against the path component of each **<soap:address>** URL published from the server. When **wsdl-path-rewrite-rule** is not defined, JBossWS retains the original path component of each **<soap:address>** URL. When **modify-wsdl-address** is set to **false** this element is ignored.
wsdl-port	Set this property to explicitly define the HTTP port that will be used for rewriting the SOAP address. Otherwise the HTTP port will be identified by querying the list of installed HTTP connectors.
wsdl-secure-port	Set this property to explicitly define the HTTPS port that will be used for rewriting the SOAP address. Otherwise the HTTPS port will be identified by querying the list of installed HTTPS connectors.

Name	Description
wsdl-uri-scheme	This property explicitly sets the URI scheme to use for rewriting **<soap:address>**. Valid values are **http** and **https**. This configuration overrides the scheme computed by processing the endpoint even if a transport guarantee is specified. The provided values for **wsdl-port** and **wsdl-secure-port**, or their default values, are used depending on the specified scheme.

You can use the management CLI to update these attributes. For example:

```
/subsystem=webservices:write-attribute(name=wsdl-uri-scheme, value=https)
```

3.4.5. Viewing Runtime Information

Each web service endpoint is exposed through the deployment that provides the endpoint implementation. Each endpoint can be queried as a deployment resource. Each web service endpoint specifies a web context and a WSDL URL. You can access this runtime information using the management CLI or the management console.

The following management CLI command shows the details of the **TestService** endpoint from the **jaxws-samples-handlerchain.war** deployment.

```
/deployment="jaxws-samples-handlerchain.war"/subsystem=webservices/endpoint="jaxws-samples-handlerchain:TestService":read-resource(include-runtime=true)
{
    "outcome" => "success",
    "result" => {
        "average-processing-time" => 23L,
        "class" => "org.jboss.test.ws.jaxws.samples.handlerchain.EndpointImpl",
        "context" => "jaxws-samples-handlerchain",
        "fault-count" => 0L,
        "max-processing-time" => 23L,
        "min-processing-time" => 23L,
        "name" => "TestService",
        "request-count" => 1L,
        "response-count" => 1L,
        "total-processing-time" => 23L,
        "type" => "JAXWS_JSE",
        "wsdl-url" => "http://localhost:8080/jaxws-samples-handlerchain?wsdl"
    }
}
```

NOTE

Using the **include-runtime=true** flag on the **read-resource** operation returns runtime statistics in the result. However, the collection of statistics for web service endpoints is disabled by default. You can enable statistics for web service endpoints using the following management CLI command.

```
/subsystem=webservices:write-attribute(name=statistics-enabled,value=true)
```

You can also view runtime information for web services endpoints from the **Runtime** tab of the management console by selecting the applicable server, selecting **Webservices**, then choosing the endpoint.

3.5. ASSIGNING CLIENT AND ENDPOINT CONFIGURATIONS

Client and endpoint configurations can be assigned in the following ways:

- Explicit assignment through annotations, for endpoints, or API programmatic usage, for clients.
- Automatic assignment of configurations from default descriptors.
- Automatic assignment of configurations from the container.

3.5.1. Explicit Configuration Assignment

The explicit configuration assignment is meant for developers that know in advance their endpoint or client has to be set up according to a specified configuration. The configuration is coming from either a descriptor that is included in the application deployment, or is included in the **webservices** subsystem.

3.5.1.1. Configuration Deployment Descriptor

Jakarta EE archives that can contain Jakarta XML Web Services client and endpoint implementations may also contain predefined client and endpoint configuration declarations. All endpoint or client configuration definitions for a given archive must be provided in a single deployment descriptor file, which must be an implementation of the schema that can be found at **EAP_HOME/docs/schema/jbossws-jaxws-config_4_0.xsd**. Many endpoint or client configurations can be defined in the deployment descriptor file. Each configuration must have a name that is unique within the server on which the application is deployed. The configuration name cannot be referred to by endpoint or client implementations outside the application.

Example: Descriptor with Two Endpoint Configurations

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
 xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
 <endpoint-config>
  <config-name>org.jboss.test.ws.jaxws.jbws3282.Endpoint4Impl</config-name>
  <pre-handler-chains>
   <javaee:handler-chain>
    <javaee:handler>
     <javaee:handler-name>Log Handler</javaee:handler-name>
     <javaee:handler-class>org.jboss.test.ws.jaxws.jbws3282.LogHandler</javaee:handler-class>
```

```
      </javaee:handler>
    </javaee:handler-chain>
  </pre-handler-chains>
  <post-handler-chains>
    <javaee:handler-chain>
      <javaee:handler>
        <javaee:handler-name>Routing Handler</javaee:handler-name>
        <javaee:handler-class>org.jboss.test.ws.jaxws.jbws3282.RoutingHandler</javaee:handler-
class>
      </javaee:handler>
    </javaee:handler-chain>
  </post-handler-chains>
 </endpoint-config>
 <endpoint-config>
  <config-name>EP6-config</config-name>
  <post-handler-chains>
    <javaee:handler-chain>
      <javaee:handler>
        <javaee:handler-name>Authorization Handler</javaee:handler-name>
        <javaee:handler-
class>org.jboss.test.ws.jaxws.jbws3282.AuthorizationHandler</javaee:handler-class>
      </javaee:handler>
    </javaee:handler-chain>
  </post-handler-chains>
 </endpoint-config>
</jaxws-config>
```

Similarly, a client configuration can be specified in descriptors, which is still implementing the schema mentioned above:

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
 xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
 <client-config>
  <config-name>Custom Client Config</config-name>
  <pre-handler-chains>
    <javaee:handler-chain>
      <javaee:handler>
        <javaee:handler-name>Routing Handler</javaee:handler-name>
        <javaee:handler-class>org.jboss.test.ws.jaxws.clientConfig.RoutingHandler</javaee:handler-
class>
      </javaee:handler>
      <javaee:handler>
        <javaee:handler-name>Custom Handler</javaee:handler-name>
        <javaee:handler-class>org.jboss.test.ws.jaxws.clientConfig.CustomHandler</javaee:handler-
class>
      </javaee:handler>
    </javaee:handler-chain>
  </pre-handler-chains>
 </client-config>
 <client-config>
  <config-name>Another Client Config</config-name>
  <post-handler-chains>
    <javaee:handler-chain>
```

```
        <javaee:handler>
          <javaee:handler-name>Routing Handler</javaee:handler-name>
          <javaee:handler-class>org.jboss.test.ws.jaxws.clientConfig.RoutingHandler</javaee:handler-
class>
        </javaee:handler>
      </javaee:handler-chain>
    </post-handler-chains>
  </client-config>
</jaxws-config>
```

3.5.1.2. Application Server Configuration

JBoss EAP allows declaring JBossWS client and server predefined configurations in the **webservices** subsystem. As a result it is possible to declare server-wide handlers to be added to the chain of each endpoint or client assigned to a given configuration.

Standard Configuration
Clients running in the same JBoss EAP instance, as well as endpoints, are assigned standard configurations by default. The defaults are used unless different a configuration is set. This enables administrators to tune the default handler chains for client and endpoint configurations. The names of the default client and endpoint configurations used in the **webservices** subsystem are **Standard-Client-Config** and **Standard-Endpoint-Config**.

Handlers Classloading
When setting a server-wide handler, the handler class needs to be available through each ws deployment classloader. As a result proper module dependencies may need to be specified in the deployments that are going to use a given predefined configuration. One way to ensure the proper module dependencies are specified in the deployment is to add a dependency to the module containing the handler class in one of the modules which are already automatically set as dependencies to any deployment, for instance **org.jboss.ws.spi**.

Example Configuration

Example: Default Subsystem Configuration

```
<subsystem xmlns="urn:jboss:domain:webservices:2.0">
    <!-- ... -->
    <endpoint-config name="Standard-Endpoint-Config"/>
    <endpoint-config name="Recording-Endpoint-Config">
        <pre-handler-chain name="recording-handlers" protocol-bindings="##SOAP11_HTTP
##SOAP11_HTTP_MTOM ##SOAP12_HTTP ##SOAP12_HTTP_MTOM">
            <handler name="RecordingHandler"
class="org.jboss.ws.common.invocation.RecordingServerHandler"/>
        </pre-handler-chain>
    </endpoint-config>
    <client-config name="Standard-Client-Config"/>
</subsystem>
```

A configuration file for a deployment specific ws-security endpoint setup:

```
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:javaee="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="urn:jboss:jbossws-jaxws-
config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <endpoint-config>
```

```
    <config-name>Custom WS-Security Endpoint</config-name>
    <property>
      <property-name>ws-security.signature.properties</property-name>
      <property-value>bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.properties</property-name>
      <property-value>bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.signature.username</property-name>
      <property-value>bob</property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.username</property-name>
      <property-value>alice</property-value>
    </property>
    <property>
      <property-name>ws-security.callback-handler</property-name>
      <property-
value>org.jboss.test.ws.jaxws.samples.wsse.policy.basic.KeystorePasswordCallback</property-
value>
    </property>
  </endpoint-config>
</jaxws-config>
```

JBoss EAP default configuration modified to default to SOAP messages schema-validation on:

```
<subsystem xmlns="urn:jboss:domain:webservices:2.0">
    <!-- ... -->
    <endpoint-config name="Standard-Endpoint-Config">
        <property name="schema-validation-enabled" value="true"/>
    </endpoint-config>
    <!-- ... -->
    <client-config name="Standard-Client-Config">
        <property name="schema-validation-enabled" value="true"/>
    </client-config>
</subsystem>
```

3.5.1.3. EndpointConfig Annotation

Once a configuration is available to a given application, the **org.jboss.ws.api.annotation.EndpointConfig** annotation is used to assign an endpoint configuration to a Jakarta XML Web Services endpoint implementation. When you assign a configuration that is defined in the **webservices** subsystem, you only need to specify the configuration name. When you assign a configuration that is defined in the application, you need to specify the relative path to the deployment descriptor and the configuration name.

Example: EndpointConfig Annotation

```
@EndpointConfig(configFile = "WEB-INF/my-endpoint-config.xml", configName = "Custom WS-
Security Endpoint")
public class ServiceImpl implements ServiceIface {
  public String sayHello() {
```

```
        return "Secure Hello World!";
    }
}
```

3.5.1.4. Jakarta XML Web Services Feature

You can also use **org.jboss.ws.api.configuration.ClientConfigFeature** to set a configuration that is a Jakarta XML Web Services Feature extension provided by JBossWS.

```
import org.jboss.ws.api.configuration.ClientConfigFeature;

Service service = Service.create(wsdlURL, serviceName);

Endpoint port = service.getPort(Endpoint.class, new ClientConfigFeature("META-INF/my-client-config.xml", "Custom Client Config"));
port.echo("Kermit");
```

You can also set properties from the specified configuration by passing in **true** to the **ClientConfigFeature** constructor.

```
Endpoint port = service.getPort(Endpoint.class, new ClientConfigFeature("META-INF/my-client-config.xml", "Custom Client Config"), true);
```

JBossWS parses the specified configuration file, after having resolved it as a resource using the current thread context class loader. The ***EAP_HOME*/docs/schema/jbossws-jaxws-config_4_0.xsd** schema defines the descriptor contents and is included in the **jbossws-spi** artifact.

If you pass in **null** for the configuration file, the configuration will be read from the current container configurations, if available.

```
Endpoint port = service.getPort(Endpoint.class, new ClientConfigFeature(null, "Container Custom Client Config"));
```

3.5.1.5. Explicit Setup Through API

Alternatively, the JBossWS API comes with facility classes that can be used for assigning configurations when building a client.

Handlers
Jakarta XML Web Services handlers are read from client configurations as follows.

```
import org.jboss.ws.api.configuration.ClientConfigUtil;
import org.jboss.ws.api.configuration.ClientConfigurer;

Service service = Service.create(wsdlURL, serviceName);
Endpoint port = service.getPort(Endpoint.class);
BindingProvider bp = (BindingProvider)port;

ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigHandlers(bp, "META-INF/my-client-config.xml", "Custom Client Config");
port.echo("Kermit");
```

You can also use the **ClientConfigUtil** utility class to set up the handlers.

-

```
ClientConfigUtil.setConfigHandlers(bp, "META-INF/my-client-config.xml", "Custom Client Config");
```

The default **ClientConfigurer** implementation parses the specified configuration file, after having resolved it as a resource using the current thread context class loader. The **EAP_HOME/docs/schema/jbossws-jaxws-config_4_0.xsd** schema defines the descriptor contents and is included in the **jbossws-spi** artifact.

If you pass in **null** for the configuration file, the configuration will be read from the current container configurations, if available.

```
ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigHandlers(bp, null, "Container Custom Client Config");
```

Properties
Similarly, properties are read from client configurations as follows.

```
import org.jboss.ws.api.configuration.ClientConfigUtil;
import org.jboss.ws.api.configuration.ClientConfigurer;

Service service = Service.create(wsdlURL, serviceName);
Endpoint port = service.getPort(Endpoint.class);

ClientConfigUtil.setConfigProperties(port, "META-INF/my-client-config.xml", "Custom Client Config");
port.echo("Kermit");
```

You can also use the **ClientConfigUtil** utility class to set up the properties.

```
ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigProperties(port, "META-INF/my-client-config.xml", "Custom Client Config");
```

The default **ClientConfigurer** implementation parses the specified configuration file, after having resolved it as a resource using the current thread context class loader. The **EAP_HOME/docs/schema/jbossws-jaxws-config_4_0.xsd** schema defines the descriptor contents and is included in the **jbossws-spi** artifact.

If you pass in **null** for the configuration file, the configuration will be read from the current container configurations, if available.

```
ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigProperties(port, null, "Container Custom Client Config");
```

3.5.2. Automatic Configuration from Default Descriptors

In some cases, the application developer might not be aware of the configuration that will need to be used for its client and endpoint implementation. In other cases, explicit usage of the JBossWS API might not be accepted because it is a compile-time dependency. To cope with such scenarios, JBossWS allows including default client, **jaxws-client-config.xml**, and endpoint, **jaxws-endpoint-config.xml**, descriptors within the application in its root directory. These are parsed for getting configurations whenever a configuration file name is not specified.

```
<config-file>WEB-INF/jaxws-endpoint-config.xml</config-file>
```

If the configuration name is not specified, JBossWS automatically looks for a configuration named as:

- The fully qualified name (FQN) of the endpoint implementation class, in case of Jakarta XML Web Services endpoints.
- The FQN of the service endpoint interface, in case of Jakarta XML Web Services clients.

No automatic configuration name is selected for **Dispatch** clients.

For example, an endpoint implementation class **org.foo.bar.EndpointImpl**, for which no predefined configuration is explicitly set, will cause JBossWS to look for a **org.foo.bar.EndpointImpl** named configuration within a **jaxws-endpoint-config.xml** descriptor in the root of the application deployment. Similarly, on the client side, a client proxy implementing **org.foo.bar.Endpoint** interface will have the setup read from a **org.foo.bar.Endpoint** named configuration in the **jaxws-client-config.xml** descriptor.

3.5.3. Automatic Configuration Assignment from Container

JBossWS falls back to getting predefined configurations from the container whenever no explicit configuration has been provided and the default descriptors are either not available or do not contain relevant configurations. This behavior gives additional control on the Jakarta XML Web Services client and endpoint setup to administrators since the container can be managed independently from the deployed applications.

JBossWS accesses the **webservices** subsystem for an explicitly named configuration. The default configuration names used are:

- The fully qualified name of the endpoint implementation class, in case of Jakarta XML Web Services endpoints.
- The fully qualified name of the service endpoint interface, in case of Jakarta XML Web Services clients.

Dispatch clients are not automatically configured. If no configuration is found using names computed as above, the **Standard-Client-Config** and **Standard-Endpoint-Config** configurations are used for clients and endpoints respectively.

3.6. SETTING MODULE DEPENDENCIES FOR WEB SERVICE APPLICATIONS

JBoss EAP web services are delivered as a set of modules and libraries, including the **org.jboss.as.webservices.*** and **org.jboss.ws.*** modules. You should not need to change these modules.

With JBoss EAP you cannot directly use JBossWS implementation classes unless you explicitly set a dependency to the corresponding module. You declare the module dependencies that you want to be added to the deployment.

The JBossWS APIs are available by default whenever the **webservices** subsystem is available. You can use them without creating an explicit dependencies declaration for those modules.

3.6.1. Using MANIFEST.MF

To configure deployment dependencies, add them to the **MANIFEST.MF** file. For example:

```
Manifest-Version: 1.0
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client services export,foo.bar
```

This **MANIFEST.MF** file declares dependencies on the **org.jboss.ws.cxf.jbossws-cxf-client** and **foo.bar** modules. For more information on declaring dependencies in a **MANIFEST.MF** file, including the **export** and **services** options, see Add a Dependency Configuration to MANIFEST.MF in the JBoss EAP *Development Guide*.

When using annotations on the endpoints and handlers, for example, Apache CXF endpoints and handlers, add the proper module dependency in your manifest file. If you skip this step, your annotations are not picked up and are completely, silently ignored.

3.6.1.1. Using Jakarta XML Binding

To successfully and directly use Jakarta XML Binding contexts in your client or endpoint running in-container, set up a Jakarta XML Binding implementation. For example, set the following dependency:

```
Dependencies: com.sun.xml.bind services export
```

3.6.1.2. Using Apache CXF

To use Apache CXF APIs and implementation classes, add a dependency to the **org.apache.cxf** (API) module or **org.apache.cxf.impl** (implementation) module. For example:

```
Dependencies: org.apache.cxf services
```

The dependency is purely Apache CXF without any JBossWS customizations or additional extensions. For this reason, a client-side aggregation module is available with all the web service dependencies that you might need.

3.6.1.3. Client-side Web Services Aggregation Module

When you want to use all of the web services features and functionality, you can set a dependency to the convenient client module. For example:

```
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client services
```

The **services** option is required to enable all JBossWS features by loading JBossWS specific classes. The **services** option is almost always needed when declaring dependencies on the **org.jboss.ws.cxf.jbossws-cxf-client** and **org.apache.cxf** modules. The option affects the loading of classes through the **Service** API, which is what is used to wire most of the JBossWS components and the Apache CXF Bus extensions.

3.6.1.4. Annotation Scanning

The application server uses an annotation index for detecting Jakarta XML Web Services endpoints in user deployments. When declaring web service endpoints for a class that belongs to a different module, for instance referring to it in the **web.xml** descriptor, use an **annotations** type dependency. Without that dependency your endpoints are ignored as they do not appear as annotated classes to the **webservices** subsystem.

```
Dependencies: my.org annotations
```

3.6.2. Using jboss-deployment-structure.xml

In some circumstances, the convenient approach of setting module dependencies in the **MANIFEST.MF** file might not work. For example, setting dependencies in the **MANIFEST.MF** file does not work when importing and exporting specific resources from a given module dependency. In these scenarios, add a **jboss-deployment-structure.xml** descriptor file to your deployment and set module dependencies in it.

For more information on using **jboss-deployment-structure.xml**, see Add a Dependency Configuration to the jboss-deployment-structure.xml in the JBoss EAP *Development Guide*.

3.7. CONFIGURING HTTP TIMEOUT

The HTTP session timeout defines the period after which an HTTP session is considered to have become invalid because there was no activity within the specified period.

The HTTP session timeout can be configured, in order of precedence, in the following places:

1. Application
   You can define the HTTP session timeout in the application's **web.xml** configuration file by adding the following configuration to the file. This value is in minutes.

   ```
   <session-config>
     <session-timeout>30</session-timeout>
   </session-config>
   ```

   If you modified the WAR file, redeploy the application. If you exploded the WAR file, no further action is required because JBoss EAP automatically undeploys and redeploys the application.

2. Server
   You can use the following management CLI command to set the default HTTP session timeout in the **undertow** subsystem. This value is in minutes.

   ```
   /subsystem=undertow/servlet-container=default:write-attribute(name=default-session-timeout,value=30)
   ```

3. Default
   The default HTTP session timeout is 30 minutes.

3.8. SECURING JAKARTA XML WEB SERVICES

WS-Security provides the means to secure your services beyond transport level protocols such as HTTPS. Through a number of standards, such as headers defined in the WS-Security standard, you can:

- Pass authentication tokens between services.
- Encrypt messages or parts of messages.
- Sign messages.
- Timestamp messages.

WS-Security makes use of public and private key cryptography. With public key cryptography, a user has a pair of public and private keys. These are generated using a large prime number and a key function.

The keys are related mathematically, but cannot be derived from one another. With these keys we can encrypt messages. For example, if Scott wants to send a message to Adam, he can encrypt a message

using his public key. Adam can then decrypt this message using his private key. Only Adam can decrypt this message as he is the only one with the private key.

Messages can also be signed. This allows you to ensure the authenticity of the message. If Adam wants to send a message to Scott, and Scott wants to be sure that it is from Adam, Adam can sign the message using his private key. Scott can then verify that the message is from Adam by using his public key.

3.8.1. Applying Web Services Security (WS-Security)

Web services support many real-world scenarios requiring WS-Security functionality. These scenarios include signature and encryption support through X509 certificates, authentication and authorization through username tokens, and all WS-Security configurations covered by the WS-SecurityPolicy specification.

For other WS-* features, the core of WS-Security functionality is provided through the Apache CXF engine. In addition, the JBossWS integration adds a few configuration enhancements to simplify the setup of WS-Security enabled endpoints.

3.8.1.1. Apache CXF WS-Security Implementation

Apache CXF features a WS-Security module that supports multiple configurations and is easily extendible.

The system is based on *interceptors* that delegate to Apache WSS4J for the low-level security operations. Interceptors can be configured in different ways, either through Spring configuration files or directly using the Apache CXF client API.

Recent versions of Apache CXF introduced support for WS-SecurityPolicy, which aims at moving most of the security configuration into the service contract, through policies, so that clients can be easily configured almost completely automatically from that. This way users do not need to manually deal with configuring and installing the required interceptors; the Apache CXF WS-Policy engine internally takes care of that instead.

3.8.1.2. WS-Security Policy Support

WS-SecurityPolicy describes the actions that are required to securely communicate with a service advertised in a given WSDL contract. The WSDL bindings and operations reference WS-Policy fragments with the security requirements to interact with the service. The WS-SecurityPolicy specification allows for specifying things such as asymmetric and symmetric keys, using transports (HTTPS) for encryption, which parts or headers to encrypt or sign, whether to sign then encrypt or encrypt then sign, whether to include timestamps, whether to use derived keys, or something else.

However some mandatory configuration elements are not covered by WS-SecurityPolicy because they are not meant to be public or part of the published endpoint contract. These include things such as keystore locations, and usernames and passwords. Apache CXF allows configuring these elements either through Spring XML descriptors or using the client API or annotations.

Table 3.4. Supported Configuration Properties

Configuration property	Description
ws-security.username	The username used for **UsernameToken** policy assertions.

Configuration property	Description
ws-security.password	The password used for **UsernameToken** policy assertions. If not specified, the callback handler will be called.
ws-security.callback-handler	The WSS4J security **CallbackHandler** that will be used to retrieve passwords for keystores and **UsernameToken**.
ws-security.signature.properties	The properties file/object that contains the WSS4J properties for configuring the signature keystore and crypto objects.
ws-security.encryption.properties	The properties file/object that contains the WSS4J properties for configuring the encryption keystore and crypto objects.
ws-security.signature.username	The username or alias for the key in the signature keystore that will be used. If not specified, it uses the default alias set in the properties file. If that is also not set, and the keystore only contains a single key, that key will be used.
ws-security.encryption.username	The username or alias for the key in the encryption keystore that will be used. If not specified, it uses the default alias set in the properties file. If that is also not set, and the keystore only contains a single key, that key will be used. For the web service provider, the **useReqSigCert** keyword can be used to accept (encrypt) any client whose public key is in the service's truststore (defined in **ws-security.encryption.properties**).
ws-security.signature.crypto	Instead of specifying the signature properties, this can point to the full WSS4J **Crypto** object. This can allow easier programmatic configuration of the crypto information.
ws-security.encryption.crypto	Instead of specifying the encryption properties, this can point to the full WSS4J **Crypto** object. This can allow easier programmatic configuration of the crypto information.
ws-security.enable.streaming	Enable streaming (StAX based) processing of WS-Security messages.

3.8.2. WS-Trust

WS-Trust is a web service specification that defines extensions to WS-Security. It is a general framework for implementing security in a distributed system. The standard is based on a centralized Security Token

Service (STS), which is capable of authenticating clients and issuing tokens containing various types of authentication and authorization data. The specification describes a protocol used for issuance, exchange, and validation of security tokens. The following specifications play an important role in the WS-Trust architecture:

- WS-SecurityPolicy 1.2
- SAML 2.0
- Username Token Profile
- X.509 Token Profile
- SAML Token Profile
- Kerberos Token Profile

The WS-Trust extensions address the needs of applications that span multiple domains and requires the sharing of security keys. This occurs by providing a standards-based trusted third party web service (STS) to broker trust relationships between a web service requester and a web service provider. This architecture also alleviates the pain of service updates that require credential changes by providing a common location for this information. The STS is the common access point from which both the requester and provider retrieves and verifies security tokens.

There are three main components of the WS-Trust specification:

- The Security Token Service (STS) for issuing, renewing, and validating security tokens.
- The message formats for security token requests and responses.
- The mechanisms for key exchange.

The following section explains a basic WS-Trust scenario. For advanced scenarios, see Advanced WS-Trust Scenarios.

3.8.2.1. Scenario: Basic WS-Trust

In this section we provide an example of a basic WS-Trust scenario. It comprises a web service requester (**ws-requester**), a web service provider (**ws-provider**), and a Security Token Service (STS).

The **ws-provider** requires SAML 2.0 token issued from a designated STS to be presented by the **ws-requester** using asymmetric binding. These communication requirements are declared in the WSDL of the **ws-provider**. STS requires **ws-requester** credentials to be provided in a WSS UsernameToken format request using symmetric binding. The response from STS is provided containing SAML 2.0 token. These communication requirements are declared in the WSDL of the STS.

1. The **ws-requester** contacts the **ws-provider** and consumes its WSDL. On finding the security token issuer requirement, the **ws-requester** creates and configures the **STSClient** with the information required to generate a valid request.
2. The **STSClient** contacts the STS and consumes its WSDL. The security policies are discovered. The **STSClient** creates and sends an authentication request with appropriate credentials.
3. The STS verifies the credentials.
4. In response, the STS issues a security token that provides proof that the **ws-requester** has authenticated with the STS.

5. The **STSClient** presents a message with the security token to the **ws-provider**.
6. The **ws-provider** verifies that the token was issued by the STS, and hence proves that the **ws-requester** has successfully authenticated with the STS.
7. The **ws-provider** executes the requested service and returns the results to the **ws-requester**.

3.8.2.2. Apache CXF Support

Apache CXF is an open-source, fully-featured web services framework. The JBossWS open source project integrates the JBoss Web Services (JBossWS) stack with the Apache CXF project modules to provide WS-Trust and other Jakarta XML Web Services functionality. This integration helps in easy deployment of Apache CXF STS implementations. The Apache CXF API also provides a **STSClient** utility to facilitate web service requester communication with its STS.

3.8.3. Security Token Service (STS)

The Security Token Service (STS) is the core of the WS-Trust specification. It is a standards-based mechanism for authentication and authorization. The STS is an implementation of the WS-Trust specification's protocol for issuing, exchanging, and validating security tokens, based on token format, namespace, or trust boundaries. The STS is a web service that acts as a trusted third party to broker trust relationships between a web service requester and a web service provider. It is a common access point trusted by both requester and provider to provide interoperable security tokens. It removes the need for a direct relationship between the requestor and provider. The STS helps ensure interoperability across realms and between different platforms because it is a standards-based mechanism for authentication.

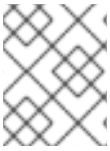
The STS's WSDL contract defines how other applications and processes interact with it. In particular, the WSDL defines the WS-Trust and WS-Security policies that a requester must fulfill to successfully communicate with the STS's endpoints. A web service requester consumes the STS's WSDL and, with the aid of an **STSClient** utility, generates a message request compliant with the stated security policies and submits it to the STS endpoint. The STS validates the request and returns an appropriate response.

3.8.3.1. Configuring a PicketLink WS-Trust Security Token Service (STS)

PicketLink STS provides options for building an alternative to the Apache CXF Security Token Service implementation. You can also use PicketLink to configure SAML SSO for web applications. For more details on configuring SAML SSO using PicketLink, see *How To Set Up SSO with SAML v2* .

To set up an application to serve as a PicketLink WS-Trust STS, the following steps must be performed:

1. Create a security domain for the WS-Trust STS application.
2. Configure the **web.xml** file for the WS-Trust STS application.
3. Configure the authenticator for the WS-Trust STS application.
4. Declare the necessary dependencies for the WS-Trust STS application.
5. Configure the web-service portion of the WS-Trust STS application.
6. Create and configure a **picketlink.xml** file for the WS-Trust STS application.



NOTE

The security domain should be created and configured before creating and deploying the application.

3.8.3.1.1. Create a Security Domain for the STS

The STS handles authentication of a principal based on the credentials provided and issues the proper security token based on that result. This requires that an identity store be configured via a security domain. The only requirement around creating this security domain and identity store is that it has authentication and authorization mechanisms properly defined. This means that many different identity stores, for example properties files, database, and LDAP, and their associated login modules could be used to support an STS application. For more information on security domains, see the *Security Domains* section of the JBoss EAP *Security Architecture* documentation.

In the below example, a simple **UsersRoles** login module using properties files for an identity store is used.

CLI Commands for Creating a Security Domain

```
/subsystem=security/security-domain=sts:add(cache-type=default)
```

```
/subsystem=security/security-domain=sts/authentication=classic:add
```

```
/subsystem=security/security-domain=sts/authentication=classic/login-
module=UsersRoles:add(code=UsersRoles,flag=required,module-options=
[usersProperties=${jboss.server.config.dir}/sts-
users.properties,rolesProperties=${jboss.server.config.dir}/sts-roles.properties])
```

```
reload
```

Resulting XML

```
<security-domain name="sts" cache-type="default">
  <authentication>
    <login-module code="UsersRoles" flag="required">
      <module-option name="usersProperties" value="${jboss.server.config.dir}/sts-users.properties"/>
      <module-option name="rolesProperties" value="${jboss.server.config.dir}/sts-roles.properties"/>
    </login-module>
  </authentication>
</security-domain>
```

NOTE

The management CLI commands shown assume that you are running a JBoss EAP standalone server. For more details on using the management CLI for a JBoss EAP managed domain, see the JBoss EAP *Management CLI Guide*.

Property Files

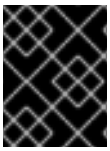
The **UsersRoles** login module utilizes properties files to store the user/password and user/role information. For more specifics of the **UsersRoles** module, please see the JBoss EAP *Login Module Reference*. In this example, the properties files contain the following:

Example: sts-users.properties File

```
Eric=samplePass
Alan=samplePass
```

Example: sts-roles.properties File

```
Eric=All
Alan=
```



IMPORTANT

You also need to create a keystore for signing and encrypting the security tokens. This keystore will be used when configuring the **picketlink.xml** file.

3.8.3.1.2. Configure the web.xml File for the STS

The **web.xml** file for an STS should contain the following:

- A **<servlet>** to enable the STS functionality and a **<servlet-mapping>** to map its URL.
- A **<security-constraint>** with a **<web-resource-collection>** containing a **<url-pattern>** that maps to the URL pattern of the secured area. Optionally, **<security-constraint>** may also contain an **<auth-constraint>** stipulating the allowed roles.
- A **<login-config>** configured for BASIC authentication.
- If any roles were specified in the **<auth-constraint>**, those roles should be defined in a **<security-role>**.

Example web.xml file:

```
<web-app>
  <!-- Define STS servlet -->
  <servlet>
    <servlet-name>STS-servlet</servlet-name>
    <servlet-class>com.example.sts.PicketLinkSTService</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>STS-servlet</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
  <!-- Define a security constraint that requires the All role to access resources -->
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>STS</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>All</role-name>
    </auth-constraint>
  </security-constraint>
  <!-- Define the Login Configuration for this Application -->
  <login-config>
```

```
    <auth-method>BASIC</auth-method>
    <realm-name>STS Realm</realm-name>
  </login-config>
  <!-- Security roles referenced by this web application -->
  <security-role>
    <description>The role that is required to log in to the IDP Application</description>
    <role-name>All</role-name>
  </security-role>
</web-app>
```

3.8.3.1.3. Configure the Authenticator for the STS

The authenticator is responsible for the authentication of users for issuing and validating security tokens. The authenticator is configured by defining the security domain to be used in authenticating and authorizing principals.

The **jboss-web.xml** file should have the following:

- A **<security-domain>** to specify which security domain to use for authentication and authorization.

Example: jboss-web.xml File

```
<jboss-web>
  <security-domain>sts</security-domain>
  <context-root>SecureTokenService</context-root>
</jboss-web>
```

3.8.3.1.4. Declare the Necessary Dependencies for the STS

The web application serving as the STS requires a dependency to be defined in the **jboss-deployment-structure.xml** file so that the **org.picketlink** classes can be located. As JBoss EAP provides all necessary **org.picketlink** and related classes, the application just needs to declare them as dependencies to use them.

Example: Using jboss-deployment-structure.xml to Declare Dependencies

```
<jboss-deployment-structure>
  <deployment>
    <dependencies>
      <module name="org.picketlink"/>
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

3.8.3.1.5. Configure the Web-Service Portion of the STS

The web application serving as the STS requires that you define a web-service for clients to call to obtain their security tokens. This requires that you define in your WSDL a service name called **PicketLinkSTS**, and a port called **PicketLinkSTSPort**. You can, however, change the SOAP address to better reflect your target deployment environment.

Example: PicketLinkSTS.wsdl File

```
<?xml version="1.0"?>
<wsdl:definitions name="PicketLinkSTS" targetNamespace="urn:picketlink:identity-federation:sts"
 xmlns:tns="urn:picketlink:identity-federation:sts"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:wsap10="http://www.w3.org/2006/05/addressing/wsdl"
 xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">
  <wsdl:types>
    <xs:schema targetNamespace="urn:picketlink:identity-federation:sts"
      xmlns:tns="urn:picketlink:identity-federation:sts"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      version="1.0" elementFormDefault="qualified">
      <xs:element name="MessageBody">
        <xs:complexType>
          <xs:sequence>
            <xs:any minOccurs="0" maxOccurs="unbounded" namespace="##any"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </wsdl:types>
  <wsdl:message name="RequestSecurityToken">
    <wsdl:part name="rstMessage" element="tns:MessageBody"/>
  </wsdl:message>
  <wsdl:message name="RequestSecurityTokenResponse">
    <wsdl:part name="rstrMessage" element="tns:MessageBody"/>
  </wsdl:message>
  <wsdl:portType name="SecureTokenService">
    <wsdl:operation name="IssueToken">
      <wsdl:input wsap10:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
message="tns:RequestSecurityToken"/>
      <wsdl:output wsap10:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Issue"
message="tns:RequestSecurityTokenResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="STSBinding" type="tns:SecureTokenService">
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="IssueToken">
      <soap12:operation soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
style="document"/>
      <wsdl:input>
        <soap12:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="PicketLinkSTS">
    <wsdl:port name="PicketLinkSTSPort" binding="tns:STSBinding">
      <soap12:address location="http://localhost:8080/SecureTokenService/PicketLinkSTS"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

In addition, you need a class for your web-service to use your WSDL:

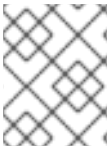
Example: PicketLinkSTS Class

```
@WebServiceProvider(serviceName = "PicketLinkSTS", portName = "PicketLinkSTSPort",
targetNamespace = "urn:picketlink:identity-federation:sts", wsdlLocation = "WEB-
INF/wsdl/PicketLinkSTS.wsdl")
@ServiceMode(value = Service.Mode.MESSAGE)
public class PicketLinkSTService extends PicketLinkSTS {
    private static Logger log = Logger.getLogger(PicketLinkSTService.class.getName());

    @Resource
    public void setWSC(WebServiceContext wctx) {
        log.debug("Setting WebServiceContext = " + wctx);
        this.context = wctx;
    }
}
```

3.8.3.1.6. Create and Configure a picketlink.xml File for the STS

The **picketlink.xml** file is responsible for the behavior of the authenticator and is loaded at the application's startup.

The JBoss EAP Security Token Service defines several interfaces that provide extension points. Implementations can be plugged in and the default values can be specified for some properties using configuration. Similar to the IDP and SP configuration in *How To Set Up SSO with SAML v2* , all STS configurations are specified in the **picketlink.xml** file of the deployed application. The following are the elements that can be configured in the **picketlink.xml** file.



NOTE

In the following text, a service provider refers to the web service that requires a security token to be presented by its clients.

- **<PicketLinkSTS>**: This is the root element. It defines some properties that allow the STS administrator to set the following properties:
    - **STSName**: A string representing the name of the security token service. If not specified, the default **PicketLinkSTS** value is used.
    - **TokenTimeout**: The token lifetime value in seconds. If not specified, the default value of **3600** (one hour) is used.
    - **EncryptToken**: A boolean specifying whether issued tokens are to be encrypted or not. The default value is **false**.
- **<KeyProvider>**: This element and all its subelements are used to configure the keystore that are used by PicketLink STS to sign and encrypt tokens. Properties like the keystore location, its password, and the signing (private key) alias and password are all configured in this section.
- **<TokenProviders>**: This section specifies the **TokenProvider** implementations that must be used to handle each type of security token. In the example we have two providers - one that handles tokens of type **SAMLV1.1** and one that handles tokens of type **SAMLV2.0**. The **WSTrustRequestHandler** calls the **getProviderForTokenType(String type)** method of **STSConfiguration** to obtain a reference to the appropriate **TokenProvider**.
- **<ServiceProviders>**: This section specifies the token types that must be used for each service

provider, the web service that requires a security token. When a WS-Trust request does not contain the token type, the **WSTrustRequestHandler** must use the service provider endpoint to find out the type of the token that must be issued.

NOTE

When configuring PicketLink, POST binding is recommended as it provides enhanced security and does not pass the response within URL parameters.

Example: picketlink.xml Configuration File

```
<!DOCTYPE PicketLinkSTS>
<PicketLinkSTS xmlns="urn:picketlink:federation:config:2.1"
               STSName="PicketLinkSTS" TokenTimeout="7200" EncryptToken="false">
   <KeyProvider
         ClassName="org.picketlink.identity.federation.core.impl.KeyStoreKeyManager">
      <Auth Key="KeyStoreURL" Value="sts_keystore.jks"/>
      <Auth Key="KeyStorePass" Value="testpass"/>
      <Auth Key="SigningKeyAlias" Value="sts"/>
      <Auth Key="SigningKeyPass" Value="keypass"/>
      <ValidatingAlias Key="http://services.testcorp.org/provider1"
                  Value="service1"/>
   </KeyProvider>
   <TokenProviders>
      <TokenProvider

ProviderClass="org.picketlink.identity.federation.core.wstrust.plugins.saml.SAML11TokenProvider"
               TokenType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1"
               TokenElement="Assertion" TokenElementNS="urn:oasis:names:tc:SAML:1.0:assertion"/>
      <TokenProvider

ProviderClass="org.picketlink.identity.federation.core.wstrust.plugins.saml.SAML20TokenProvider"
               TokenType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0"
               TokenElement="Assertion" TokenElementNS="urn:oasis:names:tc:SAML:2.0:assertion"/>
   </TokenProviders>
   <ServiceProviders>
      <ServiceProvider Endpoint="http://services.testcorp.org/provider1"
                  TokenType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV2.0"
                  TruststoreAlias="service1"/>
   </ServiceProviders>
</PicketLinkSTS>
```

By default, the **picketlink.xml** file is located in the **WEB-INF/classes** directory of the STS web application. The PicketLink configuration file can also be loaded from the file system. To load the PicketLink configuration file from the file system, it must be named **picketlink-sts.xml** and be located in the **${user.home}/picketlink-store/sts/** directory.

3.8.3.2. Using a WS-Trust Security Token Service (STS) with a Client

To configure a client to obtain a security token from the STS, you need to make use of the **org.picketlink.identity.federation.api.wstrust.WSTrustClient** class to connect to the STS and ask for a token to be issued.

First you need to instantiate the client:

Example: Creating a WSTrustClient

```
WSTrustClient client = new WSTrustClient("PicketLinkSTS", "PicketLinkSTSPort",
    "http://localhost:8080/SecureTokenService/PicketLinkSTS",
    new SecurityInfo(username, password));
```

Next you need to use the **WSTrustClient** to ask for a token, for example a SAML assertion, to be issued:

Example: Obtaining an Assertion

```
org.w3c.dom.Element assertion = null;
try {
  assertion = client.issueToken(SAMLUtil.SAML2_TOKEN_TYPE);
} catch (WSTrustException wse) {
  System.out.println("Unable to issue assertion: " + wse.getMessage());
  wse.printStackTrace();
}
```

Once you have the assertion, there are two ways by which it can be included in and sent via the SOAP message:

- The client can push the SAML2 **Assertion** into the SOAP **MessageContext** under the key **org.picketlink.trust.saml.assertion**. For example:

  ```
  bindingProvider.getRequestContext().put(SAML2Constants.SAML2_ASSERTION_PROPERTY, assertion);
  ```

- The SAML2 **Assertion** is available as part of the JAAS subject on the security context. This can happen if there has been a JAAS interaction with the usage of PicketLink STS login modules.

3.8.3.3. STS Client Pooling

WARNING

The STS client pooling feature is *NOT* supported in JBoss EAP.

STS client pooling is a feature that allows you to configure a pool of STS clients on the server, thereby eliminating a possible bottleneck of STS client creation. Client pooling can be used for login modules that need an STS client to obtain SAML tickets. These include:

- **org.picketlink.identity.federation.core.wstrust.auth.STSIssuingLoginModule**
- **org.picketlink.identity.federation.core.wstrust.auth.STSValidatingLoginModule**
- **org.picketlink.trust.jbossws.jaas.JBWSTokenIssuingLoginModule**

The default number of clients in the pool for each login module is configured using the **initialNumberOfClients** login module option.

The **org.picketlink.identity.federation.bindings.stspool.STSClientPoolFactory** class provides client pool functionality to applications.

Using STSClientPoolFactory
STS clients are inserted into subpools using their STSClientConfig configuration as a key. To insert an STS client into a subpool, you need to obtain the **STSClientPool** instance and then initialize a subpool based on the configuration. Optionally, you can specify the initial number of STS clients when initializing the pool, or you can rely on the default number.

Example: Inserting an STS Client into a Subpool

```
final STSClientPool pool = STSClientPoolFactory.getPoolInstance();
pool.createPool(20, stsClientConfig);
final STSClient client = pool.getClient(stsClientConfig);
```

When you are done with a client, you can return it to the pool by calling the **returnClient()** method.

Example: Returning an STS Client to the Subpool

```
pool.returnClient();
```

Example: Checking If a Subpool Exists with a Given Configuration

```
if (! pool.configExists(stsClientConfig) {
    pool.createPool(stsClientConfig);
}
```

If the **picketlink-federation** subsystem is enabled, all client pools created for a deployment are destroyed automatically during the undeploy process. To manually destroy a pool:

Example: Manually Destroying a Subpool

```
pool.destroyPool(stsClientConfig);
```

3.8.4. Propagating Authenticated Identity to the Jakarta Enterprise Beans Subsystem

The **webservices** subsystem contains an adapter that allows you to configure an Elytron security domain to secure web service endpoints using either annotations or deployment descriptors.

When Elytron security is enabled, the JAAS subject or principal can be pushed to the Apache CXF endpoint's **SecurityContext** to propagate the authenticated identity to the Jakarta Enterprise Beans container.

The following is an example of how to use an Apache CXF interceptor to propagate authenticated information to the Jakarta Enterprise Beans container.

```
public class PropagateSecurityInterceptor extends WSS4JInInterceptor {
   public PropagateSecurityInterceptor() {
     super();
```

```
    getAfter().add(PolicyBasedWSS4JInInterceptor.class.getName());
  }
  @Override
  public void handleMessage(SoapMessage message) throws Fault {
    ...
    final Endpoint endpoint = message.getExchange().get(Endpoint.class);
    final SecurityDomainContext securityDomainContext = endpoint.getSecurityDomainContext();
    //push subject principal retrieved from CXF to ElytronSecurityDomainContext
    securityDomainContext.pushSubjectContext(subject, principal, null)
    }
  }
```

3.9. JAKARTA XML WEB SERVICES LOGGING

You can handle logging for inbound and outbound messages using Jakarta XML Web Services handlers or Apache CXF logging interceptors.

3.9.1. Using Jakarta XML Web Services Handlers

You can configure a Jakarta XML Web Services handler to log messages that are passed to it. This approach is portable as the handler can be added to the desired client and endpoints programatically by using the **@HandlerChain** Jakarta XML Web Services annotation.

The predefined client and endpoint configuration mechanism allows you to add the logging handler to any client and endpoint combination, or to only some of the clients and endpoints. To add the logging handler to only some of the clients or endpoints, use the **@EndpointConfig** annotation and the JBossWS API.

The **org.jboss.ws.api.annotation.EndpointConfig** annotation is used to assign an endpoint configuration to a Jakarta XML Web Services endpoint implementation. When assigning a configuration that is defined in the **webservices** subsystem, only the configuration name is specified. When assigning a configuration that is defined in the application, the relative path to the deployment descriptor and the configuration name must be specified.

3.9.2. Using Apache CXF Logging Interceptors

Apache CXF also comes with logging interceptors that can be used to log messages to the console, client log files, or server log files. Those interceptors can be added to client, endpoint, and buses in multiple ways, including:

- System property
  Setting the **org.apache.cxf.logging.enabled** system property to **true** causes the logging interceptors to be added to any bus instance being created on the JVM. You can also set the system property to **pretty** to output nicely-formatted XML output. You can use the following management CLI command to set this system property.

  ```
  /system-property=org.apache.cxf.logging.enabled:add(value=true)
  ```

- Manual interceptor addition
  Logging interceptors can be selectively added to endpoints using the Apache CXF annotations **@org.apache.cxf.interceptor.InInterceptors** and **@org.apache.cxf.interceptor.OutInterceptors**. The same outcome is achieved on the client side by programmatically adding new instances of the logging interceptors to the client or the bus.

3.10. ENABLING WEB SERVICES ADDRESSING (WS-ADDRESSING)

Web Services Addressing, or WS-Addressing, provides a transport-neutral mechanism to address web services and their associated messages. To enable WS-Addressing, you must add the **@Addressing** annotation to the web service endpoint and then configure the client to access it.

The following examples assume your application has an existing Jakarta XML Web Services service and client configuration. See the **jaxws-addressing** quickstart that ships with JBoss EAP for a complete working example.

1. Add the **@Addressing** annotation to the application's Jakarta XML Web Services endpoint code.

   Example: Jakarta XML Web Services Endpoint with @Addressing Annotation

```
package org.jboss.quickstarts.ws.jaxws.samples.wsa;

import org.jboss.quickstarts.ws.jaxws.samples.wsa.ServiceIface;

import javax.jws.WebService;
import javax.xml.ws.soap.Addressing;

@WebService(
    portName = "AddressingServicePort",
    serviceName = "AddressingService",
    wsdlLocation = "WEB-INF/wsdl/AddressingService.wsdl",
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wsaddressing",
    endpointInterface = "org.jboss.quickstarts.ws.jaxws.samples.wsa.ServiceIface")
@Addressing(enabled = true, required = true)
public class ServiceImpl implements ServiceIface {
    public String sayHello() {
        return "Hello World!";
    }
}
```

2. Update the Jakarta XML Web Services client code to configure WS-Addressing.

   Example: Jakarta XML Web Services Client Configured for WS-Addressing

```
package org.jboss.quickstarts.ws.jaxws.samples.wsa;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.soap.AddressingFeature;

public final class AddressingClient {
    private static final String serviceURL =
        "http://localhost:8080/jaxws-addressing/AddressingService";

    public static void main(String[] args) throws Exception {
        // construct proxy
        QName serviceName =
            new QName("http://www.jboss.org/jbossws/ws-extensions/wsaddressing",
                "AddressingService");
```

```
        URL wsdlURL = new URL(serviceURL + "?wsdl");
        Service service = Service.create(wsdlURL, serviceName);
        org.jboss.quickstarts.ws.jaxws.samples.wsa.ServiceIface proxy =
            (org.jboss.quickstarts.ws.jaxws.samples.wsa.ServiceIface)
service.getPort(org.jboss.quickstarts.ws.jaxws.samples.wsa.ServiceIface.class,
                new AddressingFeature());
        // invoke method
        System.out.println(proxy.sayHello());
    }
}
```

The client and endpoint now communicate using WS-Addressing.

3.11. ENABLING WEB SERVICES RELIABLE MESSAGING

Web Services Reliable Messaging (WS-Reliable Messaging) is implemented internally in Apache CXF. A set of interceptors interacts with the low-level requirements of the reliable messaging protocol.

To enable WS-Reliable Messaging, complete one of the following steps:

- Consume a WSDL contract that specifies proper WS-Reliable Messaging policies, assertions, or both.
- Manually add and configure the reliable messaging interceptors.
- Specify the reliable messaging policies in an optional CXF Spring XML descriptor.
- Specify the Apache CXF reliable messaging feature in an optional CXF Spring XML descriptor.

The first approach, which is the only portable approach, relies on the Apache CXF WS-Policy engine. The other approaches, which are proprietary, allow for fine-grained configuration of the protocol aspects that are not covered by the WS-Reliable Messaging Policy.

3.12. SPECIFYING WEB SERVICES POLICIES

Web Services Policies (WS-Policy) rely on the Apache CXF WS-Policy framework. This framework is compliant with the following specifications:

- Web Services Policy 1.5 - Framework
- Web Services Policy 1.5 - Attachment

You can work with the policies in different ways, including:

- Add policy assertions to WSDL contracts and let the runtime consume the assertions and behave accordingly.
- Specify endpoint policy attachments using either CXF annotations or features.
- Use the Apache CXF policy framework to define custom assertions and complete other tasks.

3.13. APACHE CXF INTEGRATION

All Jakarta XML Web Services functionality provided by JBossWS on top of JBoss EAP is currently served through a proper integration of the JBossWS stack with most of the Apache CXF project modules.

Apache CXF is an open source services framework. It allows building and developing services using front-end programming APIs, including Jakarta XML Web Services, with services speaking a variety of protocols such as SOAP and XML/HTTP over a variety of transports such as HTTP and Jakarta Messaging.

The integration layer between JBossWS and Apache CXF is mainly meant for:

- Allowing use of standard web services APIs, including Jakarta XML Web Services, on JBoss EAP; this is performed internally leveraging Apache CXF without requiring the user to deal with it;
- Allowing use of Apache CXF advanced features, including WS-*, on top of JBoss EAP without requiring the user to deal with, set up, or care about the required integration steps for running in such a container.

In support of those goals, the JBossWS integration with Apache CXF supports the JBossWS endpoint deployment mechanism and comes with many internal customizations on top of Apache CXF.

For more in-depth details on the Apache CXF architecture, refer to the Apache CXF official documentation.

3.13.1. Server-side Integration Customization

The JBossWS server-side integration with Apache CXF takes care of internally creating proper Apache CXF structures for the provided web service deployment. If the deployment includes multiple endpoints, they will all exist within the same Apache CXF Bus, which is separate from other deployments' bus instances.

While JBossWS sets sensible defaults for most of the Apache CXF configuration options on the server side, users might want to fine-tune the Bus instance that is created for their deployment; a **jboss-webservices.xml** descriptor can be used for deployment-level customizations.

3.13.1.1. Deployment Descriptor Properties

The **jboss-webservices.xml** descriptor can be used to provide property values.

```
<webservices xmlns="http://www.jboss.com/xml/ns/javaee" version="1.2">
  ...
  <property>
    <name>...</name>
    <value>...</value>
  </property>
  ...
</webservices>
```

JBossWS integration with Apache CXF comes with a set of allowed property names to control Apache CXF internals.

3.13.1.2. WorkQueue Configuration

Apache CXF uses **WorkQueue** instances for dealing with some operations, for example **@Oneway** request processing. A **WorkQueueManager** is installed in the Bus as an extension and allows for adding or removing queues as well as controlling the existing ones.

On the server side, queues can be provided by using the **cxf.queue.<queue-name>.*** properties in **jboss-webservices.xml**. For example, you can use the **cxf.queue.default.maxQueueSize** property to configure the maximum queue size of the default **WorkQueue**. At the deployment time, the JBossWS integration can add new instances of **AutomaticWorkQueueImpl** to the currently configured **WorkQueueManager**. The properties below are used to fill in the **AutomaticWorkQueueImpl** constructor parameters:

Table 3.5. AutomaticWorkQueueImpl Constructor Properties

Property	Default Value
cxf.queue.<queue-name>.maxQueueSize	256
cxf.queue.<queue-name>.initialThreads	0
cxf.queue.<queue-name>.highWaterMark	25
cxf.queue.<queue-name>.lowWaterMark	5
cxf.queue.<queue-name>.dequeueTimeout	120000

3.13.1.3. Policy Alternative Selector

The Apache CXF policy engine supports different strategies to deal with policy alternatives. JBossWS integration currently defaults to the **MaximalAlternativeSelector**, but still allows for setting different selector implementation using the **cxf.policy.alternativeSelector** property in the **jboss-webservices.xml** file.

3.13.1.4. MBean Management

Apache CXF allows you to manage its MBean objects that are installed into the JBoss EAP MBean server. You can enable this feature on a deployment basis through the **cxf.management.enabled** property in the **jboss-webservices.xml** file. You can also use the **cxf.management.installResponseTimeInterceptors** property to control installation of the CXF response time interceptors. These interceptors are added by default when enabling the MBean management, but it might not be required in some cases.

Example: MBean Management in the jboss-webservices.xml File

```
<webservices xmlns="http://www.jboss.com/xml/ns/javaee" version="1.2">
  <property>
    <name>cxf.management.enabled</name>
    <value>true</value>
  </property>
  <property>
    <name>cxf.management.installResponseTimeInterceptors</name>
    <value>false</value>
  </property>
</webservices>
```

-

3.13.1.5. Schema Validation

Apache CXF includes a feature for validating incoming and outgoing SOAP messages on both the client and the server side. The validation is performed against the relevant schema in the endpoint WSDL contract (server side) or the WSDL contract used for building up the service proxy (client side).

You can enable schema validation in any of the following ways:

- In the JBoss EAP server configuration.
  For example, the management CLI command below enables schema validation for the default **Standard-Endpoint-Config** endpoint configuration.

  ```
  /subsystem=webservices/endpoint-config=Standard-Endpoint-Config/property=schema-validation-enabled:add(value=true)
  ```

- In a predefined client or endpoint configuration file.
  You can associate any endpoint or client running in-container to a JBossWS predefined configuration by setting the **schema-validation-enabled** property to **true** in the referenced configuration file.

- Programmatically on the client side.
  On the client side, you can enable schema validation programmatically. For example:

  ```
  ((BindingProvider)proxy).getRequestContext().put("schema-validation-enabled", true);
  ```

- Using the **@org.apache.cxf.annotations.SchemaValidation** annotation on the server side.
  On the server side, you can use the **@org.apache.cxf.annotations.SchemaValidation** annotation. For example:

  ```
  import javax.jws.WebService;
  import org.apache.cxf.annotations.SchemaValidation;

  @WebService(...)
  @SchemaValidation
  public class ValidatingHelloImpl implements Hello {
     ...
  }
  ```

3.13.1.6. Apache CXF Interceptors

The **jboss-webservices.xml** descriptor enables specifying the **cxf.interceptors.in** and **cxf.interceptors.out** properties. These properties allow you to attach the declaring interceptors to the **Bus** instance that is created for serving the deployment.

Example: jboss-webservices.xml File

```
<?xml version="1.1" encoding="UTF-8"?>
<webservices
  xmlns="http://www.jboss.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="1.2"
  xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee">
```

```
  <property>
    <name>cxf.interceptors.in</name>
    <value>org.jboss.test.ws.jaxws.cxf.interceptors.BusInterceptor</value>
  </property>
  <property>
    <name>cxf.interceptors.out</name>
    <value>org.jboss.test.ws.jaxws.cxf.interceptors.BusCounterInterceptor</value>
  </property>
</webservices>
```

You can declare interceptors using one of the following approaches:

- Annotation usage on endpoint classes, for example **@org.apache.cxf.interceptor.InInterceptor** or **@org.apache.cxf.interceptor.OutInterceptor**.
- Direct API usage on the client side through the **org.apache.cxf.interceptor.InterceptorProvider** interface.
- JBossWS descriptor usage.

Because Spring integration is no longer supported in JBoss EAP, the JBossWS integration uses the **jaxws-endpoint-config.xml** descriptor file to avoid requiring modifications to the actual client or endpoint code. You can declare interceptors within predefined client and endpoint configurations by specifying a list of interceptor class names for the **cxf.interceptors.in** and **cxf.interceptors.out** properties.

Example: jaxws-endpoint-config.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <endpoint-config>
    <config-name>org.jboss.test.ws.jaxws.cxf.interceptors.EndpointImpl</config-name>
    <property>
      <property-name>cxf.interceptors.in</property-name>
      <property-
value>org.jboss.test.ws.jaxws.cxf.interceptors.EndpointInterceptor,org.jboss.test.ws.jaxws.cxf.intercepto
s.FooInterceptor</property-value>
    </property>
    <property>
      <property-name>cxf.interceptors.out</property-name>
      <property-value>org.jboss.test.ws.jaxws.cxf.interceptors.EndpointCounterInterceptor</property-
value>
    </property>
  </endpoint-config>
</jaxws-config>
```

NOTE

A new instance of each specified interceptor class will be added to the client or endpoint to which the configuration is assigned. The interceptor classes must have a no-argument constructor.

3.13.1.7. Apache CXF Features

The **jboss-webservices.xml** descriptor enables specifying the **cxf.features** property. This property allows you to declare features to be attached to any endpoint belonging to the **Bus** instance that is created for serving the deployment.

Example: jboss-webservices.xml File

```
<?xml version="1.1" encoding="UTF-8"?>
<webservices
  xmlns="http://www.jboss.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="1.2"
  xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee">

  <property>
    <name>cxf.features</name>
    <value>org.apache.cxf.feature.FastInfosetFeature</value>
  </property>
</webservices>
```

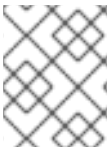
You can declare features using one of the following approaches:

- Annotation usage on endpoint classes, for example **@org.apache.cxf.feature.Features**.
- Direct API usage on client side through extensions of the **org.apache.cxf.feature.AbstractFeature** class.
- JBossWS descriptor usage.

Since Spring integration is no longer supported in JBoss EAP, the JBossWS integration adds an additional descriptor, a **jaxws-endpoint-config.xml** file-based approach to avoid requiring modifications to the actual client or endpoint code. You can declare features within predefined client and endpoint configurations by specifying a list of feature class names for the **cxf.features** property.

Example: jaxws-endpoint-config.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <endpoint-config>
    <config-name>Custom FI Config</config-name>
    <property>
      <property-name>cxf.features</property-name>
      <property-value>org.apache.cxf.feature.FastInfosetFeature</property-value>
    </property>
  </endpoint-config>
</jaxws-config>
```



NOTE

A new instance of each specified feature class will be added to the client or endpoint the configuration is assigned to. The feature classes must have a no-argument constructor.

3.13.1.8. Properties-Driven Bean Creation

The Apache CXF Interceptors and Apache CXF Features sections explain how to declare CXF interceptors and features through properties either in a client or endpoint predefined configuration or in a **jboss-webservices.xml** descriptor. By only getting the feature or interceptor class name specified, the container tries to create a bean instance using the class default constructor. This sets a limitation on the feature or interceptor configuration, unless custom extensions of vanilla CXF classes are provided, with the default constructor setting properties before eventually using the super constructor.

To address this issue, JBossWS integration comes with a mechanism for configuring simple bean hierarchies when building them up from properties. Properties can have bean reference values, which are strings starting with **##**. Property reference keys are used to specify the bean class name and the value for each attribute.

So for instance, the following properties result in the stack installing two feature instances:

Key	Value
cxf.features	##foo, ##bar
##foo	org.jboss.Foo
##foo.par	34
##bar	org.jboss.Bar
##bar.color	blue

The same result can be created by the following code:

```
import org.Bar;
import org.Foo;
...
Foo foo = new Foo();
foo.setPar(34);
Bar bar = new Bar();
bar.setColor("blue");
```

This mechanism assumes that the classes are valid beans with proper **getter()** and **setter()** methods. Value objects are cast to the correct primitive type by inspecting the class definition. Nested beans can also be configured.

APPENDIX A. REFERENCE MATERIAL

A.1. JAKARTA RESTFUL WEB SERVICES/RESTEASY ANNOTATIONS

Table A.1. Jakarta RESTful Web Services/RESTEasy Annotations

Annotation	Usage
Cache	Set response **Cache-Control** header automatically.
ClientInterceptor	Identifies an interceptor as a client-side interceptor.
ContentEncoding	Meta annotation that specifies a **Content-Encoding** to be applied via the annotated annotation.
Context	Allows you to specify instances of **javax.ws.rs.core.HttpHeaders**, **javax.ws.rs.core.UriInfo**, **javax.ws.rs.core.Request**, **javax.servlet.HttpServletRequest**, **javax.servlet.HttpServletResponse**, and **javax.ws.rs.core.SecurityContext** objects.
CookieParam	Allows you to specify the value of a cookie or object representation of an HTTP request cookie into the method invocation.
DecorateTypes	Must be placed on a **DecoratorProcessor** class to specify the supported types.
Decorator	Meta-annotation to be placed on another annotation that triggers decoration.
DefaultValue	Can be combined with the other **@*Param** annotations to define a default value when the HTTP request item does not exist.
DELETE	An annotation that signifies that the method responds to HTTP **DELETE** requests.
DoNotUseJAXBProvider	Put this on a class or parameter when you do not want the Jakarta XML Binding **MessageBodyReader/Writer** used but instead have a more specific provider you want to use to marshal the type.

Annotation	Usage
Encoded	Can be used on a class, method, or param. By default, inject **@PathParam** and **@QueryParams** are decoded. By adding the **@Encoded** annotation, the value of these params are provided in encoded form.
Form	This can be used as a value object for incoming/outgoing request/responses.
Formatted	Format XML output with indentations and newlines. This is a Jakarta XML Binding Decorator.
GET	An annotation that signifies that the method responds to HTTP **GET** requests.
IgnoreMediaTypes	Placed on a type, method, parameter, or field to tell Jakarta RESTful Web Services not to use Jakarta XML Binding provider for a certain media type
ImageWriterParams	An annotation that a resource class can use to pass parameters to the **IIOImageProvider**.
Mapped	A **JSONConfig**.
MultipartForm	This can be used as a value object for incoming/outgoing request/responses of the multipart/form-data MIME type.
NoCache	Set **Cache-Control** response header of**nocache**.
NoJackson	Placed on class, parameter, field or method when you do not want the Jackson provider to be triggered.
PartType	Must be used in conjunction with Multipart providers when writing out a List or Map as a **multipart/*** type.
Path	This must exist either in the class or resource method. If it exists in both, the relative path to the resource method is a concatenation of the class and method.
PathParam	Allows you to map variable URI path fragments into a method call.
POST	An annotation that signifies that the method responds to HTTP **POST** requests.

Annotation	Usage
Priority	An annotation to indicate what order a class should be used. Uses an integer parameter with a lower value signifying a higher priority.
Provider	Marks a class to be discoverable as a provider by Jakarta RESTful Web Services runtime during a provider scanning phase.
PUT	An annotation that signifies that the method responds to HTTP **PUT** requests.
QueryParam	Allows you to map URI query string parameter or URL form encoded parameter to the method invocation.
ServerInterceptor	Identifies an interceptor as a server-side interceptor.
StringParameterUnmarshallerBinder	Meta-annotation to be placed on another annotation that triggers a **StringParameterUnmarshaller** to be applied to a string-based annotation injector.
Stylesheet	Specifies an XML stylesheet header.
Wrapped	Put this on a method or parameter when you want to marshal or unmarshal a collection or array of Jakarta XML Binding objects.
WrappedMap	Put this on a method or parameter when you want to marshal or unmarshal a map of Jakarta XML Binding objects.
XmlHeader	Sets an XML header for the returned document.
XmlNsMap	A **JSONToXml**.
XopWithMultipartRelated	This annotation can be used to process/produce incoming/outgoing XOP messages (packaged as multipart/related) to/from Jakarta XML Binding annotated objects.

A.2. RESTEASY CONFIGURATION PARAMETERS

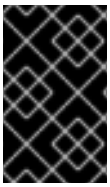
Table A.2. Elements

Option Name	Default Value	Description
resteasy.servlet.mapping.prefix	No default	If the URL-pattern for the Resteasy servlet-mapping is not **/***.
resteasy.scan	false	Automatically scan **WEB-INF/lib** JARs and **WEB-INF/classes** directory for both **@Provider** and Jakarta RESTful Web Services resource classes (such as **@Path**, **@GET**, **@POST**) and register them.
resteasy.scan.providers	false	Scan for **@Provider** classes and register them.
resteasy.scan.resources	false	Scan for Jakarta RESTful Web Services resource classes.
resteasy.providers	no default	A comma delimited list of fully qualified **@Provider** class names you want to register.
resteasy.use.builtin.providers	true	Whether or not to register default, built-in **@Provider** classes.
resteasy.resources	No default	A comma delimited list of fully qualified Jakarta RESTful Web Services resource class names you want to register.
resteasy.jndi.resources	No default	A comma delimited list of JNDI names which reference objects you want to register as Jakarta RESTful Web Services resources.
javax.ws.rs.Application	No default	Fully qualified name of **Application** class to bootstrap in a spec portable way.
resteasy.media.type.mappings	No default	Replaces the need for an **Accept** header by mapping file name extensions (like **.xml** or **.txt**) to a media type. Used when the client is unable to use a **Accept** header to choose a representation (i.e. a browser). You configure this in the **WEB-INF/web.xml** file using the **resteasy.media.type.mappings** and **resteasy.language.mappings**.

Option Name	Default Value	Description
resteasy.language.mappings	No default	Replaces the need for an **Accept-Language** header by mapping file name extensions (like **.en** or **.fr**) to a language. Used when the client is unable to use a **Accept-Language** header to choose a language (i.e. a browser).
resteasy.document.expand.entity.references	false	Whether to expand external entities or replace them with an empty string. In JBoss EAP, this parameter defaults to **false**, so it replaces them with an empty string.
resteasy.document.secure.processing.feature	true	Impose security constraints in processing **org.w3c.dom.Document** documents and Jakarta XML Binding object representations.
resteasy.document.secure.disableDTDs	true	Prohibit DTDs in **org.w3c.dom.Document** documents and Jakarta XML Binding object representations.
resteasy.wider.request.matching	true	Turn off class-level expression filtering as defined in the Jakarta RESTful Web Services specification and instead match based on the full expression of each Jakarta RESTful Web Services method.
resteasy.use.container.form.params	true	Use the **HttpServletRequest.getParameterMap()** method to obtain form parameters. Use this switch if you are calling this method within a servlet filter or consuming the input stream within the filter.
resteasy.add.charset	true	If a resource method returns a **text/*** or **application/xml*** media type without an explicit charset, RESTEasy adds **charset=UTF-8** to the returned content-type header. Note that the charset defaults to UTF-8 in this case, independent of the setting of this parameter.

NOTE

These parameters are configured in the **WEB-INF/web.xml** file.



IMPORTANT

In a Servlet 3.0 container, the **resteasy.scan.*** configurations in the **web.xml** file are ignored, and all Jakarta RESTful Web Services annotated components will be automatically scanned.

For example, **javax.ws.rs.Application** parameter is configured within **init-param** of the servlet configuration:

```
<servlet>
   <servlet-name>Resteasy</servlet-name>
   <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-class>
   <init-param>
      <param-name>javax.ws.rs.Application</param-name>
      <param-value>org.jboss.resteasy.utils.TestApplication</param-value>
   </init-param>
</servlet>
```

For example, **resteasy.document.expand.entity.references** is configured within **context-param**:

```
<context-param>
   <param-name>resteasy.document.expand.entity.references</param-name>
   <param-value>true</param-value>
</context-param>
```

WARNING

Changing the default values of the following RESTEasy parameters may cause RESTEasy applications to be potentially vulnerable against XXE attacks:

- resteasy.document.expand.entity.references
- resteasy.document.secure.processing.feature
- resteasy.document.secure.disableDTDs

A.3. RESTEASY JAVASCRIPT API PARAMETERS

Table A.3. Parameter Properties

Property	Default Value	Description
$entity		The entity to send as a **PUT**, **POST** request.
$contentType		The MIME type of the body entity sent as the **Content-Type** header. Determined by the **@Consumes** annotation.

Property	Default Value	Description
$accepts	*/*	The accepted MIME types sent as the **Accept** header. Determined by the **@Provides** annotation.
$callback		Set to a function (**httpCode**, **xmlHttpRequest**, **value**) for an asynchronous call. If not present, the call will be synchronous and return the value.
$apiURL		Set to the base URI of the Jakarta RESTful Web Services endpoint, not including the last slash.
$username		If username and password are set, they will be used for credentials for the request.
$password		If username and password are set, they will be used for credentials for the request.

A.4. REST.REQUEST CLASS MEMBERS

Table A.4. REST.Request Class

Member	Description
execute(callback)	Executes the request with all the information set in the current object. The value is passed to the optional argument callback, not returned.
setAccepts(acceptHeader)	Sets the **Accept** request header. Defaults to ***/***.
setCredentials(username, password)	Sets the request credentials.
setEntity(entity)	Sets the request entity.
setContentType(contentTypeHeader)	Sets the **Content-Type** request header.
setURI(uri)	Sets the request URI. This should be an absolute URI.
setMethod(method)	Sets the request method. Defaults to **GET**.
setAsync(async)	Controls whether the request should be asynchronous. Defaults to **true**.
addCookie(name, value)	Sets the given cookie in the current document when executing the request. This will be persistent in the browser.

Member	Description
addQueryParameter(name, value)	Adds a query parameter to the URI query part.
addMatrixParameter(name, value)	Adds a matrix parameter (path parameter) to the last path segment of the request URI.
addHeader(name, value)	Adds a request header.
addForm(name, value)	Adds a form.
addFormParameter(name, value)	Adds a form parameter.

A.5. RESTEASY ASYNCHRONOUS JOB SERVICE CONFIGURATION PARAMETERS

The table below details the configurable **context-params** for the Asynchronous Job Service. These parameters can be configured in the **web.xml** file.

Table A.5. Configuration Parameters

Parameter	Description
resteasy.async.job.service.max.job.results	Number of job results that can be held in the memory at any one time. Default value is **100**.
resteasy.async.job.service.max.wait	Maximum wait time on a job when a client is querying for it. Default value is **300000**.
resteasy.async.job.service.thread.pool.size	Thread pool size of the background threads that run the job. Default value is **100**.
resteasy.async.job.service.base.path	Sets the base path for the job URIs. Default value is **/asynch/jobs**.

```
<web-app>
   <context-param>
      <param-name>resteasy.async.job.service.enabled</param-name>
      <param-value>true</param-value>
   </context-param>

   <context-param>
      <param-name>resteasy.async.job.service.max.job.results</param-name>
      <param-value>100</param-value>
   </context-param>
   <context-param>
      <param-name>resteasy.async.job.service.max.wait</param-name>
      <param-value>300000</param-value>
   </context-param>
```

```
    <context-param>
      <param-name>resteasy.async.job.service.thread.pool.size</param-name>
      <param-value>100</param-value>
    </context-param>
    <context-param>
      <param-name>resteasy.async.job.service.base.path</param-name>
      <param-value>/asynch/jobs</param-value>
    </context-param>

    <listener>
      <listener-class>
        org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap
      </listener-class>
    </listener>

    <servlet>
      <servlet-name>Resteasy</servlet-name>
      <servlet-class>
        org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
      </servlet-class>
    </servlet>

    <servlet-mapping>
      <servlet-name>Resteasy</servlet-name>
      <url-pattern>/*</url-pattern>
    </servlet-mapping>

</web-app>
```

A.6. JAKARTA XML WEB SERVICES TOOLS

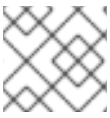
wsconsume

wsconsume is a command-line tool provided with JBoss EAP that consumes a WSDL and produces portable Jakarta XML Web Services service and client artifacts.

Usage

The **wsconsume** tool is located in the **EAP_HOME/bin** directory and uses the following syntax.

```
EAP_HOME/bin/wsconsume.sh [options] <wsdl-url>
```



NOTE

Use the **wsconsume.bat** script for Windows.

Example usage:

- Generate Java class files from the **Example.wsdl** WSDL file:

```
EAP_HOME/bin/wsconsume.sh Example.wsdl
```

- Generate Java source and class files from the **Example.wsdl** WSDL file:

```
EAP_HOME/bin/wsconsume.sh -k Example.wsdl
```

- Generate Java source and class files in the **my.org** package from the **Example.wsdl** WSDL file:

```
EAP_HOME/bin/wsconsume.sh -k -p my.org Example.wsdl
```

- Generate Java source and class files using multiple binding files:

```
EAP_HOME/bin/wsconsume.sh -k -b schema-binding1.xsd -b schema-binding2.xsd
Example.wsdl
```

- Configure the **-Dlog4j** property, in the ***JAVA_OPTS*** environment variable, to use with the **wsconsume.sh** script:
  - This configuration allows you to create a customizable logging file and set the log levels of the application.
  - To load the logging configuration file, use either **-Dlog4j.configuration=file:log4j.prsoperties** or **-Dlog4j.configuration=file:log4j.xml** as shown in the following example:

```
JAVA_OPTS="-Dlog4j.configuration=file:log4j.properties" ./bin/wsconsume.sh
<WSDL_URL>
```

    where, ***<WSDL_URL>*** denotes the WSDL file path to use when generating the code.

Use the **--help** argument or see the below table for a listing of all available **wsconsume** options.

Table A.6. wsconsume Options

Option	Description
-a, --additionalHeaders	Enable processing of implicit SOAP headers.
-b, --binding=<file>	One or more Jakarta XML Web Services or Jakarta XML Binding binding files.
-c --catalog=<file>	Oasis XML Catalog file for entity resolution.
-d --encoding=<charset>	The charset encoding to use for generated sources.
-e, --extension	Enable SOAP 1.2 binding extension.
-h, --help	Show this help message.
-j --clientjar=<name>	Create a JAR file of the generated artifacts for calling the web. service
-k, --keep	Keep/Generate Java source.
-l, --load-consumer	Load the consumer and exit (debug utility).
-n, --nocompile	Do not compile generated sources.

Option	Description
-o, --output=<directory>	The directory to put generated artifacts.
-p --package=<name>	The target package for generated source.
-q, --quiet	Be somewhat more quiet.
-s, --source=<directory>	The directory to put Java source.
-t, --target=<2.1|2.2>	The Jakarta XML Web Services specification target.
-v, --verbose	Show full exception stack traces.
-w --wsdlLocation=<loc>	Value to use for **@WebService.wsdlLocation**.

wsprovide

wsprovide is a command-line tool provided with JBoss EAP that generates portable Jakarta XML Web Services artifacts for a service endpoint implementation. It also has the option to generate a WSDL file.

Usage

The **wsprovide** tool is located in the **EAP_HOME/bin** directory and uses the following syntax.

```
EAP_HOME/bin/wsprovide.sh [options] <endpoint class name>
```

NOTE

Use the **wsprovide.bat** script for Windows.

Example usage:

- Generate wrapper classes for portable artifacts in the **output** directory.

```
EAP_HOME/bin/wsprovide.sh -o output my.package.MyEndpoint
```

- Generate wrapper classes and WSDL in the **output** directory.

```
EAP_HOME/bin/wsprovide.sh -o output -w my.package.MyEndpoint
```

- Generate wrapper classes in the **output** directory for an endpoint that references other JARs.

```
EAP_HOME/bin/wsprovide.sh -o output -c myapplication1.jar:myapplication2.jar
my.org.MyEndpoint
```

- Configure the **-Dlog4j** property, in the ***JAVA_OPTS*** environment variable, to use with the **wsprovide.sh** script:
  - This configuration allows you to create a customizable logging file and set the log levels of the application.

- To load the logging configuration file, use either **-Dlog4j.configuration=file:log4j.properties** or **-Dlog4j.configuration=file:log4j.xml** as shown in the following example:

```
JAVA_OPTS="-Dlog4j.configuration=file:log4j.properties" ./bin/wsprovide.sh
```

Use the **--help** argument or see the below table for a listing of all available **wsprovide** options.

Table A.7. wsprovide Options

Option	Description
-a, --address=<address>	The generated port soap:address in WSDL.
-c, --classpath=<path>	The classpath that contains the endpoint.
-e, --extension	Enable SOAP 1.2 binding extension.
-h, --help	Show this help message.
-k, --keep	Keep/Generate Java source.
-l, --load-provider	Load the provider and exit (debug utility).
-o, --output=<directory>	The directory to put generated artifacts.
-q, --quiet	Be somewhat more quiet.
-r, --resource=<directory>	The directory to put resource artifacts.
-s, --source=<directory>	The directory to put Java source.
-t, --show-traces	Show full exception stack traces.
-w, --wsdl	Enable WSDL file generation.

A.7. JAKARTA XML WEB SERVICES COMMON API REFERENCE

Several Jakarta XML Web Services development concepts are shared between web service endpoints and clients. These include the handler framework, message context, and fault handling.

Handler Framework

The handler framework is implemented by a Jakarta XML Web Services protocol binding in the runtime of the client and the endpoint, which is the server component. Proxies and **Dispatch** instances, known collectively as binding providers, each use protocol bindings to bind their abstract functionality to specific protocols.

Client and server-side handlers are organized into an ordered list known as a handler chain. The handlers within a handler chain are invoked each time a message is sent or received. Inbound messages are processed by handlers before the binding provider processes them. Outbound messages are processed

by handlers after the binding provider processes them.

Handlers are invoked with a message context which provides methods to access and modify inbound and outbound messages and to manage a set of properties. Message context properties facilitate communication between individual handlers, as well as between handlers and client and service implementations. Different types of handlers are invoked with different types of message contexts.

Logical Handler

Logical handlers only operate on message context properties and message payloads. Logical handlers are protocol-independent and cannot affect protocol-specific parts of a message. Logical handlers implement interface **javax.xml.ws.handler.LogicalHandler**.

Protocol Handler

Protocol handlers operate on message context properties and protocol-specific messages. Protocol handlers are specific to a particular protocol and may access and change protocol-specific aspects of a message. Protocol handlers implement any interface derived from **javax.xml.ws.handler.Handler**, except **javax.xml.ws.handler.LogicalHandler**.

Service Endpoint Handler

On a service endpoint, handlers are defined using the **@HandlerChain** annotation. The location of the handler chain file can be either an absolute **java.net.URL** in **externalForm** or a relative path from the source file or class file.

```
@WebService
@HandlerChain(file = "jaxws-server-source-handlers.xml")
public class SOAPEndpointSourceImpl {
  ...
}
```

Service Client Handler

On a Jakarta XML Web Services client, handlers are defined either by using the **@HandlerChain** annotation, as in service endpoints, or dynamically, using the Jakarta XML Web Services API.

```
Service service = Service.create(wsdlURL, serviceName);
Endpoint port = (Endpoint)service.getPort(Endpoint.class);

BindingProvider bindingProvider = (BindingProvider)port;
List<Handler> handlerChain = new ArrayList<Handler>();
handlerChain.add(new LogHandler());
handlerChain.add(new AuthorizationHandler());
handlerChain.add(new RoutingHandler());
bindingProvider.getBinding().setHandlerChain(handlerChain);
```

The call to the **setHandlerChain** method is required.

Message Context

The **MessageContext** interface is the super interface for all Jakarta XML Web Services message contexts. It extends **Map<String,Object>** with additional methods and constants to manage a set of properties that enable handlers in a handler chain to share processing related state. For example, a handler may use the **put** method to insert a property into the message context. One or more other handlers in the handler chain may subsequently obtain the message using the **get** method.

Properties are scoped as either **APPLICATION** or **HANDLER**. All properties are available to all handlers for an instance of a message exchange pattern (MEP) of a particular endpoint. For instance, if a logical

handler puts a property into the message context, that property is also available to any protocol handlers in the chain during the execution of an MEP instance.

NOTE

An asynchronous Message Exchange Pattern (MEP) allows for sending and receiving messages asynchronously at the HTTP connection level. You can enable it by setting additional properties in the request context.

Properties scoped at the **APPLICATION** level are also made available to client applications and service endpoint implementations. The **defaultscope** for a property is **HANDLER**.

Logical and SOAP messages use different contexts.

Logical Message Context

When logical handlers are invoked, they receive a message context of type **LogicalMessageContext**. **LogicalMessageContext** extends **MessageContext** with methods which obtain and modify the message payload. It does not provide access to the protocol-specific aspects of a message. A protocol binding defines which components of a message are available through a logical message context. A logical handler deployed in a SOAP binding can access the contents of the SOAP body but not the SOAP headers. On the other hand, the XML/HTTP binding defines that a logical handler can access the entire XML payload of a message.

SOAP Message Context

When SOAP handlers are invoked, they receive a **SOAPMessageContext**. **SOAPMessageContext** extends **MessageContext** with methods which obtain and modify the SOAP message payload.

Fault Handling

An application may throw a **SOAPFaultException** or an application-specific user exception. In the case of the latter, the required fault wrapper beans are generated at runtime if they are not already part of the deployment.

```
public void throwSoapFaultException() {
  SOAPFactory factory = SOAPFactory.newInstance();
  SOAPFault fault = factory.createFault("this is a fault string!", new QName("http://foo", "FooCode"));
  fault.setFaultActor("mr.actor");
  fault.addDetail().addChildElement("test");
  throw new SOAPFaultException(fault);
}
```

```
public void throwApplicationException() throws UserException {
  throw new UserException("validation", 123, "Some validation error");
}
```

Jakarta XML Web Services Annotations

The annotations available by the Jakarta XML Web Services API are defined in Jakarta XML Web Services Specification 2.3 specification These annotations are in the **javax.xml.ws** package.

The annotations available by the JWS API are defined in Web Services Metadata is in the Jakarta Web Services Metadata Specification 2.1 specification. These annotations are in the **javax.jws** package.

A.8. ADVANCED WS-TRUST SCENARIOS

A.8.1. Scenario: SAML Holder-Of-Key Assertion Scenario

WS-Trust helps in managing software security tokens. A SAML assertion is a type of security token. In the Holder-Of-Key method, STS creates a SAML token containing the client's public key and signs the SAML token with its private key. The client includes the SAML token and signs the outgoing soap envelope to the web service with its private key. The web service validates the SOAP message and SAML token.

Implementation of this scenario requires the following:

- SAML tokens with a Holder-Of-Key subject confirmation method must be protected so the token cannot be snooped. In most cases, a Holder-Of-Key token combined with HTTPS is sufficient to prevent getting possession of the token. This means the security policy uses a **sp:TransportBinding** and **sp:HttpsToken**.
- A Holder-Of-Key token has no encryption or signing keys associated with it, therefore a **sp:IssuedToken** of **SymmetricKey** or **PublicKey** keyType should be used with a **sp:SignedEndorsingSupportingTokens**.

A.8.1.1. Web Service Provider

This section lists the web service elements for the SAML Holder-Of-Key scenario. The components include:

- Web Service Provider WSDL
- SSL Configuration
- Web Service Provider Interface
- Web Service Provider Implementation
- Crypto Properties and Keystore Files
- Default MANIFEST.MF

A.8.1.1.1. Web Service Provider WSDL

The Web Service Provider is a contract-first endpoint. All WS-trust and security policies for it are declared in the **HolderOfKeyService.wsdl** WSDL. For this scenario, a **ws-requester** is required to provide a SAML 2.0 token of **SymmetricKey** keyType, issued from a designated STS. The STS address is provided in the WSDL. A transport binding policy is used. The token is declared to be signed and endorsed, **sp:SignedEndorsingSupportingTokens**.

A detailed explanation of the security settings are provided in the comments in the following listing:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-
extensions/holderofkeywssecuritypolicy"
             name="HolderOfKeyService"
    xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
```

```
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsaws="http://www.w3.org/2005/08/addressing"
    xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex"
    xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
    xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">

  <types>
    <xsd:schema>
      <xsd:import namespace="http://www.jboss.org/jbossws/ws-
extensions/holderofkeywssecuritypolicy"
                  schemaLocation="HolderOfKeyService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <portType name="HolderOfKeyIface">
    <operation name="sayHello">
      <input message="tns:sayHello"/>
      <output message="tns:sayHelloResponse"/>
    </operation>
  </portType>
<!--
        The wsp:PolicyReference binds the security requirements on all the endpoints.
        The wsp:Policy wsu:Id="#TransportSAML2HolderOfKeyPolicy" element is defined later in this
file.
-->
  <binding name="HolderOfKeyServicePortBinding" type="tns:HolderOfKeyIface">
    <wsp:PolicyReference URI="#TransportSAML2HolderOfKeyPolicy" />
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="sayHello">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
<!--
  The soap:address has been defined to use JBoss's https port, 8443.  This is
  set in conjunction with the sp:TransportBinding policy for https.
-->
  <service name="HolderOfKeyService">
    <port name="HolderOfKeyServicePort" binding="tns:HolderOfKeyServicePortBinding">
      <soap:address location="https://@jboss.bind.address@:8443/jaxws-samples-wsse-policy-trust-
holderofkey/HolderOfKeyService"/>
    </port>
  </service>


  <wsp:Policy wsu:Id="TransportSAML2HolderOfKeyPolicy">
```

```
   <wsp:ExactlyOne>
     <wsp:All>
  <!--
       The wsam:Addressing element, indicates that the endpoints of this
       web service MUST conform to the WS-Addressing specification.  The
       attribute wsp:Optional="false" enforces this assertion.
  -->
       <wsam:Addressing wsp:Optional="false">
         <wsp:Policy />
       </wsam:Addressing>
<!--
  The sp:TransportBinding element indicates that security is provided by the
  message exchange transport medium, https.  WS-Security policy specification
  defines the sp:HttpsToken for use in exchanging messages transmitted over HTTPS.
-->
        <sp:TransportBinding
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
            <sp:TransportToken>
              <wsp:Policy>
                <sp:HttpsToken>
                  <wsp:Policy/>
                </sp:HttpsToken>
              </wsp:Policy>
            </sp:TransportToken>
<!--
    The sp:AlgorithmSuite element, requires the TripleDes algorithm suite
    be used in performing cryptographic operations.
-->
            <sp:AlgorithmSuite>
              <wsp:Policy>
                <sp:TripleDes />
              </wsp:Policy>
            </sp:AlgorithmSuite>
<!--
    The sp:Layout element,  indicates the layout rules to apply when adding
    items to the security header.  The sp:Lax sub-element indicates items
    are added to the security header in any order that conforms to
    WSS: SOAP Message Security.
-->
            <sp:Layout>
              <wsp:Policy>
                <sp:Lax />
              </wsp:Policy>
            </sp:Layout>
            <sp:IncludeTimestamp />
          </wsp:Policy>
        </sp:TransportBinding>

<!--
  The sp:SignedEndorsingSupportingTokens, when transport level security level is
  used there will be no message signature and the signature generated by the
  supporting token will sign the Timestamp.
-->
       <sp:SignedEndorsingSupportingTokens
         xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
```

```
        <wsp:Policy>
<!--
  The sp:IssuedToken element asserts that a SAML 2.0 security token of type
  Bearer is expected from the STS.  The
  sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
  attribute instructs the runtime to include the initiator's public key
  with every message sent to the recipient.

  The sp:RequestSecurityTokenTemplate element directs that all of the
  children of this element will be copied directly into the body of the
  RequestSecurityToken (RST) message that is sent to the STS when the
  initiator asks the STS to issue a token.
-->
          <sp:IssuedToken
            sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
            <sp:RequestSecurityTokenTemplate>
              <t:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV2.0</t:TokenType>
 <!--
   KeyType of "SymmetricKey", the client must prove to the WS service that it
   possesses a particular symmetric session key.
 -->
              <t:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey</t:KeyType>
            </sp:RequestSecurityTokenTemplate>
            <wsp:Policy>
              <sp:RequireInternalReference />
            </wsp:Policy>
<!--
  The sp:Issuer element defines the STS's address and endpoint information
  This information is used by the STSClient.
-->
            <sp:Issuer>
              <wsaws:Address>http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-sts-
holderofkey/SecurityTokenService</wsaws:Address>
              <wsaws:Metadata
                xmlns:wsdli="http://www.w3.org/2006/01/wsdl-instance"
                wsdli:wsdlLocation="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-
sts-holderofkey/SecurityTokenService?wsdl">
                <wsaw:ServiceName
                  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
                  xmlns:stsns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
                  EndpointName="UT_Port">stsns:SecurityTokenService</wsaw:ServiceName>
              </wsaws:Metadata>
            </sp:Issuer>

          </sp:IssuedToken>
        </wsp:Policy>
      </sp:SignedEndorsingSupportingTokens>
<!--
    The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
    to be supported by the STS.  These particular elements generally refer
    to how keys are referenced within the SOAP envelope.  These are normally handled by Apache
CXF.
-->
```

```
      <sp:Wss11>
        <wsp:Policy>
          <sp:MustSupportRefIssuerSerial />
          <sp:MustSupportRefThumbprint />
          <sp:MustSupportRefEncryptedKey />
        </wsp:Policy>
      </sp:Wss11>
<!--
   The sp:Trust13 element declares controls for WS-Trust 1.3 options.
   They are policy assertions related to exchanges specifically with
   client and server challenges and entropy behaviors.  Again these are
   normally handled by Apache CXF.
-->
      <sp:Trust13>
        <wsp:Policy>
          <sp:MustSupportIssuedTokens />
          <sp:RequireClientEntropy />
          <sp:RequireServerEntropy />
        </wsp:Policy>
      </sp:Trust13>
    </wsp:All>
  </wsp:ExactlyOne>
 </wsp:Policy>

</definitions>
```

A.8.1.1.2. SSL Configuration

This web service uses HTTPS, therefore the JBoss EAP server must be configured to provide SSL/TLS support in the **undertow** subsystem.

For information on how to configure HTTPS for web applications, see Configure One-way and Two-way SSL/TLS for Applications in *How to Configure Server Security* .

A.8.1.1.3. Web Service Provider Interface

The web service provider interface **HolderOfKeyIface** class is a simple web service definition.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
  targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy"
)
public interface HolderOfKeyIface {
  @WebMethod
  String sayHello();
}
```

A.8.1.1.4. Web Service Provider Implementation

The web service provider implementation **HolderOfKeyImpl** class is a simple POJO. It uses the

standard **WebService** annotation to define the service endpoint. In addition there are two Apache CXF annotations, **EndpointProperties** and **EndpointProperty** used for configuring the endpoint for the Apache CXF runtime. These annotations come from the Apache WSS4J project, which provides a Java implementation of the primary WS-Security standards for web services. These annotations programmatically add properties to the endpoint. With plain Apache CXF, these properties are often set using the **<jaxws:properties>** element on the **<jaxws:endpoint>** element in the Spring configuration. These annotations allow the properties to be configured in the code.

WSS4J uses the Crypto interface to get keys and certificates for signature creation/verification, as asserted by the WSDL for this service. The WSS4J configuration information provided by **HolderOfKeyImpl** is for Crypto's Merlin implementation.

The first **EndpointProperty** statement in the listing disables ensurance of compliance with the Basic Security Profile 1.1. The next **EndpointProperty** statements declares the Java properties file that contains the (Merlin) Crypto configuration information. The last **EndpointProperty** statement declares the **STSHolderOfKeyCallbackHandler** implementation class. It is used to obtain the user's password for the certificates in the keystore file.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;

import javax.jws.WebService;

@WebService
  (
    portName = "HolderOfKeyServicePort",
    serviceName = "HolderOfKeyService",
    wsdlLocation = "WEB-INF/wsdl/HolderOfKeyService.wsdl",
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy",
    endpointInterface =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey.HolderOfKeyIface"
  )
@EndpointProperties(value = {
  @EndpointProperty(key = "ws-security.is-bsp-compliant", value = "false"),
  @EndpointProperty(key = "ws-security.signature.properties", value = "serviceKeystore.properties"),
  @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey.HolderOfKeyCallbackHandler")
})
public class HolderOfKeyImpl implements HolderOfKeyIface {
  public String sayHello() {
    return "Holder-Of-Key WS-Trust Hello World!";
  }
}
```

A.8.1.1.5. Crypto Properties and Keystore Files

WSS4J's Crypto implementation is loaded and configured using a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and so on. This application uses the Merlin implementation. The **serviceKeystore.properties** file contains this information.

The **servicestore.jks** file is a Java KeyStore (JKS) repository. It contains self-signed certificates for **myservicekey** and **mystskey**.

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=sspass
org.apache.ws.security.crypto.merlin.keystore.alias=myservicekey
org.apache.ws.security.crypto.merlin.keystore.file=servicestore.jks
```

A.8.1.1.6. Default MANIFEST.MF

This application requires access to JBossWS and Apache CXF APIs provided in the **org.jboss.ws.cxf.jbossws-cxf-client** module. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version: 1.0
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client
```

A.8.2. Scenario: SAML Bearer Assertion

WS-Trust manages software security tokens. A SAML assertion is a type of security token. In the SAML Bearer scenario, the service provider automatically trusts that the incoming SOAP request came from the subject defined in the SAML token after the service verifies the token's signature.

Implementation of this scenario has the following requirements.

- SAML tokens with a **Bearer** subject confirmation method must be protected so the token can not be snooped. In most cases, a bearer token combined with HTTPS is sufficient to prevent "a man in the middle" getting possession of the token. This means a security policy that uses a **sp:TransportBinding** and **sp:HttpsToken**.
- A bearer token has no encryption or signing keys associated with it, therefore a **sp:IssuedToken** of **bearer** keyType should be used with a **sp:SupportingToken** or a **sp:SignedSupportingTokens**.

A.8.2.1. Web Service Provider

This section examines the web service elements for the SAML Bearer scenario. The components include:

- Bearer Web Service Provider WSDL
- SSL Configuration
- Bearer Web Service Provider Interface
- Bearer Web Service Provider Implementation
- Crypto Properties and Keystore Files
- Default MANIFEST.MF

A.8.2.1.1. Bearer Web Service Provider WSDL

The web service provider is a contract-first endpoint. All the WS-trust and security policies for it are declared in the **BearerService.wsdl** WSDL. For this scenario, a **ws-requester** is required to provide a SAML 2.0 Bearer token issued from a designated STS. The address of the STS is provided in the WSDL.

HTTPS, a **TransportBinding** and **HttpsToken** policy are used to protect the SOAP body of messages that are sent between **ws-requester** and **ws-provider**. The security settings details are provided as comments in the following listing.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
             name="BearerService"
             xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:wsp="http://www.w3.org/ns/ws-policy"
             xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
             xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd"
             xmlns:wsaws="http://www.w3.org/2005/08/addressing"
             xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex"
             xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
             xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">

  <types>
    <xsd:schema>
      <xsd:import namespace="http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
                  schemaLocation="BearerService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <portType name="BearerIface">
    <operation name="sayHello">
      <input message="tns:sayHello"/>
      <output message="tns:sayHelloResponse"/>
    </operation>
  </portType>

<!--
        The wsp:PolicyReference binds the security requirments on all the endpoints.
        The wsp:Policy wsu:Id="#TransportSAML2BearerPolicy" element is defined later in this file.
-->
  <binding name="BearerServicePortBinding" type="tns:BearerIface">
    <wsp:PolicyReference URI="#TransportSAML2BearerPolicy" />
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="sayHello">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
```

```
<!--
  The soap:address has been defined to use JBoss's https port, 8443.  This is
  set in conjunction with the sp:TransportBinding policy for https.
-->
  <service name="BearerService">
    <port name="BearerServicePort" binding="tns:BearerServicePortBinding">
      <soap:address location="https://@jboss.bind.address@:8443/jaxws-samples-wsse-policy-trust-
bearer/BearerService"/>
    </port>
  </service>


  <wsp:Policy wsu:Id="TransportSAML2BearerPolicy">
    <wsp:ExactlyOne>
      <wsp:All>
  <!--
       The wsam:Addressing element, indicates that the endpoints of this
       web service MUST conform to the WS-Addressing specification.  The
       attribute wsp:Optional="false" enforces this assertion.
  -->
       <wsam:Addressing wsp:Optional="false">
         <wsp:Policy />
       </wsam:Addressing>

<!--
  The sp:TransportBinding element indicates that security is provided by the
  message exchange transport medium, https.  WS-Security policy specification
  defines the sp:HttpsToken for use in exchanging messages transmitted over HTTPS.
-->
       <sp:TransportBinding
         xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
         <wsp:Policy>
           <sp:TransportToken>
             <wsp:Policy>
               <sp:HttpsToken>
                 <wsp:Policy/>
               </sp:HttpsToken>
             </wsp:Policy>
           </sp:TransportToken>
<!--
    The sp:AlgorithmSuite element, requires the TripleDes algorithm suite
    be used in performing cryptographic operations.
-->
           <sp:AlgorithmSuite>
             <wsp:Policy>
               <sp:TripleDes />
             </wsp:Policy>
           </sp:AlgorithmSuite>
<!--
    The sp:Layout element,  indicates the layout rules to apply when adding
    items to the security header.  The sp:Lax sub-element indicates items
    are added to the security header in any order that conforms to
    WSS: SOAP Message Security.
-->
           <sp:Layout>
```

```
            <wsp:Policy>
              <sp:Lax />
            </wsp:Policy>
          </sp:Layout>
          <sp:IncludeTimestamp />
        </wsp:Policy>
      </sp:TransportBinding>

<!--
  The sp:SignedSupportingTokens element causes the supporting tokens
  to be signed using the primary token that is used to sign the message.
-->
      <sp:SignedSupportingTokens
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
<!--
  The sp:IssuedToken element asserts that a SAML 2.0 security token of type
  Bearer is expected from the STS.  The
  sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
  attribute instructs the runtime to include the initiator's public key
  with every message sent to the recipient.

  The sp:RequestSecurityTokenTemplate element directs that all of the
  children of this element will be copied directly into the body of the
  RequestSecurityToken (RST) message that is sent to the STS when the
  initiator asks the STS to issue a token.
-->
          <sp:IssuedToken
            sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
            <sp:RequestSecurityTokenTemplate>
              <t:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV2.0</t:TokenType>
              <t:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer</t:KeyType>
            </sp:RequestSecurityTokenTemplate>
            <wsp:Policy>
              <sp:RequireInternalReference />
            </wsp:Policy>
<!--
  The sp:Issuer element defines the STS's address and endpoint information
  This information is used by the STSClient.
-->
            <sp:Issuer>
              <wsaws:Address>http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-sts-
bearer/SecurityTokenService</wsaws:Address>
              <wsaws:Metadata
                xmlns:wsdli="http://www.w3.org/2006/01/wsdl-instance"
                wsdli:wsdlLocation="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-
sts-bearer/SecurityTokenService?wsdl">
                <wsaw:ServiceName
                  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
                  xmlns:stsns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
                  EndpointName="UT_Port">stsns:SecurityTokenService</wsaw:ServiceName>
              </wsaws:Metadata>
            </sp:Issuer>
```

```
          </sp:IssuedToken>
        </wsp:Policy>
      </sp:SignedSupportingTokens>
<!--
   The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
   to be supported by the STS.  These particular elements generally refer
   to how keys are referenced within the SOAP envelope.  These are normally handled by Apache
CXF.
-->
      <sp:Wss11>
        <wsp:Policy>
          <sp:MustSupportRefIssuerSerial />
          <sp:MustSupportRefThumbprint />
          <sp:MustSupportRefEncryptedKey />
        </wsp:Policy>
      </sp:Wss11>
<!--
   The sp:Trust13 element declares controls for WS-Trust 1.3 options.
   They are policy assertions related to exchanges specifically with
   client and server challenges and entropy behaviors.  Again these are
   normally handled by Apache CXF.
-->
      <sp:Trust13>
        <wsp:Policy>
          <sp:MustSupportIssuedTokens />
          <sp:RequireClientEntropy />
          <sp:RequireServerEntropy />
        </wsp:Policy>
      </sp:Trust13>
    </wsp:All>
  </wsp:ExactlyOne>
 </wsp:Policy>

</definitions>
```

A.8.2.1.2. SSL Configuration

This web service is using HTTPS, therefore the JBoss EAP server must be configured to provide SSL support in the **undertow** subsystem.

For information on how to configure HTTPS for web applications, see Configure One-way and Two-way SSL/TLS for Applications in *How to Configure Server Security* .

A.8.2.1.3. Bearer Web Service Providers Interface

The **BearerIface** Bearer Web Service Provider Interface class is a simple web service definition.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.bearer;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
```

```
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
)
public interface BearerIface {
    @WebMethod
    String sayHello();
}
```

A.8.2.1.4. Bearer Web Service Providers Implementation

The **BearerImpl** Web Service Provider Implementation class is a simple POJO. It uses the standard **WebService** annotation to define the service endpoint. In addition there are two Apache CXF annotations, **EndpointProperties** and **EndpointProperty** used for configuring the endpoint for the Apache CXF runtime. These annotations come from the Apache WSS4J project, which provides a Java implementation of the primary WS-Security standards for web services. These annotations are programmatically adding properties to the endpoint. With plain Apache CXF, these properties are often set using the **<jaxws:properties>** element on the **<jaxws:endpoint>** element in the Spring configuration. These annotations allow the properties to be configured in the code.

WSS4J uses the Crypto interface to get keys and certificates for signature creation/verification, as asserted by the WSDL for this service. The WSS4J configuration information being provided by **BearerImpl** is for Crypto's Merlin implementation.

Because the web service provider automatically trusts that the incoming SOAP request that came from the subject defined in the SAML token, it is not required for a Crypto **CallbackHandler** class or a signature username, unlike in prior examples. However, in order to verify the message signature, the Java properties file that contains the (Merlin) Crypto configuration information is still required.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.bearer;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;

import javax.jws.WebService;

@WebService
(
    portName = "BearerServicePort",
    serviceName = "BearerService",
    wsdlLocation = "WEB-INF/wsdl/BearerService.wsdl",
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy",
    endpointInterface = "org.jboss.test.ws.jaxws.samples.wsse.policy.trust.bearer.BearerIface"
)
@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.properties", value = "serviceKeystore.properties")
})
public class BearerImpl implements BearerIface {
    public String sayHello() {
        return "Bearer WS-Trust Hello World!";
    }
}
```

A.8.2.1.5. Crypto Properties and Keystore Files

WSS4J's Crypto implementation is loaded and configured using a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore

location, password, default alias and so on. This application is using the Merlin implementation. The **serviceKeystore.properties** file contains this information.

The **servicestore.jks** file is a Java KeyStore (JKS) repository. It contains self-signed certificates for **myservicekey** and **mystskey**.

NOTE

Self-signed certificates are not appropriate for production use.

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=sspass
org.apache.ws.security.crypto.merlin.keystore.alias=myservicekey
org.apache.ws.security.crypto.merlin.keystore.file=servicestore.jks
```

A.8.2.1.6. Default MANIFEST.MF

When deployed, this application requires access to the JBossWS and Apache CXF APIs provided in module **org.jboss.ws.cxf.jbossws-cxf-client**. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version: 1.0
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client
```

A.8.2.2. Bearer Security Token Service

This section lists the crucial elements in providing the Security Token Service functionality for providing a SAML Bearer token. The components include:

- Security Domain
- STS WSDL
- STS Implementation Class
- STSBearerCallbackHandler Class
- Crypto Properties and Keystore Files
- Default MANIFEST.MF

A.8.2.2.1. Security Domain

STS requires a JBoss security domain be configured. The **jboss-web.xml** descriptor declares a named security domain,**JBossWS-trust-sts** to be used by this service for authentication. This security domain requires two properties files and the addition of a security domain declaration in the JBoss EAP server configuration file.

For this scenario the domain needs to contain user **alice**, password **clarinet**, and role **friend**. Refer to the following listings for **jbossws-users.properties** and **jbossws-roles.properties**. In addition, the following XML must be added to the JBoss **security** subsystem in the server configuration file.

NOTE

Replace "SOME_PATH" with appropriate information.

```
<security-domain name="JBossWS-trust-sts">
  <authentication>
    <login-module code="UsersRoles" flag="required">
      <module-option name="usersProperties" value="/SOME_PATH/jbossws-users.properties"/>
      <module-option name="unauthenticatedIdentity" value="anonymous"/>
      <module-option name="rolesProperties" value="/SOME_PATH/jbossws-roles.properties"/>
    </login-module>
  </authentication>
</security-domain>
```

Example: .jboss-web.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web PUBLIC "-//JBoss//DTD Web Application 2.4//EN" ">
<jboss-web>
  <security-domain>java:/jaas/JBossWS-trust-sts</security-domain>
</jboss-web>
```

Example: .jbossws-users.properties File

```
# A sample users.properties file for use with the UsersRolesLoginModule
alice=clarinet
```

Example: jbossws-roles.properties File

```
# A sample roles.properties file for use with the UsersRolesLoginModule
alice=friend
```

A.8.2.2.2. STS WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  xmlns:wstrust="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsap10="http://www.w3.org/2006/05/addressing/wsdl"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:wsp="http://www.w3.org/ns/ws-policy"
  xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">

  <wsdl:types>
    <xs:schema elementFormDefault="qualified"
          targetNamespace='http://docs.oasis-open.org/ws-sx/ws-trust/200512'>
```

```
    <xs:element name='RequestSecurityToken'
            type='wst:AbstractRequestSecurityTokenType'/>
    <xs:element name='RequestSecurityTokenResponse'
            type='wst:AbstractRequestSecurityTokenType'/>

    <xs:complexType name='AbstractRequestSecurityTokenType'>
     <xs:sequence>
      <xs:any namespace='##any' processContents='lax' minOccurs='0'
            maxOccurs='unbounded'/>
     </xs:sequence>
     <xs:attribute name='Context' type='xs:anyURI' use='optional'/>
     <xs:anyAttribute namespace='##other' processContents='lax'/>
    </xs:complexType>
    <xs:element name='RequestSecurityTokenCollection'
            type='wst:RequestSecurityTokenCollectionType'/>
    <xs:complexType name='RequestSecurityTokenCollectionType'>
     <xs:sequence>
      <xs:element name='RequestSecurityToken'
               type='wst:AbstractRequestSecurityTokenType' minOccurs='2'
               maxOccurs='unbounded'/>
     </xs:sequence>
    </xs:complexType>

    <xs:element name='RequestSecurityTokenResponseCollection'
            type='wst:RequestSecurityTokenResponseCollectionType'/>
    <xs:complexType name='RequestSecurityTokenResponseCollectionType'>
     <xs:sequence>
      <xs:element ref='wst:RequestSecurityTokenResponse' minOccurs='1'
               maxOccurs='unbounded'/>
     </xs:sequence>
     <xs:anyAttribute namespace='##other' processContents='lax'/>
    </xs:complexType>

  </xs:schema>
</wsdl:types>

<!-- WS-Trust defines the following GEDs -->
<wsdl:message name="RequestSecurityTokenMsg">
  <wsdl:part name="request" element="wst:RequestSecurityToken"/>
</wsdl:message>
<wsdl:message name="RequestSecurityTokenResponseMsg">
  <wsdl:part name="response"
          element="wst:RequestSecurityTokenResponse"/>
</wsdl:message>
<wsdl:message name="RequestSecurityTokenCollectionMsg">
  <wsdl:part name="requestCollection"
          element="wst:RequestSecurityTokenCollection"/>
</wsdl:message>
<wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
  <wsdl:part name="responseCollection"
          element="wst:RequestSecurityTokenResponseCollection"/>
</wsdl:message>

<!-- This portType an example of a Requestor (or other) endpoint that
Accepts SOAP-based challenges from a Security Token Service -->
<wsdl:portType name="WSSecurityRequestor">
```

```
  <wsdl:operation name="Challenge">
    <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
    <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
</wsdl:portType>

<!-- This portType is an example of an STS supporting full protocol -->
<!--
    The wsdl:portType and data types are XML elements defined by the
    WS_Trust specification.  The wsdl:portType defines the endpoints
    supported in the STS implementation.  This WSDL defines all operations
    that an STS implementation can support.
-->
<wsdl:portType name="STS">
  <wsdl:operation name="Cancel">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/CancelFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Issue">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTRC/IssueFinal"
      message="tns:RequestSecurityTokenResponseCollectionMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Renew">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/RenewFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Validate">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/ValidateFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="KeyExchangeToken">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KET"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KETFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="RequestCollection">
    <wsdl:input message="tns:RequestSecurityTokenCollectionMsg"/>
```

```
    <wsdl:output message="tns:RequestSecurityTokenResponseCollectionMsg"/>
  </wsdl:operation>
</wsdl:portType>

<!-- This portType is an example of an endpoint that accepts
Unsolicited RequestSecurityTokenResponse messages -->
<wsdl:portType name="SecurityTokenResponseService">
  <wsdl:operation name="RequestSecurityTokenResponse">
    <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
</wsdl:portType>

<!--
    The wsp:PolicyReference binds the security requirments on all the STS endpoints.
    The wsp:Policy wsu:Id="UT_policy" element is later in this file.
-->
<wsdl:binding name="UT_Binding" type="wstrust:STS">
  <wsp:PolicyReference URI="#UT_policy"/>
  <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="Issue">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"/>
    <wsdl:input>
      <wsp:PolicyReference
        URI="#Input_policy"/>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <wsp:PolicyReference
        URI="#Output_policy"/>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="Validate">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate"/>
    <wsdl:input>
      <wsp:PolicyReference
        URI="#Input_policy"/>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <wsp:PolicyReference
        URI="#Output_policy"/>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="Cancel">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
```

```
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="Renew">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="KeyExchangeToken">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KeyExchangeToken"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="RequestCollection">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/RequestCollection"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="SecurityTokenService">
  <wsdl:port name="UT_Port" binding="tns:UT_Binding">
    <soap:address location="http://localhost:8080/SecurityTokenService/UT"/>
  </wsdl:port>
</wsdl:service>


<wsp:Policy wsu:Id="UT_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <!--
          The sp:UsingAddressing element, indicates that the endpoints of this
          web service conforms to the WS-Addressing specification.  More detail
          can be found here: [http://www.w3.org/TR/2006/CR-ws-addr-wsdl-20060529]
      -->
      <wsap10:UsingAddressing/>
      <!--
          The sp:SymmetricBinding element indicates that security is provided
          at the SOAP layer and any initiator must authenticate itself by providing
          WSS UsernameToken credentials.
      -->
      <sp:SymmetricBinding
```

```
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
          <!--
             In a symmetric binding, the keys used for encrypting and signing in both
             directions are derived from a single key, the one specified by the
             sp:ProtectionToken element.  The sp:X509Token sub-element declares this
             key to be a X.509 certificate and the
             IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/Never"
             attribute adds the requirement that the token MUST NOT be included in
             any messages sent between the initiator and the recipient; rather, an
             external reference to the token should be used.  Lastly the WssX509V3Token10
             sub-element declares that the Username token presented by the initiator
             should be compliant with Web Services Security UsernameToken Profile
             1.0 specification. [ http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-
token-profile-1.0.pdf ]
          -->
          <sp:ProtectionToken>
            <wsp:Policy>
              <sp:X509Token
                sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/Never">
                <wsp:Policy>
                  <sp:RequireDerivedKeys/>
                  <sp:RequireThumbprintReference/>
                  <sp:WssX509V3Token10/>
                </wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:ProtectionToken>
          <!--
             The sp:AlgorithmSuite element, requires the Basic256 algorithm suite
             be used in performing cryptographic operations.
          -->
          <sp:AlgorithmSuite>
            <wsp:Policy>
              <sp:Basic256/>
            </wsp:Policy>
          </sp:AlgorithmSuite>
          <!--
             The sp:Layout element,  indicates the layout rules to apply when adding
             items to the security header.  The sp:Lax sub-element indicates items
             are added to the security header in any order that conforms to
             WSS: SOAP Message Security.
          -->
          <sp:Layout>
            <wsp:Policy>
              <sp:Lax/>
            </wsp:Policy>
          </sp:Layout>
          <sp:IncludeTimestamp/>
          <sp:EncryptSignature/>
          <sp:OnlySignEntireHeadersAndBody/>
        </wsp:Policy>
      </sp:SymmetricBinding>
```

```
      <!--
         The sp:SignedSupportingTokens element declares that the security header
         of messages must contain a sp:UsernameToken and the token must be signed.
         The attribute IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient"
         on sp:UsernameToken indicates that the token MUST be included in all
         messages sent from initiator to the recipient and that the token MUST
         NOT be included in messages sent from the recipient to the initiator.
         And finally the element sp:WssUsernameToken10 is a policy assertion
         indicating the Username token should be as defined in  Web Services
         Security UsernameToken Profile 1.0
      -->
      <sp:SignedSupportingTokens
       xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
       <wsp:Policy>
         <sp:UsernameToken
          sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
           <wsp:Policy>
            <sp:WssUsernameToken10/>
           </wsp:Policy>
         </sp:UsernameToken>
       </wsp:Policy>
      </sp:SignedSupportingTokens>
      <!--
         The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
         to be supported by the STS.  These particular elements generally refer
         to how keys are referenced within the SOAP envelope.  These are normally
         handled by Apache CXF.
      -->
      <sp:Wss11
       xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
       <wsp:Policy>
         <sp:MustSupportRefKeyIdentifier/>
         <sp:MustSupportRefIssuerSerial/>
         <sp:MustSupportRefThumbprint/>
         <sp:MustSupportRefEncryptedKey/>
       </wsp:Policy>
      </sp:Wss11>
      <!--
         The sp:Trust13 element declares controls for WS-Trust 1.3 options.
         They are policy assertions related to exchanges specifically with
         client and server challenges and entropy behaviors.  Again these are
         normally handled by Apache CXF.
      -->
      <sp:Trust13
       xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
       <wsp:Policy>
         <sp:MustSupportIssuedTokens/>
         <sp:RequireClientEntropy/>
         <sp:RequireServerEntropy/>
       </wsp:Policy>
      </sp:Trust13>
     </wsp:All>
    </wsp:ExactlyOne>
   </wsp:Policy>
```

```
<wsp:Policy wsu:Id="Input_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SignedParts
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <sp:Body/>
        <sp:Header Name="To"
                Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="From"
                Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="FaultTo"
                Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="ReplyTo"
                Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="MessageID"
                Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="RelatesTo"
                Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="Action"
                Namespace="http://www.w3.org/2005/08/addressing"/>
      </sp:SignedParts>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>

<wsp:Policy wsu:Id="Output_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SignedParts
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <sp:Body/>
        <sp:Header Name="To"
                Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="From"
                Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="FaultTo"
                Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="ReplyTo"
                Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="MessageID"
                Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="RelatesTo"
                Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="Action"
                Namespace="http://www.w3.org/2005/08/addressing"/>
      </sp:SignedParts>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>

</wsdl:definitions>
```

A.8.2.2.3. STS Implementation Class

The Apache CXF's STS, **SecurityTokenServiceProvider**, is a web service provider that is compliant with

the protocols and functionality defined by the WS-Trust specification. It has a modular architecture and its components are configurable or replaceable. There are optional features that are enabled by implementing and configuring plugins. You can customize your own STS by extending from **SecurityTokenServiceProvider** and overriding the default settings.

The **SampleSTSBearer** STS implementation class is a POJO that extends from **SecurityTokenServiceProvider**.

NOTE

The **SampleSTSBearer** class is defined with a **WebServiceProvider** annotation and not a **WebService** annotation. This annotation defines the service as a **Provider**-based endpoint, it supports a messaging-oriented approach to web services. In particular, it signals that the exchanged messages will be XML documents. **SecurityTokenServiceProvider** is an implementation of the **javax.xml.ws.Provider** interface. In comparison the **WebService** annotation defines a service endpoint interface-based endpoint, which supports message exchange using SOAP envelopes.

As done in the **BearerImpl** class, the WSS4J annotations **EndpointProperties** and **EndpointProperty** provide endpoint configuration for the Apache CXF runtime. The first **EndpointProperty** statement in the listing is declaring the user's name to use for the message signature. It is used as the alias name in the keystore to get the user's certificate and private key for signature. The next two **EndpointProperty** statements declare the Java properties file that contains the (Merlin) Crypto configuration information. In this case both for signing and encrypting the messages. WSS4J reads this file and required information for message handling. The last **EndpointProperty** statement declares the **STSBearerCallbackHandler** implementation class. It is used to obtain the user's password for the certificates in the keystore file.

In this implementation we are customizing the operations of token issuance, token validation, and their static properties.

StaticSTSProperties is used to set select properties for configuring resources in STS. This may seem like duplication of the settings made with the WSS4J annotations. The values are the same but the underlaying structures being set are different, thus this information must be declared in both places.

The **setIssuer** setting is important because it uniquely identifies the issuing STS. The issuer string is embedded in issued tokens and, when validating tokens, the STS checks the issuer string value. Consequently, it is important to use the issuer string in a consistent way, so that the STS can recognize the tokens that are issued.

The **setEndpoints** call allows the declaration of a set of allowed token recipients by address. The addresses are specified as reg-ex patterns.

TokenIssueOperation has a modular structure. This allows custom behaviors to be injected into the processing of messages. In this case we are overriding the **SecurityTokenServiceProvider** default behavior and performing SAML token processing. Apache CXF provides an implementation of a **SAMLTokenProvider**, which can be used rather than creating one.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsbearer;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.sts.StaticSTSProperties;
import org.apache.cxf.sts.operation.TokenIssueOperation;
import org.apache.cxf.sts.service.ServiceMBean;
```

```
import org.apache.cxf.sts.service.StaticService;
import org.apache.cxf.sts.token.provider.SAMLTokenProvider;
import org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider;

import javax.xml.ws.WebServiceProvider;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

@WebServiceProvider(serviceName = "SecurityTokenService",
    portName = "UT_Port",
    targetNamespace = "http://docs.oasis-open.org/ws-sx/ws-trust/200512/",
    wsdlLocation = "WEB-INF/wsdl/bearer-ws-trust-1.4-service.wsdl")
//dependency on org.apache.cxf module or on module that exports org.apache.cxf (e.g.
org.jboss.ws.cxf.jbossws-cxf-client) is needed, otherwise Apache CXF annotations are ignored
@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.username", value = "mystskey"),
    @EndpointProperty(key = "ws-security.signature.properties", value = "stsKeystore.properties"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsbearer.STSBearerCallbackHandler")
})
public class SampleSTSBearer extends SecurityTokenServiceProvider {

  public SampleSTSBearer() throws Exception {
    super();

    StaticSTSProperties props = new StaticSTSProperties();
    props.setSignatureCryptoProperties("stsKeystore.properties");
    props.setSignatureUsername("mystskey");
    props.setCallbackHandlerClass(STSBearerCallbackHandler.class.getName());
    props.setEncryptionCryptoProperties("stsKeystore.properties");
    props.setEncryptionUsername("myservicekey");
    props.setIssuer("DoubleItSTSIssuer");

    List<ServiceMBean> services = new LinkedList<ServiceMBean>();
    StaticService service = new StaticService();
    service.setEndpoints(Arrays.asList(
      "https://localhost:(\\d)*/jaxws-samples-wsse-policy-trust-bearer/BearerService",
      "https://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-bearer/BearerService",
      "https://\\[0:0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-bearer/BearerService"
    ));
    services.add(service);

    TokenIssueOperation issueOperation = new TokenIssueOperation();
    issueOperation.getTokenProviders().add(new SAMLTokenProvider());
    issueOperation.setServices(services);
    issueOperation.setStsProperties(props);
    this.setIssueOperation(issueOperation);
  }
}
```

A.8.2.2.4. STSBearerCallbackHandler Class

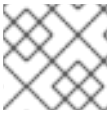
STSBearerCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables Apache CXF to retrieve the password of the user name to use for the message signature.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsbearer;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;

import java.util.HashMap;
import java.util.Map;

public class STSBearerCallbackHandler extends PasswordCallbackHandler {
  public STSBearerCallbackHandler() {
    super(getInitMap());
  }

  private static Map<String, String> getInitMap() {
    Map<String, String> passwords = new HashMap<String, String>();
    passwords.put("mystskey", "stskpass");
    passwords.put("alice", "clarinet");
    return passwords;
  }
}
```

A.8.2.2.5. Crypto Properties and Keystore Files

WSS4J's Crypto implementation is loaded and configured using a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and so on. This application is using the Merlin implementation. The **stsKeystore.properties** file contains this information.

The **servicestore.jks** file is a Java KeyStore (JKS) repository. It contains self-signed certificates for **myservicekey** and **mystskey**.



NOTE

Self-signed certificates are not appropriate for production use.

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=stsspass
org.apache.ws.security.crypto.merlin.keystore.file=stsstore.jks
```

A.8.2.2.6. Default MANIFEST.MF

This application requires access to the JBossWS and Apache CXF APIs provided in the **org.jboss.ws.cxf.jbossws-cxf-client** module. The **org.jboss.ws.cxf.sts** module is also needed to build the STS configuration in the **SampleSTS** constructor. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version: 1.0
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client,org.jboss.ws.cxf.sts
```

A.8.2.3. Web Service Requester

This section provides the details of crucial elements in calling a web service that implements endpoint security as described in the SAML Bearer scenario. The components that will be discussed include:

- Web Service Requester Implementation
- ClientCallbackHandler
- Crypto Properties and Keystore Files

A.8.2.3.1. Web Service Requester Implementation

The **ws-requester**, the client, uses standard procedures for creating a reference to the web service. To address the endpoint security requirements, the web service's "Request Context" is configured with the information required for message generation. In addition, the **STSClient** that communicates with the STS is configured with similar values.

NOTE

The key strings ending with a **.it** suffix flags these settings as belonging to the **STSClient**. The internal Apache CXF code assigns this information to the **STSClient** that is auto-generated for this service call.

There is an alternate method of setting up the **STSCLient**. The user may provide their own instance of the **STSClient**. The Apache CXF code uses this object and does not auto-generate one. When providing the **STSClient** in this way, the user must provide a **org.apache.cxf.Bus** for it and the configuration keys must not have the **.it** suffix. This is used in the ActAs and OnBehalfOf examples.

```
String serviceURL = "https://" + getServerHost() + ":8443/jaxws-samples-wsse-policy-trust-
bearer/BearerService";

final QName serviceName = new QName("http://www.jboss.org/jbossws/ws-
extensions/bearerwssecuritypolicy", "BearerService");
Service service = Service.create(new URL(serviceURL + "?wsdl"), serviceName);
BearerIface proxy = (BearerIface) service.getPort(BearerIface.class);

Map<String, Object> ctx = ((BindingProvider)proxy).getRequestContext();

// set the security related configuration information for the service "request"
ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
  Thread.currentThread().getContextClassLoader().getResource(
  "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
  Thread.currentThread().getContextClassLoader().getResource(
  "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");
ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");

//-- Configuration settings that will be transfered to the STSClient
// "alice" is the name provided for the WSS Username. Her password will
// be retreived from the ClientCallbackHander by the STSClient.
ctx.put(SecurityConstants.USERNAME + ".it", "alice");
ctx.put(SecurityConstants.CALLBACK_HANDLER + ".it", new ClientCallbackHandler());
ctx.put(SecurityConstants.ENCRYPT_PROPERTIES + ".it",
  Thread.currentThread().getContextClassLoader().getResource(
  "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.ENCRYPT_USERNAME + ".it", "mystskey");
ctx.put(SecurityConstants.STS_TOKEN_USERNAME + ".it", "myclientkey");
```

```
ctx.put(SecurityConstants.STS_TOKEN_PROPERTIES + ".it",
  Thread.currentThread().getContextClassLoader().getResource(
  "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO + ".it", "true");

proxy.sayHello();
```

A.8.2.3.2. ClientCallbackHandler

ClientCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables Apache CXF to retrieve the password of the user name to use for the message signature.

NOTE

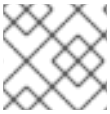
The user **alice** and password have been provided here. This information is not in the (JKS) keystore but provided in the security domain. It is declared in **jbossws-users.properties** file.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class ClientCallbackHandler implements CallbackHandler {

   public void handle(Callback[] callbacks) throws IOException,
         UnsupportedCallbackException {
      for (int i = 0; i < callbacks.length; i++) {
         if (callbacks[i] instanceof WSPasswordCallback) {
            WSPasswordCallback pc = (WSPasswordCallback) callbacks[i];
            if ("myclientkey".equals(pc.getIdentifier())) {
               pc.setPassword("ckpass");
               break;
            } else if ("alice".equals(pc.getIdentifier())) {
               pc.setPassword("clarinet");
               break;
            } else if ("bob".equals(pc.getIdentifier())) {
               pc.setPassword("trombone");
               break;
            } else if ("myservicekey".equals(pc.getIdentifier())) {  // rls test  added for bearer test
              pc.setPassword("skpass");
              break;
            }
         }
      }
   }
}
```

A.8.2.3.3. Crypto Properties and Keystore Files

WSS4J's Crypto implementation is loaded and configured using a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and so on. This application is using the Merlin implementation. The **clientKeystore.properties** file contains this information.

The **clientstore.jks** file is a Java KeyStore (JKS) repository. It contains self-signed certificates for **myservicekey** and **mystskey**.



NOTE

Self-signed certificates are not appropriate for production use.

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=cspass
org.apache.ws.security.crypto.merlin.keystore.alias=myclientkey
org.apache.ws.security.crypto.merlin.keystore.file=META-INF/clientstore.jks
```

A.8.3. Scenario: OnBehalfOf WS-Trust

The **OnBehalfOf** feature is used in scenarios that use the proxy pattern. In such scenarios, the client cannot access the STS directly, instead it communicates through a proxy gateway. The proxy gateway authenticates the caller and puts information about the caller into the **OnBehalfOf** element of the **RequestSecurityToken** (RST) sent to the real STS for processing. The resulting token contains only claims related to the client of the proxy, making the proxy completely transparent to the receiver of the issued token.

OnBehalfOf is nothing more than a new sub-element in the RST. It provides additional information about the original caller when a token is negotiated with the STS. The **OnBehalfOf** element usually takes the form of a token with identity claims such as name, role, and authorization code, for the client to access the service.

The **OnBehalfOf** scenario is an extension of the basic WS-Trust scenario. In this example the **OnBehalfOf** service calls the **ws-service** on behalf of a user. There are only a couple of additions to the basic scenario's code. An **OnBehalfOf** web service provider and callback handler have been added. The **OnBehalfOf** web services' WSDL imposes the same security policies as the **ws-provider**. **UsernameTokenCallbackHandler** is a utility shared with **ActAs**. It generates the content for the **OnBehalfOf** element. Lastly, there are code additions in the STS that both **OnBehalfOf** and **ActAs** share in common.

A.8.3.1. Web Service Provider

This section provides the web service elements from the basic WS-Trust scenario that have been updated to address the requirements of the **OnBehalfOf** example. The components include:

- Web Service Provider WSDL
- Web Service Provider Interface
- Web Service Provider Implementation
- OnBehalfOfCallbackHandler Class

A.8.3.1.1. Web Service Provider WSDL

The **OnBehalfOf** web service provider's WSDL is a clone of the **ws-provider's** WSDL. The **wsp:Policy** section is the same. There are updates to the service endpoint, **targetNamespace**, **portType**, **binding** name, and **service**.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-
extensions/onbehalfofwssecuritypolicy" name="OnBehalfOfService"
             xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/onbehalfofwssecuritypolicy"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:wsp="http://www.w3.org/ns/ws-policy"
             xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
             xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd"
             xmlns:wsaws="http://www.w3.org/2005/08/addressing"
             xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
             xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">
   <types>
      <xsd:schema>
         <xsd:import namespace="http://www.jboss.org/jbossws/ws-
extensions/onbehalfofwssecuritypolicy"
                schemaLocation="OnBehalfOfService_schema1.xsd"/>
      </xsd:schema>
   </types>
   <message name="sayHello">
      <part name="parameters" element="tns:sayHello"/>
   </message>
   <message name="sayHelloResponse">
      <part name="parameters" element="tns:sayHelloResponse"/>
   </message>
   <portType name="OnBehalfOfServiceIface">
      <operation name="sayHello">
         <input message="tns:sayHello"/>
         <output message="tns:sayHelloResponse"/>
      </operation>
   </portType>
   <binding name="OnBehalfOfServicePortBinding" type="tns:OnBehalfOfServiceIface">
      <wsp:PolicyReference URI="#AsymmetricSAML2Policy" />
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
      <operation name="sayHello">
         <soap:operation soapAction=""/>
         <input>
            <soap:body use="literal"/>
            <wsp:PolicyReference URI="#Input_Policy" />
         </input>
         <output>
            <soap:body use="literal"/>
            <wsp:PolicyReference URI="#Output_Policy" />
         </output>
      </operation>
   </binding>
   <service name="OnBehalfOfService">
      <port name="OnBehalfOfServicePort" binding="tns:OnBehalfOfServicePortBinding">
         <soap:address location="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-
onbehalfof/OnBehalfOfService"/>
```

```
    </port>
  </service>
</definitions>
```

A.8.3.1.2. Web Service Provider Interface

The **OnBehalfOfServiceIface** web service provider interface class is a simple web service definition.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
  targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/onbehalfofwssecuritypolicy"
)
public interface OnBehalfOfServiceIface {
  @WebMethod
  String sayHello();
}
```

A.8.3.1.3. Web Service Provider Implementation

The **OnBehalfOfServiceImpl** web service provider implementation class is a simple POJO. It uses the standard **WebService** annotation to define the service endpoint and two Apache WSS4J annotations, **EndpointProperties** and **EndpointProperty** used for configuring the endpoint for the Apache CXF runtime. The WSS4J configuration information provided is for WSS4J's Crypto Merlin implementation.

OnBehalfOfServiceImpl calls the **ServiceImpl** acting on behalf of the user. The **setupService** method performs the required configuration setup.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof;

import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.ws.security.SecurityConstants;
import org.apache.cxf.ws.security.trust.STSClient;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service.ServiceIface;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared.WSTrustAppUtils;

import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Service;
import java.net.*;
import java.util.Map;

@WebService
(
  portName = "OnBehalfOfServicePort",
  serviceName = "OnBehalfOfService",
  wsdlLocation = "WEB-INF/wsdl/OnBehalfOfService.wsdl",
```

```
  targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/onbehalfofwssecuritypolicy",
  endpointInterface =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof.OnBehalfOfServiceIface"
)

@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.username", value = "myactaskey"),
    @EndpointProperty(key = "ws-security.signature.properties", value =  "actasKeystore.properties"),
    @EndpointProperty(key = "ws-security.encryption.properties", value =
"actasKeystore.properties"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof.OnBehalfOfCallbackHandler")
})

public class OnBehalfOfServiceImpl implements OnBehalfOfServiceIface {
  public String sayHello() {
    try {

      ServiceIface proxy = setupService();
      return "OnBehalfOf " + proxy.sayHello();

    } catch (MalformedURLException e) {
      e.printStackTrace();
    }
    return null;
  }

  /**
   *
   * @return
   * @throws MalformedURLException
   */
  private  ServiceIface setupService()throws MalformedURLException {
    ServiceIface proxy = null;
    Bus bus = BusFactory.newInstance().createBus();

    try {
      BusFactory.setThreadDefaultBus(bus);

      final String serviceURL = "http://" + WSTrustAppUtils.getServerHost() + ":8080/jaxws-samples-
wsse-policy-trust/SecurityService";
      final QName serviceName = new QName("http://www.jboss.org/jbossws/ws-
extensions/wssecuritypolicy", "SecurityService");
      final URL wsdlURL = new URL(serviceURL + "?wsdl");
      Service service = Service.create(wsdlURL, serviceName);
      proxy = (ServiceIface) service.getPort(ServiceIface.class);

      Map<String, Object> ctx = ((BindingProvider) proxy).getRequestContext();
      ctx.put(SecurityConstants.CALLBACK_HANDLER, new OnBehalfOfCallbackHandler());

      ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
        "actasKeystore.properties" ));
      ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myactaskey" );
      ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
```

```
            "../../META-INF/clientKeystore.properties" ));
        ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");

        STSClient stsClient = new STSClient(bus);
        Map<String, Object> props = stsClient.getProperties();
        props.put(SecurityConstants.USERNAME, "bob");
        props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
        props.put(SecurityConstants.STS_TOKEN_USERNAME, "myactaskey" );
        props.put(SecurityConstants.STS_TOKEN_PROPERTIES,
          Thread.currentThread().getContextClassLoader().getResource(
          "actasKeystore.properties" ));
        props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

        ctx.put(SecurityConstants.STS_CLIENT, stsClient);

      } finally {
        bus.shutdown(true);
      }

      return proxy;
    }

}
```

A.8.3.1.4. OnBehalfOfCallbackHandler Class

The **OnBehalfOfCallbackHandler** is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables Apache CXF to retrieve the password of the user name to use for the message signature. This class has been updated to return the passwords for this service, **myactaskey** and the **OnBehalfOf** user, **alice**.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;
import java.util.HashMap;
import java.util.Map;

public class OnBehalfOfCallbackHandler extends PasswordCallbackHandler {

  public OnBehalfOfCallbackHandler() {
    super(getInitMap());
  }

  private static Map<String, String> getInitMap() {
    Map<String, String> passwords = new HashMap<String, String>();
    passwords.put("myactaskey", "aspass");
    passwords.put("alice", "clarinet");
    passwords.put("bob", "trombone");
    return passwords;
  }

}
```

A.8.3.2. Web Service Requester

This section provides details of the **ws-requester** elements from the basic WS-Trust scenario that have been updated to address the requirements of the **OnBehalfOf** example. The component is:

- OnBehalfOf Web Service Requester Implementation Class

A.8.3.2.1. OnBehalfOf Web Service Requester Implementation Class

The **OnBehalfOf ws-requester**, the client, uses standard procedures for creating a reference to the web service in the first four lines. To address the endpoint security requirements, the web service's request context is configured using the **BindingProvider**. Information needed in the message generation is provided through it. The **OnBehalfOf** user, **alice**, is declared in this section and the **callbackHandler**, **UsernameTokenCallbackHandler** is provided to the **STSClient** for generation of the contents for the **OnBehalfOf** message element. In this example an **STSClient** object is created and provided to the proxy's request context. The alternative is to provide keys tagged with the **.it** suffix as done in the Basic Scenario client. The use of **OnBehalfOf** is configured by the **stsClient.setOnBehalfOf** call method. The alternative is to use the key **SecurityConstants.STS_TOKEN_ON_BEHALF_OF** and a value in the properties map.

```
final QName serviceName = new QName("http://www.jboss.org/jbossws/ws-
extensions/onbehalfofwssecuritypolicy", "OnBehalfOfService");
final URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
OnBehalfOfServiceIface proxy = (OnBehalfOfServiceIface)
service.getPort(OnBehalfOfServiceIface.class);


Bus bus = BusFactory.newInstance().createBus();
try {

   BusFactory.setThreadDefaultBus(bus);

   Map<String, Object> ctx = proxy.getRequestContext();

   ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
   ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
      Thread.currentThread().getContextClassLoader().getResource(
      "META-INF/clientKeystore.properties"));
   ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myactaskey");
   ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
      Thread.currentThread().getContextClassLoader().getResource(
      "META-INF/clientKeystore.properties"));
   ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");

   // user and password OnBehalfOf user
   // UsernameTokenCallbackHandler will extract this information when called
   ctx.put(SecurityConstants.USERNAME,"alice");
   ctx.put(SecurityConstants.PASSWORD, "clarinet");

   STSClient stsClient = new STSClient(bus);

   // Providing the STSClient the mechanism to create the claims contents for OnBehalfOf
   stsClient.setOnBehalfOf(new UsernameTokenCallbackHandler());

   Map<String, Object> props = stsClient.getProperties();
   props.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
   props.put(SecurityConstants.ENCRYPT_PROPERTIES,
```

```
        Thread.currentThread().getContextClassLoader().getResource(
        "META-INF/clientKeystore.properties"));
    props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
    props.put(SecurityConstants.STS_TOKEN_USERNAME, "myclientkey");
    props.put(SecurityConstants.STS_TOKEN_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
        "META-INF/clientKeystore.properties"));
    props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

    ctx.put(SecurityConstants.STS_CLIENT, stsClient);

} finally {
    bus.shutdown(true);
}
proxy.sayHello();
```

A.8.4. Scenario: ActAs WS-Trust

The **ActAs** feature is used in scenarios that require composite delegation. It is commonly used in multi-tiered systems where an application calls a service on behalf of a logged in user, or a service calls another service on behalf of the original caller.

ActAs is nothing more than a new sub-element in the **RequestSecurityToken** (RST). It provides additional information about the original caller when a token is negotiated with the STS. The **ActAs** element usually takes the form of a token with identity claims such as name, role, and authorization code, for the client to access the service.

The **ActAs** scenario is an extension of the basic WS-Trust scenario. In this example the **ActAs** service calls the **ws-service** on behalf of a user. There are only a couple of additions to the basic scenario's code. An **ActAs** web service provider and callback handler have been added. The **ActAs** web services' WSDL imposes the same security policies as the **ws-provider**. **UsernameTokenCallbackHandler** is a new utility that generates the content for the **ActAs** element. Lastly, there are a couple of code additions in the STS to support the **ActAs** request.

A.8.4.1. Web Service Provider

This section provides details about the web service elements from the basic WS-Trust scenario that have been changed to address the needs of the **ActAs** example. The components include:

- ActAs Web Service Provider WSDL
- ActAs Web Service Provider Interface
- ActAs Web Service Provider Implementation
- ActAsCallbackHandler Class
- UsernameTokenCallbackHandler
- Crypto properties and Keystore Files
- Default MANIFEST.MF

A.8.4.1.1. Web Service Provider WSDL

The **ActAs** web service provider's WSDL is a clone of the **ws-provider's** WSDL. The **wsp:Policy** section is the same. There are changes to the service endpoint, **targetNamespace**, **portType**, **binding** name, and **service**.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
name="ActAsService"
          xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/"
          xmlns:wsp="http://www.w3.org/ns/ws-policy"
          xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
          xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd"
          xmlns:wsaws="http://www.w3.org/2005/08/addressing"
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
          xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">
   <types>
      <xsd:schema>
         <xsd:import namespace="http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
               schemaLocation="ActAsService_schema1.xsd"/>
      </xsd:schema>
   </types>
   <message name="sayHello">
      <part name="parameters" element="tns:sayHello"/>
   </message>
   <message name="sayHelloResponse">
      <part name="parameters" element="tns:sayHelloResponse"/>
   </message>
   <portType name="ActAsServiceIface">
      <operation name="sayHello">
         <input message="tns:sayHello"/>
         <output message="tns:sayHelloResponse"/>
      </operation>
   </portType>
   <binding name="ActAsServicePortBinding" type="tns:ActAsServiceIface">
      <wsp:PolicyReference URI="#AsymmetricSAML2Policy" />
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
      <operation name="sayHello">
         <soap:operation soapAction=""/>
         <input>
            <soap:body use="literal"/>
            <wsp:PolicyReference URI="#Input_Policy" />
         </input>
         <output>
            <soap:body use="literal"/>
            <wsp:PolicyReference URI="#Output_Policy" />
         </output>
      </operation>
   </binding>
   <service name="ActAsService">
      <port name="ActAsServicePort" binding="tns:ActAsServicePortBinding">
         <soap:address location="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-
actas/ActAsService"/>
      </port>
```

```
    </service>

</definitions>
```

A.8.4.1.2. Web Service Provider Interface

The **ActAsServiceIface** web service provider interface class is a simple web service definition.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
  targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
)
public interface ActAsServiceIface {
  @WebMethod
  String sayHello();
}
```

A.8.4.1.3. Web Service Provider Implementation

The **ActAsServiceImpl** web service provider implementation class is a simple POJO. It uses the standard **WebService** annotation to define the service endpoint and two Apache WSS4J annotations, **EndpointProperties**, and **EndpointProperty**, used for configuring the endpoint for the Apache CXF runtime. The WSS4J configuration information provided is for WSS4J's Crypto Merlin implementation.

ActAsServiceImpl is calling **ServiceImpl** acting on behalf of the user. The **setupService** method performs the required configuration setup.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas;

import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.ws.security.SecurityConstants;
import org.apache.cxf.ws.security.trust.STSClient;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service.ServiceIface;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared.WSTrustAppUtils;

import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Service;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Map;

@WebService
(
  portName = "ActAsServicePort",
  serviceName = "ActAsService",
```

```
  wsdlLocation = "WEB-INF/wsdl/ActAsService.wsdl",
  targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy",
  endpointInterface = "org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas.ActAsServiceIface"
)

@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.username", value = "myactaskey"),
    @EndpointProperty(key = "ws-security.signature.properties", value =  "actasKeystore.properties"),
    @EndpointProperty(key = "ws-security.encryption.properties", value =
"actasKeystore.properties"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas.ActAsCallbackHandler")
})

public class ActAsServiceImpl implements ActAsServiceIface {
  public String sayHello() {
    try {
      ServiceIface proxy = setupService();
      return "ActAs " + proxy.sayHello();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    }
    return null;
  }

  private  ServiceIface setupService()throws MalformedURLException {
    ServiceIface proxy = null;
    Bus bus = BusFactory.newInstance().createBus();

    try {
      BusFactory.setThreadDefaultBus(bus);

      final String serviceURL = "http://" + WSTrustAppUtils.getServerHost() + ":8080/jaxws-samples-
wsse-policy-trust/SecurityService";
      final QName serviceName = new QName("http://www.jboss.org/jbossws/ws-
extensions/wssecuritypolicy", "SecurityService");
      final URL wsdlURL = new URL(serviceURL + "?wsdl");
      Service service = Service.create(wsdlURL, serviceName);
      proxy = (ServiceIface) service.getPort(ServiceIface.class);

      Map<String, Object> ctx = ((BindingProvider) proxy).getRequestContext();
      ctx.put(SecurityConstants.CALLBACK_HANDLER, new ActAsCallbackHandler());

      ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource("actasKeystore.properties" ));
      ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myactaskey" );
      ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource("../../META-
INF/clientKeystore.properties" ));
      ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");

      STSClient stsClient = new STSClient(bus);
      Map<String, Object> props = stsClient.getProperties();
      props.put(SecurityConstants.USERNAME, "alice");
      props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
      props.put(SecurityConstants.STS_TOKEN_USERNAME, "myactaskey" );
```

```
        props.put(SecurityConstants.STS_TOKEN_PROPERTIES,
          Thread.currentThread().getContextClassLoader().getResource("actasKeystore.properties" ));
        props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

        ctx.put(SecurityConstants.STS_CLIENT, stsClient);

      } finally {
        bus.shutdown(true);
      }

      return proxy;
    }

}
```

A.8.4.1.4. ActAsCallbackHandler Class

ActAsCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables Apache CXF to retrieve the password of the user name to use for the message signature. This class has been updated to return the passwords for this service, **myactaskey** and the **ActAs** user, **alice**.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;
import java.util.HashMap;
import java.util.Map;

public class ActAsCallbackHandler extends PasswordCallbackHandler {

  public ActAsCallbackHandler() {
    super(getInitMap());
  }

  private static Map<String, String> getInitMap() {
    Map<String, String> passwords = new HashMap<String, String>();
    passwords.put("myactaskey", "aspass");
    passwords.put("alice", "clarinet");
    return passwords;
  }
}
```

A.8.4.1.5. UsernameTokenCallbackHandler

The **ActAs** and **OnBeholdOf** sub-elements of the **RequestSecurityToken** have to be defined as WSSE **UsernameTokens**. This utility generates the properly formatted element.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared;

import org.apache.cxf.helpers.DOMUtils;
import org.apache.cxf.message.Message;
import org.apache.cxf.ws.security.SecurityConstants;
import org.apache.cxf.ws.security.trust.delegation.DelegationCallback;
import org.apache.ws.security.WSConstants;
```

```
import org.apache.ws.security.message.token.UsernameToken;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
import org.w3c.dom.ls.DOMImplementationLS;
import org.w3c.dom.ls.LSSerializer;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import java.io.IOException;
import java.util.Map;

/**
* A utility to provide the 3 different input parameter types for jaxws property
* "ws-security.sts.token.act-as" and "ws-security.sts.token.on-behalf-of".
* This implementation obtains a username and password via the jaxws property
* "ws-security.username" and "ws-security.password" respectively, as defined
* in SecurityConstants.  It creates a wss UsernameToken to be used as the
* delegation token.
*/

public class UsernameTokenCallbackHandler implements CallbackHandler {

  public void handle(Callback[] callbacks)
    throws IOException, UnsupportedCallbackException {
    for (int i = 0; i < callbacks.length; i++) {
      if (callbacks[i] instanceof DelegationCallback) {
        DelegationCallback callback = (DelegationCallback) callbacks[i];
        Message message = callback.getCurrentMessage();

        String username =
          (String)message.getContextualProperty(SecurityConstants.USERNAME);
        String password =
          (String)message.getContextualProperty(SecurityConstants.PASSWORD);
        if (username != null) {
          Node contentNode = message.getContent(Node.class);
          Document doc = null;
          if (contentNode != null) {
            doc = contentNode.getOwnerDocument();
          } else {
            doc = DOMUtils.createDocument();
          }
          UsernameToken usernameToken = createWSSEUsernameToken(username,password,
doc);
          callback.setToken(usernameToken.getElement());
        }
      } else {
        throw new UnsupportedCallbackException(callbacks[i], "Unrecognized Callback");
      }
    }
  }

  /**
   * Provide UsernameToken as a string.
   * @param ctx
```

```
     * @return
     */
    public String getUsernameTokenString(Map<String, Object> ctx){
        Document doc = DOMUtils.createDocument();
        String result = null;
        String username = (String)ctx.get(SecurityConstants.USERNAME);
        String password = (String)ctx.get(SecurityConstants.PASSWORD);
        if (username != null) {
            UsernameToken usernameToken = createWSSEUsernameToken(username,password, doc);
            result = toString(usernameToken.getElement().getFirstChild().getParentNode());
        }
        return result;
    }

    /**
     *
     * @param username
     * @param password
     * @return
     */
    public String getUsernameTokenString(String username, String password){
        Document doc = DOMUtils.createDocument();
        String result = null;
        if (username != null) {
            UsernameToken usernameToken = createWSSEUsernameToken(username,password, doc);
            result = toString(usernameToken.getElement().getFirstChild().getParentNode());
        }
        return result;
    }

    /**
     * Provide UsernameToken as a DOM Element.
     * @param ctx
     * @return
     */
    public Element getUsernameTokenElement(Map<String, Object> ctx){
        Document doc = DOMUtils.createDocument();
        Element result = null;
        UsernameToken usernameToken = null;
            String username = (String)ctx.get(SecurityConstants.USERNAME);
        String password = (String)ctx.get(SecurityConstants.PASSWORD);
        if (username != null) {
            usernameToken = createWSSEUsernameToken(username,password, doc);
            result = usernameToken.getElement();
        }
        return result;
    }

    /**
     *
     * @param username
     * @param password
     * @return
     */
    public Element getUsernameTokenElement(String username, String password){
        Document doc = DOMUtils.createDocument();
```

```
    Element result = null;
    UsernameToken usernameToken = null;
    if (username != null) {
      usernameToken = createWSSEUsernameToken(username,password, doc);
      result = usernameToken.getElement();
    }
    return result;
  }

  private UsernameToken createWSSEUsernameToken(String username, String password,
Document doc) {

    UsernameToken usernameToken = new UsernameToken(true, doc,
      (password == null)? null: WSConstants.PASSWORD_TEXT);
    usernameToken.setName(username);
    usernameToken.addWSUNamespace();
    usernameToken.addWSSENamespace();
    usernameToken.setID("id-" + username);

    if (password != null){
      usernameToken.setPassword(password);
    }

    return usernameToken;
  }


  private String toString(Node node) {
    String str = null;

    if (node != null) {
      DOMImplementationLS lsImpl = (DOMImplementationLS)
        node.getOwnerDocument().getImplementation().getFeature("LS", "3.0");
      LSSerializer serializer = lsImpl.createLSSerializer();
      serializer.getDomConfig().setParameter("xml-declaration", false); //by default its true, so set it to
false to get String without xml-declaration
      str = serializer.writeToString(node);
    }
    return str;
  }

}
```

A.8.4.1.6. Crypto properties and keystore files

The **ActAs** service must provide its own credentials. The requisite **actasKeystore.properties** properties file and **actasstore.jks** keystore are created.

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=aapass
org.apache.ws.security.crypto.merlin.keystore.alias=myactaskey
org.apache.ws.security.crypto.merlin.keystore.file=actasstore.jks
```

A.8.4.1.7. Default MANIFEST.MF

This application requires access to the JBossWS and Apache CXF APIs provided in the **org.jboss.ws.cxf.jbossws-cxf-client** module. The **org.jboss.ws.cxf.sts** module is also needed in handling the **ActAs** and **OnBehalfOf** extensions. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version: 1.0
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client, org.jboss.ws.cxf.sts
```

A.8.4.2. Security Token Service

This section provides the details of the STS elements from the basic WS-Trust scenario that have been changed to address the needs of the **ActAs** example. The components include:

- STS Implementation Class
- STSCallbackHandler Class

A.8.4.2.1. STS Implementation Class

The declaration of the set of allowed token recipients by address has been extended to accept **ActAs** addresses and **OnBehalfOf** addresses. The addresses are specified as reg-ex patterns.

The **TokenIssueOperation** requires the **UsernameTokenValidator** class to be provided to validate the contents of the **OnBehalfOf**, and the **UsernameTokenDelegationHandler** class to be provided to process the token delegation request of the **ActAs** on **OnBehalfOf** user.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.sts;

import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

import javax.xml.ws.WebServiceProvider;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.interceptor.InInterceptors;
import org.apache.cxf.sts.StaticSTSProperties;
import org.apache.cxf.sts.operation.TokenIssueOperation;
import org.apache.cxf.sts.operation.TokenValidateOperation;
import org.apache.cxf.sts.service.ServiceMBean;
import org.apache.cxf.sts.service.StaticService;
import org.apache.cxf.sts.token.delegation.UsernameTokenDelegationHandler;
import org.apache.cxf.sts.token.provider.SAMLTokenProvider;
import org.apache.cxf.sts.token.validator.SAMLTokenValidator;
import org.apache.cxf.sts.token.validator.UsernameTokenValidator;
import org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider;

@WebServiceProvider(serviceName = "SecurityTokenService",
    portName = "UT_Port",
    targetNamespace = "http://docs.oasis-open.org/ws-sx/ws-trust/200512/",
    wsdlLocation = "WEB-INF/wsdl/ws-trust-1.4-service.wsdl")
//dependency on org.apache.cxf module or on module that exports org.apache.cxf (e.g.
org.jboss.ws.cxf.jbossws-cxf-client) is needed, otherwise Apache CXF annotations are ignored
@EndpointProperties(value = {
```

```
    @EndpointProperty(key = "ws-security.signature.username", value = "mystskey"),
    @EndpointProperty(key = "ws-security.signature.properties", value = "stsKeystore.properties"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.sts.STSCallbackHandler"),
    @EndpointProperty(key = "ws-security.validate.token", value = "false") //to let the JAAS
integration deal with validation through the interceptor below
})
@InInterceptors(interceptors =
{"org.jboss.wsf.stack.cxf.security.authentication.SubjectCreatingPolicyInterceptor"})
public class SampleSTS extends SecurityTokenServiceProvider {
  public SampleSTS() throws Exception {
    super();

    StaticSTSProperties props = new StaticSTSProperties();
    props.setSignatureCryptoProperties("stsKeystore.properties");
    props.setSignatureUsername("mystskey");
    props.setCallbackHandlerClass(STSCallbackHandler.class.getName());
    props.setIssuer("DoubleItSTSIssuer");

    List<ServiceMBean> services = new LinkedList<ServiceMBean>();
    StaticService service = new StaticService();
    service.setEndpoints(Arrays.asList(
      "http://localhost:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",
      "http://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",
      "http://\\[0:0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",

      "http://localhost:(\\d)*/jaxws-samples-wsse-policy-trust-actas/ActAsService",
      "http://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-actas/ActAsService",
      "http://\\[0:0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-actas/ActAsService",

      "http://localhost:(\\d)*/jaxws-samples-wsse-policy-trust-onbehalfof/OnBehalfOfService",
      "http://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-onbehalfof/OnBehalfOfService",
      "http://\\[0:0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-onbehalfof/OnBehalfOfService"
    ));
    services.add(service);

    TokenIssueOperation issueOperation = new TokenIssueOperation();
    issueOperation.setServices(services);
    issueOperation.getTokenProviders().add(new SAMLTokenProvider());
    // required for OnBehalfOf
    issueOperation.getTokenValidators().add(new UsernameTokenValidator());
    // added for OnBehalfOf and ActAs
    issueOperation.getDelegationHandlers().add(new UsernameTokenDelegationHandler());
    issueOperation.setStsProperties(props);

    TokenValidateOperation validateOperation = new TokenValidateOperation();
    validateOperation.getTokenValidators().add(new SAMLTokenValidator());
    validateOperation.setStsProperties(props);

    this.setIssueOperation(issueOperation);
    this.setValidateOperation(validateOperation);
  }
}
```

A.8.4.2.2. STSCallbackHandler Class

The user, **alice**, and corresponding password was required to be added for the **ActAs** example.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.sts;

import java.util.HashMap;
import java.util.Map;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;

public class STSCallbackHandler extends PasswordCallbackHandler {
  public STSCallbackHandler() {
    super(getInitMap());
  }

  private static Map<String, String> getInitMap() {
    Map<String, String> passwords = new HashMap<String, String>();
    passwords.put("mystskey", "stskpass");
    passwords.put("alice", "clarinet");
    return passwords;
  }
}
```

A.8.4.2.3. Web Service Requester

This section provides the details of the **ws-requester** elements from the basic WS-Trust scenario that have been changed to address the requirements of the **ActAs** example. The component is:

- ActAs Web Service Requester Implementation Class

A.8.4.2.4. Web Service Requester Implementation Class

The **ActAs ws-requester**, the client, uses standard procedures for creating a reference to the web service in the first four lines. To address the endpoint security requirements, the web service's request context is configured using **BindingProvider** to provide information required for message generation. The **ActAs** user, **myactaskey**, is declared in this section and **UsernameTokenCallbackHandler** is used to provide the contents of the **ActAs** element to the **STSClient**. In this example an **STSClient** object is created and provided to the proxy's request context. The alternative is to provide keys tagged with the **.it** suffix as was done in the Basic Scenario client. The use of **ActAs** is configured through the properties map using the **SecurityConstants.STS_TOKEN_ACT_AS** key. The alternative is to use the **STSClient.setActAs** method.

```
final QName serviceName = new QName("http://www.jboss.org/jbossws/ws-
extensions/actaswssecuritypolicy", "ActAsService");
final URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
ActAsServiceIface proxy = (ActAsServiceIface) service.getPort(ActAsServiceIface.class);

Bus bus = BusFactory.newInstance().createBus();
try {
   BusFactory.setThreadDefaultBus(bus);

   Map<String, Object> ctx = proxy.getRequestContext();

   ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
   ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
```

```
            Thread.currentThread().getContextClassLoader().getResource(
            "META-INF/clientKeystore.properties"));
        ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myactaskey");
        ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
            Thread.currentThread().getContextClassLoader().getResource(
            "META-INF/clientKeystore.properties"));
        ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");

        // Generate the ActAs element contents and pass to the STSClient as a string
        UsernameTokenCallbackHandler ch = new UsernameTokenCallbackHandler();
        String str = ch.getUsernameTokenString("alice","clarinet");
        ctx.put(SecurityConstants.STS_TOKEN_ACT_AS, str);

        STSClient stsClient = new STSClient(bus);
        Map<String, Object> props = stsClient.getProperties();
        props.put(SecurityConstants.USERNAME, "bob");
        props.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
        props.put(SecurityConstants.ENCRYPT_PROPERTIES,
            Thread.currentThread().getContextClassLoader().getResource(
            "META-INF/clientKeystore.properties"));
        props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
        props.put(SecurityConstants.STS_TOKEN_USERNAME, "myclientkey");
        props.put(SecurityConstants.STS_TOKEN_PROPERTIES,
            Thread.currentThread().getContextClassLoader().getResource(
            "META-INF/clientKeystore.properties"));
        props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

        ctx.put(SecurityConstants.STS_CLIENT, stsClient);
    } finally {
        bus.shutdown(true);
    }
    proxy.sayHello();
```

Revised on 2024-01-17 05:24:55 UTC