Red Hat JBoss Enterprise Application Platform 7.0 Developing Web Services Applications

For Use with Red Hat JBoss Enterprise Application Platform 7

Red Hat Customer Content Services

Red Hat JBoss Enterprise Application Platform 7.0 Developing Web Services Applications

For Use with Red Hat JBoss Enterprise Application Platform 7

Abstract

This guide provides information about how to develop web service applications with Red Hat JBoss Enterprise Application Platform 7.

Table of Contents

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 JAX-RS WEB SERVICES

JAX-RS is the Java API for RESTful web services. It provides support for building web services using REST, through the use of 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 JAX-RS and is fully compliant with the JSR-000339 Java API for RESTful Web Services 2.0 specification. It also provides additional features to the specification.

To get started with JAX-RS, see the **`helloworld-rs`**, **`jax-rs-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. JAX-RS CLIENT

2.1.1. JAX-RS 2.0 Client API

JAX-RS 2.0 introduces 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 sub-resource 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.

Standard Way

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

Or

```
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 ResteasyClientBuilder

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

Resteasy automatically loads a set of default providers that includes all classes listed in the **META-INF/services/javax.ws.rs.ext.Providers** files. 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 may be needed.

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

Using Resteasy Client Classes

You have to add dependency for users of the Maven project:

```
<dependency>
 <groupId>org.jboss.resteasy</groupId>
 <artifactId>resteasy-client</artifactId>
 <version>VERSION IN EAP</version>
 </dependency>
```

Client Side Filters

The client side has two types of filters:

- **ClientRequestFilter**
- **ClientResponseFilter**

ClientRequestFilters run before an HTTP request is sent over the wire to the server. **ClientResponseFilters** run after a response is received from the server, but before the response body is unmarshalled. **ClientRequestFilters** are also allowed to abort the request execution and provide a canned response without going over the wire to the server. **ClientResponseFilters** can modify the Response object before it is handed to the application code. For example:

```
// 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 Client Request

For example:

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

2.1.2. Implementing RESTEasy with HTTP Client

By default, network communication between the client and server in RESTEasy is handled by HttpClient (4.x) from the Apache HttpComponents project.

The interface between the RESTEasy Client Framework and network is found in an implementation of **`org.jboss.resteasy.client.jaxrs.ClientHttpEngine`**, and **`org.jboss.resteasy.client.jaxrs.engines.ApacheHttpClient4Engine`**, which uses HttpClient (4.x), is the default implementation. RESTEasy also ships with the following client engines, all found in the **`org.jboss.resteasy.client.jaxrs.engines`** package:

- **`URLConnectionClientExecutor`**: Uses **`java.net.HttpURLConnection`**.
- **`InMemoryClientExecutor`**: Dispatches requests to a server in the same JVM.

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

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

RESTEasy and HttpClient make default decisions to use the client framework without referencing HttpClient, but for some applications it might be necessary to drill down into the HttpClient details. **`ApacheHttpClient4Engine`** can be supplied with an instance of **`org.apache.http.client.HttpClient`** and an instance of **`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();
```

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 counterexample 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 released the connection.

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

You should probably execute this in a try/finally block. Again, releasing a connection only makes it available for another use. It does not normally close the socket.

`ApacheHttpClient4Engine.finalize()` closes any open sockets, if it created the HttpClient it has been using. It is not safe to rely on JDK to call **`finalize()`**. If an **`HttpClient`** has been passed into the **`ApacheHttpClient4Executor`**, the user is responsible for closing the connections:

```
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. URL-BASED NEGOTIATION

2.2.1. 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>
```

2.2.2. 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>
```

2.3. CONTENT MARSHALLING AND PROVIDERS

2.3.1. Default Providers and Default JAX-RS 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+*`**	JAXB 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

2.3.2. Content Marshalling with @Provider classes

The JAX-RS specification allows you to plug in your own request/response body reader 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.

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.

2.3.3. 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;
 ...
}
```

2.3.4. 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 document could cause the **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 context parameter to **true** in the **web.xml** file:

```
<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 context parameter to **false**. 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. 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 context parameter to **`false`**.

```
<!--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;'-->
```

2.3.5. Using MapProvider

You can use **`MapProvider`** to accept and return a map with JAX-RS 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 JAX-RS 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...
```

2.3.6. Converting String Based Annotations to Objects

JAX-RS **`@*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, the JAX-RS 2.0 specification 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 JAX-RS runtime or must be annotated with **`@Provider`** annotation to be automatically discovered by the JAX-RS 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) {
    ...
  }
}
```

2.3.7. JAXB Providers

2.3.7.1. JAXB and XML Provider

RESTEasy provides JAXB 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 convenience 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;
   }
}
```

2.3.7.2. JAXB and JSON Provider

RESTEasy allows you to marshal JAXB 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 JAXB.

While Jackson already includes JAX-RS 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;
    }
}
```

2.3.7.2.1. 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>
```

2.3.8. Creating JAXB Decorators

RESTEasy’s JAXB 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 JAXB Decorator with RESTEasy

1. Create the Processor Class.

   a. Create a class that implements **`DecoratorProcessor<Target, Annotation>`**. The target is either the JAXB 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 JAXB **`Marshaller`** or **`Unmarshaller`** class.

   Example 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 JAXB decorator, which can be applied within a JAX-RS web service.

2.3.9. Multipart Providers in JAX-RS

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.

2.3.9.1. Input with Multipart Data

When writing a JAX-RS 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.

2.3.9.1.1. 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 JAXB.

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 Unmarshaling 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.

2.3.9.1.2. 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 JAXB.

2.3.9.1.3. Input with `multipart/form-data`

When writing a JAX-RS 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`**.

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.

2.3.9.1.4. `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 JAXB.

2.3.9.1.5. Input with `multipart/related`

When writing a JAX-RS 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`**.

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`**.

2.3.9.2. 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 may have marshalling which 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 JAXB.

2.3.9.2.1. 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 JAXB.

2.3.9.2.2. 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 may have marshalling which 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 JAXB.

2.3.9.2.3. 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 JAXB.

2.3.9.2.4. 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 will be used as the root part of the **`multipart/related`** message, and RESTEasy will automatically find a **`MessageBodyWriter`** to marshal your entity objects. Similar to **`MultipartInput`**, sometimes you may have marshalling which 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 JAXB.

2.3.9.3. 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 JAX-RS **`@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 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 was used to tell 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 Recieve `CustomerProblemForm`

Example Recieve 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...
    }
}
```

2.3.9.4. XML-binary Optimized Packaging (XOP)

If you have a JAXB annotated POJO that also holds some binary content, you may choose to send it in such a way where the binary does not need to be encoded in any way (neither base64 neither 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 JAXB annotated POJO.

Example JAXB 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 JAXB 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 an **`java.io.InputStream`** or a **`javax.activation.DataSource`**, you need to go with 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 JAXB.

The **@Consumes(MultipartConstants.MULTIPART_RELATED)** is used to tell RESTEasy that you want to send **multipart/related** packages, which is the container format that will hold 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.

2.3.9.5. 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;
    }
}
```

2.3.9.6. 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`**, if any, 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);
        ...
    }
```

2.3.9.7. 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`**.

2.3.9.8. 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.

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 first need to 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();
```

2.3.10. 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 JAXB to marshal and unmarshal the API. The provider is JAXB based, and is not limited to sending atom objects using XML. All JAXB 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;
   }
}
```

2.3.10.1. Using JAXB with Atom Provider

The **`org.jboss.resteasy.plugins.providers.atom.Content`** class allows you to unmarshal and marshal JAXB 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 JAXB to marshal appropriately. If you are using a different base format other than XML, that is **`application/atom+json`**, the attached JAXB object is marshalled in the same format. If you have an atom document as input, you can also extract JAXB 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);
   }
}
```

2.3.11. YAML Provider

Warning

The **`resteasy-yaml-provider`** module is not supported.

RESTEasy has a provider for YAML using the **SnakeYAML** library. To enable this, you must update the following dependencies into the project POM file of your application:

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 recognizes three mime types:

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

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();
  }
...
}
```

2.4. USING THE JSON API FOR JSON PROCESSING (JSON-P)

The JSON API for JSON Processing (JSON-P) is a part of the Java EE 7 specification and is defined in JSR 353. JSON-P 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

The JSON API for JSON Processing (JSON-P) is different than JSON with Padding (JSONP).

Note

JSON-P 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();
```

Corresponding JSON

```
{
  "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();
```

Corresponding JSON

```
[
  {
  "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 JAX-RS resources:

Example JAX-RS Resources with JSON-P

```
@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 JSON-P from a client to send JSON:

Example Client using JSON-P

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

2.5. RESTEASY/EJB INTEGRATION

To integrate RESTEasy with EJB, start by updating the published interfaces of your EJB. Currently, RESTEasy only has simple portable integration with EJBs, so you must also manually configure your RESTEasy WAR file.

To make an EJB function as a JAX-RS resource, annotate an SLSB's **`@Remote`** or **`@Local`** interface with JAX-RS annotations:

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

Next, in the RESTEasy **`web.xml`** file, manually register the EJB with RESTEasy using the **`resteasy.jndi.resources <context-param>`**:

```
<web-app>
   <display-name>Archetype Created Web Application</display-name>
   <context-param>
      <param-name>resteasy.jndi.resources</param-name>
      <param-value>LibraryBean/local</param-value>
   </context-param>
   <listener>
      <listener-
class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listen
er-class>
   </listener>
   <servlet>
      <servlet-name>Resteasy</servlet-name>
      <servlet-
class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</se
rvlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>Resteasy</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

2.6. SPRING INTEGRATION

Note

Your application must have an existing JAX-WS 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 JAX-RS annotations when a bean is created by a **BeanFactory**. This means that RESTEasy automatically scans for **@Provider** and JAX-RS resource annotations on your bean class and registers them as JAX-RS 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</listen
er-class>
  </listener>
  <listener>
    <listener-
class>org.jboss.resteasy.plugins.spring.SpringContextLoaderListener</list
ener-class>
  </listener>
  <servlet>
    <servlet-name>Resteasy</servlet-name>
    <servlet-
class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</se
rvlet-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.

2.7. RESTEASY FILTERS AND INTERCEPTORS

JAX-RS 2.0 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.

2.7.1. Server-Side Filters

On the server-side you have two different types of filters: **ContainerRequestFilters** and **ContainerResponseFilters**. **ContainerRequestFilters** run before your JAX-RS resource method is invoked. **ContainerResponseFilters** run after your JAX-RS 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 will execute before the JAX-RS resource method is matched with the incoming HTTP request. Post-matching **`ContainerRequestFilters`** are designated with the **`@PostMatching`** annotation and will execute after the JAX-RS 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, JAX-RS will run 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);
      }
   }
}
```

2.7.2. Client-Side Filters

More information on client-side filters can be found in the JAX-RS 2.0 Client API section.

2.7.3. RESTEasy Interceptors

2.7.3.1. Intercept JAX-RS Invocations

RESTEasy can intercept JAX-RS invocations and route them through listener-like objects called interceptors. This topic covers descriptions of the four types of 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.

RESTEasy GZIP support has interceptors that create and override the default **Output** and **Input** streams with a **GzipOutputStream** or **GzipInputStream** so that gzip encoding can work. They can also be used to append headers to the response, or the outgoing request on the client side.

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()");
   }
}
```

2.7.3.2. Registering an Interceptor

This topic covers how to register a RESTEasy JAX-RS interceptor in an application.

To register an interceptor, list it in the **`web.xml`** file under the **`resteasy.providers`** **`context-param`**, 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 PubSubApplication() {
     singletons.add(new MyResource());
     singletons.add(new MyProvider());
   }

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

2.7.4. 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: register an implementation of **`DynamicFeature`** or use the **`@NameBinding`** annotation.

The **`DynamicFeature`** interface is executed at deployment time for each resource method. You use the **`Configurable`** interface to register the filters and interceptors for the specific resource method.

`@NameBinding` works a lot like CDI interceptors. You annotate a custom annotation with **`@NameBinding`** and then apply that custom annotation to your filter and resource method.

Example Per-Resource Filter

```
@NameBinding
public @interface DoIt {}

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

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

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

2.7.5. Ordering

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

2.7.6. 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 JAX-RS methods, which tries to find an **`ExceptionMapper`** for the exception being thrown. More details on how exceptions are handled in JAX-RS methods can be found in the Exception Handling section.

2.8. EXCEPTION HANDLING

2.8.1. 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:

```
@Provider
public class EJBExceptionMapper implements
ExceptionMapper<javax.ejb.EJBException>
    {
    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.

2.8.2. 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 JAX-RS method for the resource to handle the invoked HTTP operation.
NotAcceptableException	406	There is no JAX-RS method that can produce the media types listed in the Accept header.

Exception	HTTP Code	Description
NotFoundException	404	There is no JAX-RS 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.
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 JAXB providers (XML and Jackson) throw this exception on reads which may wrap JAXBExceptions. This class extends ReaderException.
JAXBMarshalException	500	The JAXB providers (XML and Jackson) throw this exception on writes which may 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.

Exception	HTTP Code	Description
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 JAX-RS method for it, RESTEasy provides a default behavior by throwing this exception.

2.9. SECURING JAX-RS WEB SERVICES

2.9.1. Securing JAX-RS Web Services using RBAC

RESTEasy supports the **`@RolesAllowed`**, **`@PermitAll`**, and **`@DenyAll`** annotations on JAX-RS methods. However, it does not recognize these annotations by default. Follow these steps to configure the **`web.xml`** file and enable role-based security.

Warning

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

Enable Role-Based Security for a RESTEasy JAX-RS 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 JAX-RS WAR file, using the **`<security-role>`** tags:

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

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

4. Authorize access to all URLs handled by the JAX-RS runtime for all roles:

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

Result

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>

  <servlet-mapping>
    <servlet-name>Resteasy</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>

  <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>

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

</web-app>
```

2.9.2. Securing JAX-RS Web Services using Annotations

To secure JAX-RS web services using annotations, complete the following steps:

1. Enable role-based security.
2. Add security annotations to the JAX-RS 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.

2.10. 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 does have a faint idea 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.

2.10.1. 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>
```

2.10.2. Configuring Asynchronous Jobs

This topic 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 may 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 will return a **`202 Accepted`** response. It will also return 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 JAX-RS resource method invoked as a response if the job was completed. If the job has not been completed, this GET will return 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 will return 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 section in the appendix.

2.11. RESTEASY JAVASCRIPT API

2.11.1. About the RESTEasy JavaScript API

RESTEasy can generate a JavaScript API that uses AJAX calls to invoke JAX-RS operations. Each JAX-RS resource class will generate a JavaScript object of the same name as the declaring class or interface. The JavaScript object contains each JAX-RS 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){
 }
}
```

We can use the previous JAX-RS API in JavaScript using the following code:

```
var text = Foo.get({order: 'desc', 'X-Foo': 'hello',
                   colour: 'blue', 'Foo-Cookie': 123987235444});
Foo.put({$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 their name, or the API parameter properties. The properties are available in the RESTEasy Javascript API Parameters appendix.

Enable the RESTEasy JavaScript API Servlet

The RESTEasy JavaScript API is not enabled by default. Follow these steps to enable it using 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>
```

2.11.2. Build AJAX Queries

The RESTEasy JavaScript API can be used to manually construct requests. This topic covers examples of this behavior.

The **REST** object can be 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 JAX-RS root URL. Used by every JavaScript client API functions when constructing the requests.

log

Set to a **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.

The **REST.Request** class can be used 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);
});
```

CHAPTER 3. DEVELOPING JAX-WS WEB SERVICES

3.1. ABOUT JAX-WS WEB SERVICES

The Java API for XML-Based Web Services (JAX-WS) defines the mapping between WSDL and Java as well as the classes to be used for accessing webservices and publishing them. JBossWS implements the latest JAX-WS specification, hence users can reference it for any vendor agnostic webservice usage need. Below is a brief overview of the most basic functionality.

3.2. CONFIGURING THE WEB SERVICES SUBSYSTEM

3.2.1. Domain Management

JBossWS components are provided to the application server through the web services subsystem. JBossWS components handle the processing of WS endpoints. The subsystem supports the configuration of published endpoint addresses, and endpoint handler chains. A default webservice subsystem is provided in the server’s domain and standalone configuration files.

3.2.1.1. Structure of the Web Services Subsystem

3.2.1.1.1. Published Endpoint Address

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 elements are available and can be modified (all are optional and a require server restart upon modification):

Name	Type	Description

Name	Type	Description
modify-wsdl-address	boolean	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	string	The hostname 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-port	int	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	int	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	Type	Description
wsdl-uri-scheme	string	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.
wsdl-path-rewrite-rule	string	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.

You can use the management CLI to update these attributes.

Example Management CLI Command

```
/profile=full-ha/subsystem=webservices:write-attribute(name=wsdl-uri-scheme, value=https)
```

3.2.1.1.2. Predefined Endpoint Configurations

JBossWS enables extra setup configuration data to be predefined and associated with an endpoint implementation. Predefined endpoint configurations can be used for JAX-WS client and JAX-WS endpoint setup. Endpoint configurations can include JAX-WS handlers and key/value properties declarations. This feature provides a convenient way to add handlers to WS endpoints and to set key/value properties that control JBossWS and Apache CXF internals.

The web services subsystem provides schema to support the definition of named sets of endpoint configuration data. The annotation **`org.jboss.ws.api.annotation.EndpointConfig`** is provided to map the named configuration to the endpoint implementation.

There is no limit to the number of endpoint configurations that can be defined within the web services subsystem. Each endpoint configuration must have a name that is unique within the web services subsystem. Endpoint configurations defined in the web services subsystem are available for reference by name through the annotation to any endpoint in a deployed application.

There are two predefined endpoint configurations. **`Standard-Endpoint-Config`** is the default configuration. **`Recording-Endpoint-Config`** is an example of custom endpoint configuration and includes a recording handler.

```
[standalone@localhost:9999 /] /subsystem=webservices:read-resource
```

```
{
    "outcome" => "success",
    "result" => {
        "endpoint" => {},
        "modify-wsdl-address" => true,
        "wsdl-host" => expression "${jboss.bind.address:127.0.0.1}",
        "endpoint-config" => {
            "Standard-Endpoint-Config" => undefined,
            "Recording-Endpoint-Config" => undefined
        }
    }
}
```

The **`Standard-Endpoint-Config`** is a special endpoint configuration. It is used for any endpoint that does not have an explicitly assigned endpoint configuration.

3.2.1.1.2.1. Endpoint Configurations

Endpoint configs are defined using the **`endpoint-config`** element. Each endpoint configuration can include properties and handlers set to the endpoints associated with the configuration.

```
[standalone@localhost:9999 /] /subsystem=webservices/endpoint-
config=Recording-Endpoint-Config:read-resource
{
    "outcome" => "success",
    "result" => {
        "post-handler-chain" => undefined,
        "property" => undefined,
        "pre-handler-chain" => {"recording-handlers" => undefined}
    }
}
```

A new endpoint configuration can be added as follows:

```
[standalone@localhost:9999 /] /subsystem=webservices/endpoint-
config=My-Endpoint-Config:add
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-restart" => true,
        "process-state" => "restart-required"
    }
}
```

3.2.1.1.2.2. Handler chains

Each endpoint configuration may be associated with zero or more **`PRE`** and **`POST`** handler chains. Each handler chain may include JAX-WS handlers. For outbound messages the **`PRE`** handler chains are executed before any handler that is attached to the endpoint using the standard means, such as the annotation **`@HandlerChain`**, and **`POST`** handler chains are executed after those objects have executed. For inbound messages the **`POST`** handler chains are executed before any handler that is attached to the endpoint using the standard means and the **`PRE`** handler chains are executed after those objects have executed.

```
* Server inbound messages
Client --> ... --> POST HANDLER --> ENDPOINT HANDLERS --> PRE HANDLERS
--> Endpoint

* Server outbound messages
Endpoint --> PRE HANDLER --> ENDPOINT HANDLERS --> POST HANDLERS -->
... --> Client
```

The protocol-binding attribute must be used to set the protocols for which the chain are triggered.

```
[standalone@localhost:9999 /] /subsystem=webservices/endpoint-
config=Recording-Endpoint-Config/pre-handler-chain=recording-
handlers:read-resource
{
    "outcome" => "success",
    "result" => {
        "protocol-bindings" => "##SOAP11_HTTP ##SOAP11_HTTP_MTOM
##SOAP12_HTTP ##SOAP12_HTTP_MTOM",
        "handler" => {"RecordingHandler" => undefined}
    },
    "response-headers" => {"process-state" => "restart-required"}
}
```

A new handler chain can be added as follows:

```
[standalone@localhost:9999 /] /subsystem=webservices/endpoint-
config=My-Endpoint-Config/post-handler-chain=my-handlers:add(protocol-
bindings="##SOAP11_HTTP")
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-restart" => true,
        "process-state" => "restart-required"
    }
}
[standalone@localhost:9999 /] /subsystem=webservices/endpoint-
config=My-Endpoint-Config/post-handler-chain=my-handlers:read-resource
{
    "outcome" => "success",
    "result" => {
        "handler" => undefined,
        "protocol-bindings" => "##SOAP11_HTTP"
    },
    "response-headers" => {"process-state" => "restart-required"}
}
```

3.2.1.1.2.3. Handlers

JAX-WS handler can be added in handler chains:

```
[standalone@localhost:9999 /] /subsystem=webservices/endpoint-
config=Recording-Endpoint-Config/pre-handler-chain=recording-
handlers/handler=RecordingHandler:read-resource
{
    "outcome" => "success",
    "result" => {"class" =>
"org.jboss.ws.common.invocation.RecordingServerHandler"},
    "response-headers" => {"process-state" => "restart-required"}
}
[standalone@localhost:9999 /] /subsystem=webservices/endpoint-
config=My-Endpoint-Config/post-handler-chain=my-handlers/handler=foo-
handler:add(class="org.jboss.ws.common.invocation.RecordingServerHandle
r")
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-restart" => true,
        "process-state" => "restart-required"
    }
}
```

Endpoint-config Handler Classloading

The class attribute is used to provide the fully qualified class name of the handler. At deploy time, an instance of the class is created for each referencing deployment. For class creation to succeed, either the deployment classloader or the classloader for module, **`org.jboss.as.webservices.server.integration`**, must to be able to load the handler class.

3.2.2. 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:

```
[standalone@localhost:9999 /]
/deployment="*"/subsystem=webservices/endpoint="*":read-resource
{
   "outcome" => "success",
   "result" => [{
       "address" => [
           ("deployment" => "jaxws-samples-handlerchain.war"),
           ("subsystem" => "webservices"),
           ("endpoint" => "jaxws-samples-handlerchain:TestService")
       ],
       "outcome" => "success",
       "result" => {
           "class" =>
"org.jboss.test.ws.jaxws.samples.handlerchain.EndpointImpl",
           "context" => "jaxws-samples-handlerchain",
           "name" => "TestService",
           "type" => "JAXWS_JSE",
           "wsdl-url" => "http://localhost:8080/jaxws-samples-
handlerchain?wsdl"
       }
   }]
}
```

3.2.3. Configuring Handlers and Handler Chains

Each endpoint configuration may be associated with **PRE** and **POST** handler chains. Each handler chain may include JAX-WS-compliant handlers. For outbound messages, **PRE** handler chain handlers are executed before any handler attached to the endpoints using standard JAX-WS means, such as the **@HandlerChain** annotation. **POST** handler chain handlers are executed after usual endpoint handlers. For inbound messages, the opposite applies. JAX-WS is a standard API for XML-based web services, and is documented at http://jcp.org/en/jsr/detail?id=224.

A handler chain may also include a **protocol-bindings** attribute, which sets the protocols which trigger the chain to start.

A JAX-WS handler is a child element **handler** within a handler chain. The handler takes a **class** attribute, which is the fully-qualified classname 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 module **org.jboss.as.webservices.server.integration** must be able to load the handler class.

1. Start the JBoss EAP management CLI.
2. Add handler chain and handlers using the management CLI:

   Example: Add a handler chain

   ```
   [standalone@localhost:9999 /] /subsystem=webservices/endpoint-
   config=Standard-Endpoint-Config/post-handler-chain=:add(protocol-
   bindings="##SOAP11_HTTP")
   ```

   Example: Add a handler

   ```
   [standalone@localhost:9999 /] /subsystem=webservices/endpoint-
   config=Standard-Endpoint-Config/post-handler-
   chain=/handler=:add(class="org.jboss.ws.common.invocation.Recordi
   ngServerHandler")
   ```

   Example: Add a handler

   ```
   [standalone@localhost:9999 /] /subsystem=webservices/endpoint-
   config=Standard-Endpoint-Config/post-handler-
   chain=/handler=:add(class="com.arjuna.webservices11.wsarj.handler
   .InstanceIdentifierInHandler")
   ```

3. Reload the server:

   ```
   [standalone@localhost:9999 /] reload
   ```

4. Confirm the handler-chain and handlers were added correctly:

   Example: Read a handler-chain

   ```
   [standalone@localhost:9999 /]
   /profile=default/subsystem=webservices/endpoint-config=Standard-
   Endpoint-Config/post-handler-chain=:read-resource
   ```

   Example: Read a handler

```
[standalone@localhost:9999 /]
/profile=default/subsystem=webservices/endpoint-config=Standard-
Endpoint-Config/post-handler-chain=/handler=:read-resource
```

The options used in the commands above can be modified as required to add or modify handlers.

3.2.4. Configuring HTTP Timeout per Application

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 several places. In order of precedence these are:

- Application - defined in the application’s **`web.xml`** configuration file.
- Server - specified via the **`default-session-timeout`** attribute.
- Default - 30 minutes.

To configure the HTTP session timeout in an application’s **`web.xml`**, add the following configuration XML to the file, changing **`30`** to the desired timeout (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.

3.3. USING JAX-WS TOOLS

The following JAX-WS 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. JAX-WS Command-Line Tools

Command	Description
wsprovide	Generates JAX-WS 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 JAX-WS Tools for more details on the usage of these tools.

3.3.1. 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 EJB3 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 JAX-WS 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 JAX-WS 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 JAX-WS 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
EchoResponse.java	Wrapper bean for response message
EchoService.java	Used only by JAX-WS clients
Echo_Type.java	Wrapper bean for request message
ObjectFactory.java	JAXB XML Registry

File	Purpose
package-info.java	Holder for JAXB 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;
   }
}
```

3.3.2. 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 JAX-WS, **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 WS 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));
```

3.4. JAX-WS WEB SERVICE ENDPOINTS

3.4.1. About JAX-WS Web Service Endpoints

A JAX-WS 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 two ways:

1. You can write WSDL descriptors manually.

2. You can use JAX-WS annotations that create the WSDL descriptors automatically for you. This is the most common method for creating WSDL descriptors.

An endpoint implementation bean is annotated with JAX-WS 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 Java Architecture for XML Binding (JAXB) service.

The endpoint itself may be a POJO (Plain Old Java Object) or a Java EE Web Application. You can also expose endpoints using an EJB3 stateless session bean. It is packaged into a Web Archive (WAR) file. The specification for packaging the endpoint, called a Java Service Endpoint (JSE) is defined in JSR-181, which can be found at http://jcp.org/aboutJava/communityprocess/mrel/jsr181/index2.html.

Development Requirements

A Web Service must fulfill the requirements of the JAX-WS API and the JSR 181: Web Services Metadata for the Java Platform specification. A valid implementation meets the following requirements:

- It contains a **`javax.jws.WebService`** annotation.
- All method parameters and return types are compatible with the JSR 222: JavaTM Architecture for XML Binding (JAXB) 2.0 specification.

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</serv
let-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>TestService</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

This EJB3 stateless session bean exposes the same method on the remote interface and 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)
    {
        ...
    }
}
```

Endpoint Providers

JAX-WS services typically implement a Java service endpoint interface (SEI), which may be mapped from a WSDL port type, either directly or using annotations. This SEI provides a high-level abstraction which hides the details between Java objects and their XML representations. However, in some cases, services need the ability to operate at the XML message level. The endpoint **`Provider`** interface provides this functionality to Web Services which 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.

3.4.2. Developing and Deploying JAX-WS Web Service Endpoint

A JAX-WS service endpoint is a server-side component that responds to requests from JAX-WS clients and publishes the WSDL definition for itself.

Note

See the following quickstarts that ship with JBoss EAP for working examples of how to develop JAX-WS 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 JAX-WS API and the JSR 181: Web Services Metadata for the Java Platform specification. A valid implementation meets the following requirements:

- It contains a **`javax.jws.WebService`** annotation.
- All method parameters and return types are compatible with the JSR 222: JavaTM Architecture for XML Binding (JAXB) 2.0 specification.

The following is an example of a Web Service implementation that meets these requirements.

Web Service Code Example

```
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 JAXB defaults are reasonable and do not need to be specified.

DiscountRequest Class Code Example

```
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;
   }

}
```

Package 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. 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 correct structure for a JAR deployment of an EJB Web Service.

```
[user@host ~]$ 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
```

3.5. JAX-WS WEB SERVICE CLIENTS

3.5.1. Consume and Access a JAX-WS Web Service

After creating a Web Service endpoint, either manually or using JAX-WS annotations, you can access its WSDL, which can be used to create the basic client application which 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 topic.

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/jboss-jaxws-
retail/ProfileMgmtBean'/>
      </port>
   </service>
</definitions>
```

Note

If you use JAX-WS annotations to create your Web Service endpoint, the WSDL contract is generated automatically, and you only need its URL. You can get this URL from the Web services section of the Runtime section of the management console, after the endpoint is deployed.

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.

```
[user@host bin]$ ./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 JAXB 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 specification 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]$
output/org/jboss/test/ws/jaxws/samples/retail/profile/Customer.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.j
ava
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.
java
output/org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.jav
a
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtServic
e.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.c
lass
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.
class
output/org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.cla
ss
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtServic
e.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.

File	Description
`ObjectFactory.java`	JAXB XML registry.
`package-info.java`	JAXB package annotations.
`ProfileMgmtService.java`	Service factory.

The **`wsconsume`** command generates all custom data types (JAXB 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 WS 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. Select the **Runtime** tab, select your server, then under **Subsystems**, select **Webservices** and click **View**. View the **Attributes** tab to review your deployments details.

```
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
```

3.5.2. Develop a JAX-WS Client Application

The client communicates with, and requests work from, the JAX-WS 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 JAX-WS client assumes that you already have a WSDL contract. This may be generated by an external tool or generated by using the correct JAX-WS annotations when you create your JAX-WS 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);
   }
   ...
}
```

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

JAX-WS provides a flexible plug-in framework for message processing modules, known as handlers. These handlers extend the capabilities of a JAX-WS 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.

- 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 may (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 may (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 Java EE container, and any clients. The XML messages use an XML language called Simple Object Access Protocol (SOAP). The JAX-WS 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 may 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.conn
ectionTimeout", "6000");

   //Set timeout until the response is received
   ((BindingProvider)
port).getRequestContext().put("javax.xml.ws.client.receiveTimeout",
"1000");

   port.echo("testTimeout");
}
```

3.6. PREDEFINED CLIENT AND ENDPOINT CONFIGURATIONS

Extra setup configuration data can be predefined and associated with an endpoint or a client. A configuration can include JAX-WS handlers and key/value property declarations that control JBossWS and Apache CXF internals. Predefined configurations can be used for JAX-WS client and JAX-WS endpoint setup.

A configuration can be defined in the webservices subsystem and in an application’s deployment descriptor file. There can be many configuration definitions in the webservices subsystem and in an application. Each configuration must have a name that is unique within the server. A configuration defined in an application is local to the application. Endpoint implementations declare the use of a specific configuration through the use of the **`org.jboss.ws.api.annotation.EndpointConfig`** annotation. An endpoint configuration defined in the webservices subsystem is available to all deployed applications on the JBoss EAP instance and can be referenced by name in the annotation. An endpoint configuration defined in an application must be referenced by both deployment descriptor file name and configuration name by the annotation.

Handlers

Each endpoint configuration may be associated with zero or more **PRE** and **POST** handler chains. Each handler chain may include JAX-WS handlers. For outbound messages the **PRE** handler chains are executed before any handler that is attached to the endpoint using the standard means, such as with annotation **@HandlerChain**, and **POST** handler chains are executed after those objects have executed. For inbound messages the **POST** handler chains are executed before any handler that is attached to the endpoint using the standard means and the **PRE** handler chains are executed after those objects have executed.

```
* Server inbound messages
Client --> ... --> POST HANDLER --> ENDPOINT HANDLERS --> PRE HANDLERS
--> Endpoint

* Server outbound messages
Endpoint --> PRE HANDLER --> ENDPOINT HANDLERS --> POST HANDLERS -->
... --> Client
```

This also applies for a client configuration.

Properties

Key/value properties are used for controlling both some Apache CXF internals and some JBossWS options. Specific supported values are mentioned where relevant in the rest of the documentation.

3.6.1. Assigning a Configuration

Endpoints and clients are assigned configuration through different means. Users can explicitly require a given configuration or rely on container defaults. The assignment process can be split up as follows:

- 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

3.6.1.1. 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.

3.6.1.1.1. Configuration Deployment Descriptor

Java EE archives that can contain JAX-WS 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 schema jbossws-jaxws-config. 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 Endpoints

```
<?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:handl
er-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>
```

3.6.1.1.2. 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.KeystorePasswordC
allback</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>
```

3.6.1.1.3. 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 JAX-WS 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!";
   }
}
```

3.6.1.1.4. JAX-WS Feature

You can also use **`org.jboss.ws.api.configuration.ClientConfigFeature`** to set a configuration which is a JAX-WS 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");

... or ....

port = service.getPort(Endpoint.class, new ClientConfigFeature("META-
INF/my-client-config.xml", "Custom Client Config"), true); //setup
properties too from the configuration
port.echo("Kermit");
```

```
... or ...

port = service.getPort(Endpoint.class, new ClientConfigFeature(null,
testConfigName)); //reads from current container configurations if
available
port.echo("Kermit");
```

JBossWS parses the specified configuration file. The configuration file must be found as a resource by the classloader of the current thread. The jbossws-jaxws-config schema defines the descriptor contents and is included in the **`jbossws-spi`** artifact.

3.6.1.1.5. Explicit Setup Through API

Alternatively, the JBossWS API comes with facility classes that can be used for assigning configurations when building a client. JAX-WS handlers 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;
ClientConfigUtil.setConfigHandlers(bp, "META-INF/my-client-config.xml",
"Custom Client Config 1");
port.echo("Kermit");

...OR...

ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigHandlers(bp, "META-INF/my-client-config.xml", "Custom
Client Config 2");
port.echo("Kermit");

...OR...

configurer.setConfigHandlers(bp, "META-INF/my-client-config.xml", "Custom
Client Config 3");
port.echo("Kermit");

...OR...

configurer.setConfigHandlers(bp, null, "Container Custom Client Config");
//reads from current container configurations if available
port.echo("Kermit");
```

Similarly, properties are read from client configurations as follows:

```
import org.jboss.ws.api.configuration.ClientConfigUtil;
import org.jboss.ws.api.configuration.ClientConfigurer;

...OR...
```

```
Service service = Service.create(wsdlURL, serviceName);
Endpoint port = service.getPort(Endpoint.class);

ClientConfigUtil.setConfigProperties(port, "META-INF/my-client-
config.xml", "Custom Client Config 1");
port.echo("Kermit");

...OR...

ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigProperties(port, "META-INF/my-client-config.xml",
"Custom Client Config 2");
port.echo("Kermit");

...OR...

configurer.setConfigProperties(port, "META-INF/my-client-config.xml",
"Custom Client Config 3");
port.echo("Kermit");

...OR...

configurer.setConfigProperties(port, null, "Container Custom Client
Config"); //reads from current container configurations if available
port.echo("Kermit");
```

The default **`ClientConfigurer`** implementation parses the specified configuration file, if any, after having resolved it as a resources using the current thread context classloader. The jbossws-jaxws-config schema defines the descriptor contents and is included in the **`jbossws-spi`** artifact.

3.6.1.2. 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, which is a compile-time dependency, of JBossWS API might not be accepted. 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). These are parsed for getting configurations any time a configuration file name is not specified.

If the configuration name is also not specified, JBossWS automatically looks for a configuration named the same as:

- the endpoint implementation class (fully qualified name), in case of JAX-WS endpoints.
- the service endpoint interface (fully qualified name), in case of JAX-WS clients.

No automatic configuration name is selected for **`Dispatch`** clients.

For example, an endpoint implementation class **`org.foo.bar.EndpointImpl`**, for which no pre-defined 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.

3.6.1.3. 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 JAX-WS 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 endpoint implementation class (fully qualified name), in case of JAX-WS endpoints.
- the service endpoint interface (fully qualified name), in case of JAX-WS 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.

3.7. SECURING JAX-WS 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 heavy 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.

3.7.1. Applying Web Services Security (WS-Security)

Web Services supports 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.

3.7.1.1. 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.

3.7.1.2. 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.
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.

Configuration property	Description
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.

3.7.2. 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

3.7.2.1. Scenario: Basic WS-Trust

In this section we have provided 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 a SAML 2.0 token issued from a designed STS to be presented by the **`ws-requester`** using asymmetric binding. These communication requirements are declared in the **`ws-provider's`** WSDL. STS requires **`ws-requester`** credentials to be provided in a WSS UsernameToken format request using symmetric binding. The STS's response is provided containing a SAML 2.0 token. These communication requirements are declared in the STS's WSDL.

1. A **`ws-requester`** contacts the **`ws-provider`** and consumes its WSDL. On finding the security token issuer requirement, **`ws-requester`** creates and configures a **`STSClient`** with the information it requires 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 **`STSlient`** presents a message with the security token to the **`ws-provider`**.
6. The **`ws-provider`** verifies the token was issued by the STS, thus proving 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`**.

3.7.2.2. 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 JAX-WS 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.

3.7.3. 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.

3.7.3.1. 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 the How to Set Up SSO with SAML v2 guide.

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.

3.7.3.1.1. 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 file, database, LDAP, etc.) 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, please 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:

sts-users.properties

```
Eric=samplePass
Alan=samplePass
```

sts-roles.properties

```
Eric=All
Alan=
```

Important

You will also need to create a keystore for signing and encrypting the security tokens. This keystore will be used when configuring the **`picketlink.xml`** file.

3.7.3.1.2. 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>
```

3.7.3.1.3. 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>
```

3.7.3.1.4. Declare the Necessary Dependencies for the STS

The web application serving as the STS requires a dependency to be defined in **`jboss-deployment-structure.xml`**, 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.

Using `jboss-deployment-structure.xml` to Declare Dependencies

```
<jboss-deployment-structure>
  <deployment>
    <dependencies>
      <module name="org.picketlink"/>
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

3.7.3.1.5. 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`

```
<?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 will need a class for your web-service to use your WSDL:

Example 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 PicketLinkSTSService extends PicketLinkSTS {
    private static Logger log =
Logger.getLogger(PicketLinkSTSService.class.getName());

    @Resource
    public void setWSC(WebServiceContext wctx) {
        log.debug("Setting WebServiceContext = " + wctx);
        this.context = wctx;
    }
}
```

3.7.3.1.6. 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 via configuration, and the default values can be specified for some properties via configuration. Similar to the IDP and SP configuration in the *How to Set Up SSO with SAML v2* guide, 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 allows the STS administrator to set a the following default values:
  - **`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 sub elements 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.

Example picketlink.xml Configuration

```
<!DOCTYPE PicketLinkSTS>
<PicketLinkSTS xmlns="urn:picketlink:federation:config:2.1"
               STSName="PicketLinkSTS" TokenTimeout="7200"
EncryptToken="false">
    <KeyProvider

ClassName="org.picketlink.identity.federation.core.impl.KeyStoreKeyManage
r">
        <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.sa
ml.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.sa
ml.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.

Note

Another example of an STS can be found in the picketlink-sts quickstart.

3.7.3.2. Using a WS-Trust Security Token Service (STS) with a Client

To configure a client to obtain a security token from the STS, you will 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 will 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 will need to use the **WSTrustClient** to ask for a token, for example a SAML assertion, to be issued:

Example of 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.

3.7.3.3. 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, allowing STS Client creation to be removed as a possible bottleneck. Client pooling can be used from login modules which 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 **`STSClientPoolFactory`** class **`org.picketlink.identity.federation.bindings.stspool.STSClientPoolFactory`** provides client pool functionality to applications.

Using STSClientPoolFactory

STS clients are inserted into subpools using their 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 default number.

Example of 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 of Returning an STS Client to the Subpool

```
pool.returnClient();
```

To check if a subpool already exists for a given configuration:

Example of Checking if a Subpool Exists with a Given Configuration

```
if (! pool.configExists(stsClientConfig) {
    pool.createPool(stsClientConfig);
}
```

If the Federation subsystem is enabled, all client pools created for a deployment are destroyed automatically during the undeploy process. To manually destroy a pool:

Example of Manually Destroying a Subpool

```
pool.destroyPool(stsClientConfig);
```

3.8. 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 JAX-WS 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 JAX-WS endpoint code.

   JAX-WS Endpoint Code Example 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 JAX-WS client code to configure WS-Addressing.

JAX-WS Client Code Example 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/jboss-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.ServiceI
face.class,
                new AddressingFeature());
        // invoke method
        System.out.println(proxy.sayHello());
    }
}
```

The client and endpoint now communicate using WS-Addressing.

3.9. ENABLE WEB SERVICES RELIABLE MESSAGING

Web Services Reliable Messaging (WS-Reliable Messaging) is implemented internally in Apache CXF. A set of interceptors interact 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 and assertions.
- 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.

If you specify the Apache CXF reliable messaging feature in an optional CXF Spring XML descriptor, you have to use the Apache CXF WS-Policy engine. The advantage of this option is that it is portable.

The other approaches are Apache CXF proprietary options, and give you more control over the protocol configuration options that are not specified by the WS-Reliable Messaging Policy.

3.10. 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.

3.11. ADVANCED USER GUIDE

3.11.1. Logging: JAX-WS Handler and Apache CXF Approaches

There are options for logging inbound and outbound messages.

3.11.1.1. JAX-WS Handler Approach

You can write a JAX-WS handler to dump 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`** JAX-WS annotation.

The predefinted 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 JAX-WS 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.

3.11.1.2. Apache CXF Approach

Apache CXF also comes with logging interceptors that can be used to log messages to the console or configured client/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.
- Manual interceptor addition and logging feature: 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 client side by programmatically adding new instances of the logging interceptors to the client or the bus.

3.11.2. Address Rewrite

You can configure the **`soap:address`** attribute in the WSDL contract of deployed services.

3.11.3. Schema Validation of SOAP Messages

Apache CXF includes a feature for validating incoming and outgoing SOAP messages on both the client and 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).

On the client side, schema validation can be turned on programmatically. For example:

```
((BindingProvider)proxy).getRequestContext().put("schema-validation-
enabled", true);
```

On the server side, 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 {
   ...
}
```

3.12. JBOSS MODULES AND WS APPLICATIONS

3.12.1. 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.

While you can provide modules for custom needs, the following information provides suggestions about using the modules and web services development with JBoss EAP.

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 JBoss Web Services APIs are available by default whenever the webservices subsystem is available. You can use them without creating an explicit dependencies declaration for those modules.

3.12.1.1. 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
```

In this example, **`org.jboss.ws.cxf.jbossws-cxf-client`** and **`foo.bar`** are the modules that you want to set dependencies to. The **`services`** option tells the modules framework that you want to also import **`META-INF/services/..`** declarations from the dependency, while the **`export`** option exports the classes from the module to any other module that might depend on the module implicitly created for the deployment.

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 added to the annotation index. The annotations are completely, silently ignored.

3.12.1.2. Using JAXB

To successfully and directly use JAXB contexts in your client or endpoint running in-container, set up a JAXB implementation. For example, set the following dependency:

```
Dependencies: com.sun.xml.bind services export
```

3.12.1.3. 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 does not come with any JBossWS-CXF customizations nor additional extensions. For this reason, a client-side aggregation module is available with all the WS dependencies that you might need.

3.12.1.4. 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 for you to get the JBossWS-CXF version of classes that are retrieved using the Service API. 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.

3.12.1.5. Annotation Scanning

The application server uses an annotation index for detecting JAX-WS endpoints in user deployments. When declaring WS 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.

3.12.1.6. Using jboss-deployment-descriptor.xml

In some circumstances, the convenient approach of setting module dependencies in the **`MANIFEST.MF`** file might not work. For example, setting depdencies 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.

3.13. APACHE CXF INTEGRATION

All JAX-WS functionality provided by JBossWS on top of JBoss EAP are currently served through a proper integration of the JBoss Web Services 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 JAX-WS), with services speaking a variety of protocols such as SOAP and XML/HTTP over a variety of transports such as HTTP and JMS.

The integration layer, or JBossWS-CXF, is mainly meant for:

- allowing using standard webservices APIs (including JAX-WS) on JBoss EAP; this is performed internally leveraging Apache CXF without requiring the user to deal with it;
- allowing using Apache CXF advanced features (including WS-*) on top of JBoss EAP without requiring the user to deal with, setup or care about the required integration steps for running in such a container.

In support of those goals, the JBossWS-CXF integration supports the JBoss WS 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-CXF server side integration takes care of internally creating proper Apache CXF structures for the provided WS 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-CXF integration 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`** requests 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 **`cxf.queue.default.maxQueueSize`** for controlling the maximum queue size of the default **`WorkQueue`**. At deployment time, the JBossWS integration can add new instances of **`AutomaticWorkQueueImpl`** to the currently configured **`WorkQueueManager`**. The properties below are used to fill in parameters into the **`AutomaticWorkQueueImpl`** constructor:

Table 3.5. `AutomaticWorkQueueImpl` Constructor Properties

Property	Default Value
cxf.queue.<queue-name>.maxQueueSize	256

Property	Default Value
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-CXF integration currently defaults to the `MaximalAlternativeSelector`, but still allows for setting different selector implementation using the **`cxf.policy.alternativeSelector`** property in **`jboss-webservices.xml`**.

3.13.1.4. MBean Management

Apache CXF allows managing its MBean objects that are installed into the JBoss EAP MBean server. The feature is enabled on a deployment basis through the **`cxf.management.enabled`** property in **`jboss-webservices.xml`**. The **`cxf.management.installResponseTimeInterceptors`** property can also be used to control installation of CXF response time interceptors, which are added by default when enabling MBean management, but might not be desired in some cases.

Example MBean Management in jboss-webservices.xml

```
<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

You can enable schema validation of exchanged messages in **`jboss-webservices.xml`**. More details can be found in the Schema Validation of SOAP Messages section.

3.13.1.6. Apache CXF Interceptors

The **jboss-webservices.xml** descriptor enables specifying the **cxf.interceptors.in** and **cxf.interceptors.out** properties. Those properties allow for declaring interceptors to be attached to the **Bus** instance that is created for serving the deployment.

Example jboss-webservices.xml

```
<?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</va
lue>
  </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

Since Spring integration is no longer supported in JBoss EAP, the JBossWS integration uses a descriptor file, **jaxws-endpoint-config.xml**, 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

```
<?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.jb
oss.test.ws.jaxws.cxf.interceptors.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 the configuration is assigned to. The interceptor classes must have a no-argument constructor.

3.13.1.7. Apache CXF Features

The **jboss-webservices.xml** descriptor allows specifying the **cxf.features** property. This property allows declaring features to be attached to any endpoint belonging to the **Bus** instance which is created for serving the deployment.

Example jboss-webservices.xml

```
<?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

```
<?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 simply 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 for each attribute.

So for instance the following properties:

Key	Value
cxf.features	##foo, ##bar

Key	Value
##foo	org.jboss.Foo
##foo.par	34
##bar	org.jboss.Bar
##bar.color	blue

would result in the stack installing two feature instances. The same result would have been created by:

```
import org.Bar;
import org.Foo;
...
Foo foo = new Foo();
foo.setPar(34);
Bar bar = new Bar();
bar.setColor("blue");
```

The 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.

3.14. ADVANCED WS-TRUST SCENARIOS

3.14.1. 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`**.

3.14.1.1. 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

3.14.1.1.1. 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 designed 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>
```

3.14.1.1.2. SSL Configuration

This web service uses HTTPS, therefore the JBoss EAP server must be configured to provide SSL/TLS support in the Undertow subsystem. There are 2 components for SSL/TLS configuration:

- Create a certificate keystore.
- Declare an SSL connector in the Undertow subsystem of the JBoss EAP server configuration file.

The following is an example of an SSL/TLS connector declaration:

```
<subsystem xmlns="urn:jboss:domain:web:1.4" default-virtual-
server="default-host" native="false">
.....
  <connector name="jbws-https-connector" protocol="HTTP/1.1"
scheme="https" socket-binding="https" secure="true" enabled="true">
    <ssl key-alias="tomcat" password="changeit" certificate-key-
file="/myJbossHome/security/test.keystore" verify-client="false"/>
  </connector>
...
```

Warning

Red Hat recommends that SSLv2, SSLv3, and TLSv1.0 be explicitly disabled in favor of TLSv1.1 or TLSv1.2 in all affected packages.

3.14.1.1.3. 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();
}
```

3.14.1.1.4. 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.HolderOfKe
yIface"
   )
@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.HolderOfKe
yCallbackHandler")
})
public class HolderOfKeyImpl implements HolderOfKeyIface
{
   public String sayHello()
   {
      return "Holder-Of-Key WS-Trust Hello World!";
   }
}
```

3.14.1.1.5. 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.component
s.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
```

3.14.1.1.6. 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
```

3.14.2. 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`**.

3.14.2.1. 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

3.14.2.1.1. 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 designed 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>
```

3.14.2.1.2. SSL Configuration

This web service is using HTTPS, therefore the JBoss EAP server must be configured to provide SSL support in the Undertow subsystem. There are 2 components to SSL configuration:

- Create a certificate keystore.
- Declare an SSL connector in the Undertow subsystem of the JBoss EAP server configuration file.

Here is an example of an SSL connector declaration:

```
<subsystem xmlns="urn:jboss:domain:web:1.4" default-virtual-
```

```
server="default-host" native="false">
  .....
  <connector name="jbws-https-connector" protocol="HTTP/1.1"
scheme="https" socket-binding="https" secure="true" enabled="true">
    <ssl key-alias="tomcat" password="changeit" certificate-key-
file="/myJbossHome/security/test.keystore" verify-client="false"/>
  </connector>
  ...
```

Warning

Red Hat recommends that SSLv2, SSLv3, and TLSv1.0 be explicitly disabled in favor of TLSv1.1 or TLSv1.2 in all affected packages.

3.14.2.1.3. 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();
}
```

3.14.2.1.4. 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!";
   }
}
```

3.14.2.1.5. 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.component
s.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
```

3.14.2.1.6. 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
```

3.14.2.2. 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

3.14.2.2.1. 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 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>
```

jboss-web.xml

```
<?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>
```

jbossws-users.properties

```
# A sample users.properties file for use with the UsersRolesLoginModule
alice=clarinet
```

jbossws-roles.properties

```
# A sample roles.properties file for use with the UsersRolesLoginModule
alice=friend
```

3.14.2.2.2. 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>
```

3.14.2.2.3. 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.STSBearerCal
lbackHandler")
})
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);
    }
}
```

3.14.2.2.4. 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;
    }
}
```

3.14.2.2.5. 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
```

3.14.2.2.6. 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
```

3.14.2.3. 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

3.14.2.3.1. 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();
```

3.14.2.3.2. 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;
                }
            }
        }
    }
}
```

3.14.2.3.3. 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.component
s.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
```

3.14.3. 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.

3.14.3.1. 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

3.14.3.1.1. 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>
```

3.14.3.1.2. 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();
}
```

3.14.3.1.3. 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.OnBehalfOfS
erviceIface"
)

@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.OnBehalfOfC
allbackHandler")
})

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;
   }

}
```

3.14.3.1.4. 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;
    }

}
```

3.14.3.2. 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 incldues:

- OnBehalfOf Web Service Requester Implementation Class

3.14.3.2.1. 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, "myststskey");
    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();
```

3.14.4. 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.

3.14.4.1. 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

3.14.4.1.1. 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>
```

3.14.4.1.2. 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();
}
```

3.14.4.1.3. 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.ActAsServiceIfac
e"
)

@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.ActAsCallbackHan
dler")
})

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;
   }

}
```

3.14.4.1.4. 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;
   }
}
```

3.14.4.1.5. 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;
     }

}
```

3.14.4.1.6. 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.component
s.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
```

3.14.4.1.7. 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
```

3.14.4.2. 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

3.14.4.2.1. 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.SubjectCreatingPolicyIn
terceptor"})
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);
   }
}
```

3.14.4.2.2. 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;
   }
}
```

3.14.4.2.3. 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

3.14.4.2.4. 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();
```

APPENDIX A. REFERENCE MATERIAL

A.1. JAX-RS/RESTEASY ANNOTATIONS

Table A.1. JAX-RS/RESTEasy Annotations

Annotation	Usage
ContentEncoding	Meta annotation that specifies a **`Content-Encoding`** to be applied via the annotated annotation.
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.
Form	This can be used as a value object for incoming/outgoing request/responses.
StringParameterUnmarshallerBinder	Meta-annotation to be placed on another annotation that triggers a **`StringParameterUnmarshaller`** to be applied to a string based annotation injector.
Cache	Set response **`Cache-Control`** header automatically.
NoCache	Set **`Cache-Control`** response header of **`nocache`**.
ClientInterceptor	Identifies an interceptor as a client-side interceptor.
ServerInterceptor	Identifies an interceptor as a server-side interceptor.

Annotation	Usage
NoJackson	Placed on class, parameter, field or method when you don’t want the Jackson provider to be triggered.
ImageWriterParams	An annotation that a resource class can use to pass parameters to the **`IIOImageProvider`**.
DoNotUseJAXBProvider	Put this on a class or parameter when you do not want the JAXB **`MessageBodyReader`**/**`Writer`** used but instead have a more specific provider you want to use to marshal the type.
Formatted	Format XML output with indentations and newlines. This is a JAXB Decorator.
IgnoreMediaTypes	Placed on a type, method, parameter, or field to tell JAXRS not to use JAXB provider for a certain media type
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 JAXB objects.
WrappedMap	Put this on a method or parameter when you want to marshal or unmarshal a map of JAXB objects.
XmlHeader	Sets an XML header for the returned document.
Mapped	A **`JSONConfig`**.
XmlNsMap	A **`JSONToXml`**.

Annotation	Usage
MultipartForm	This can be used as a value object for incoming/outgoing request/responses of the multipart/form-data MIME type.
PartType	Must be used in conjunction with Multipart providers when writing out a List or Map as a **`multipart/*`** type.
XopWithMultipartRelated	This annotation can be used to process/produce incoming/outgoing XOP messages (packaged as multipart/related) to/from JAXB annotated objects.
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.
QueryParam	Allows you to map URI query string parameter or URL form encoded parameter to the method invocation.
CookieParam	Allows you to specify the value of a cookie or object representation of an HTTP request cookie into the method invocation.
DefaultValue	Can be combined with the other **`@*Param`** annotations to define a default value when the HTTP request item does not exist.

Annotation	Usage
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.
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.
Provider	Marks a class to be discoverable as a provider by JAX-RS runtime during a provider scanning phase.
Priority	An annotation to indicate what order a class should be used. Uses an integer parameter with a lower value signifying a higher priority.
GET, POST, PUT, DELETE	An annotation that signifies that the method responds to HTTP **`GET`**, **`POST`**, **`PUT`**, or **`DELETE`** requests.

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 **`/*`**.

Option Name	Default Value	Description
resteasy.scan	false	Automatically scan **`WEB-INF/lib`** JARs and **`WEB-INF/classes`** directory for both **`@Provider`** and JAX-RS resource classes (**`@Path`**, **`@GET`**, **`@POST`**, etc..) and register them.
resteasy.scan.providers	false	Scan for **`@Provider`** classes and register them.
resteasy.scan.resources	false	Scan for JAX-RS 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 JAX-RS 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 JAX-RS resources.
javax.ws.rs.Application	No default	Fully qualified name of **`Application`** class to bootstrap in a spec portable way.

Option Name	Default Value	Description
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`**.
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 JAXB object representations.
resteasy.document.secure.disableDTDs	true	Prohibit DTDs in **`org.w3c.dom.Document`** documents and JAXB object representations.

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 JAX-RS 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</se
rvlet-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>
```

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.
$accepts	*/*	The accepted MIME types sent as the **`Accept`** header. Determined by the **`@Provides`** annotation.

Property	Default Value	Description
$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 JAX-RS 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.

Member	Description
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.
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

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`**.

Parameter	Description
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. JAX-WS TOOLS

wsconsume

wsconsume is a command-line tool provided with JBoss EAP that consumes a WSDL and produces portable JAX-WS 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
  ```

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 JAX-WS or JAXB 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
-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 JAX-WS specification target

Option	Description
-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 JAX-WS 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
  ```

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

Option	Description
-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. JAX-WS COMMON API REFERENCE

Several JAX-WS 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 JAX-WS 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 JAX-WS client, handlers are defined either by using the **`@HandlerChain`** annotation, as in service endpoints, or dynamically, using the JAX-WS 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 JAX-WS 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 via 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 via 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 run-time 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");
}
```

JAX-WS Annotations

The annotations available via the JAX-WS API are defined in JSR-224, which can be found at http://www.jcp.org/en/jsr/detail?id=224. These annotations are in package **`javax.xml.ws`**.

The annotations available via the JWS API are defined in JSR-181. These annotations are in the **`javax.jws`** package.