JBoss Enterprise Application Platform 5

RESTEasy Reference Guide

for use with JBoss Enterprise Application Platform 5

Edition 5.2.0

JBoss Enterprise Application Platform 5 RESTEasy Reference Guide

for use with JBoss Enterprise Application Platform 5
Edition 5.2.0

Eva Kopalova

Petr Penicka

Russell Dickenson

Scott Mumford

Legal Notice

Abstract

A reference guide to RESTEasy for use with JBoss Enterprise Application Platform 5 and its patch releases.

Table of Contents

Table of Contents

CHAPTER 1. OVERVIEW

JSR-000311 JAX-RS is a Java Community Process (JCP) specification that provides a Java API for RESTful Web Services over the HTTP protocol. **RESTEasy** is a portable implementation of the JAX-RS specification that can run in any Servlet container, and integrates tightly with the JBoss Enterprise Application Platform (EAP) to provide improved user experience in the EAP environment. Where JAX-RS is a server-side only specification, RESTEasy brings JAX-RS functions to the client side through the RESTEasy JAX-RS Client Framework, allowing you to map outgoing HTTP requests to remote servers with JAX-RS and interface proxies.

- RESTEasy includes the JAX-RS implementation.

- RESTEasy is portable to any Tomcat or Application Server that runs on Java Development Kit 5.0 or higher.

- The RESTEasy server implementation can be used with Embedded JBoss for junit testing.

- RESTEasy integrates easily with Enterprise JavaBeans (EJB) and the Spring Framework.

- RESTEasy includes a client framework to simplify the process of writing HTTP clients, giving you the opportunity to define more than server bindings.

CHAPTER 2. INSTALLATION/CONFIGURATION

2.1. USING RESTEASY IN YOUR APPLICATION

RESTEasy is deployed as a **WAR** archive, so it must be deployed inside a Servlet container.

You can obtain the **resteasy-jaxrs-war** from the enterprise code source. The file is located in the **/resteasy** folder of the source code archive. Download the archive from the Red Hat Customer Support Portal.

Place your JAX-RS annotated class resources and providers in one or more **JAR**s within **/WEB-INF/lib**. Alternatively, place your raw class files within **/WEB-INF/classes**. By default, RESTEasy is configured to scan **JAR**s and classes within these directories for JAX-RS annotated classes, and to deploy and register them within the system.

RESTEasy is implemented as a **ServletContextListener** and a Servlet, and deployed within a **WAR** file. The **WEB-INF/web.xml** file contains the following:

```xml
<web-app>
    <display-name>Archetype Created Web Application</display-name>
    <!--Set this if you want Resteasy to scan for JAX-RS classes
    <context-param>
        <param-name>resteasy.scan</param-name>
        <param-value>true</param-value>
    </context-param>
    -->

    <!-- set this if you map the Resteasy servlet to something other than
/*
    <context-param>
        <param-name>resteasy.servlet.mapping.prefix</param-name>
        <param-value>/resteasy</param-value>
    </context-param>
    -->
    <!-- to turn on security
    <context-param>
        <param-name>resteasy.role.based.security</param-name>
        <param-value>true</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>
```

The **ResteasyBootstrap** listener initializes some of RESTEasy's basic components and scans for annotation classes that exist in your **WAR** file. It also receives configuration options from <context-param> elements.

These configuration options must be set if the <servlet-mapping> for the RESTEasy Servlet has a URL pattern other than /*. For example, if the URL pattern is:

```
<servlet-mapping>
    <servlet-name>Resteasy</servlet-name>
    <url-pattern>/restful-services/*</url-pattern>
</servlet-mapping>
```

Then the value of *resteasy-servlet.mapping.prefix* must be:

```
<context-param>
    <param-name>resteasy.servlet.mapping.prefix</param-name>
    <param-value>/restful-services</param-value>
</context-param>
```

The available <param-name> values are outlined below with the <param-value> defaults (or expected patterns) listed for reference.

Available <context-param> Parameter Names

resteasy.servlet.mapping.prefix

Defines the URL pattern for the RESTEasy **servlet-mapping**, if not **/***.

resteasy.scan.providers

Scans for @**Provider** classes and registers them. The default value is **false**.

resteasy.scan.resources

Scans for JAX-RS resource classes. The default value is **false**.

resteasy.scan

Scans for both @**Provider** and JAX-RS resource classes (@**Path**, @**GET**, @**POST**, etc.) and registers them.

resteasy.providers

A comma-delimited list of fully-qualified @**Provider** class names you want to register.

resteasy.use.builtin.providers

Determines whether default, built-in @**Provider** classes are registered. The default value is **true**.

resteasy.resources

Comma-delimited list of fully-qualified JAX-RS resource class names you want to register.

`resteasy.jndi.resources`

A comma-delimited list of JNDI names referencing the objects that you want to register as JAX-RS resources.

`javax.ws.rs.core.Application`

Fully-qualified name of the Application class to bootstrap in a spec-portable way.

The **`RilRinRikRaRpRjRikRiRiRi`**~~RilRinRikRaRpRjRikRiRiRi~~ **`ResteasyBootstrap`** listener configures an instance of the **`ResteasyProviderFactory`** and **`Registry`**. You can obtain **`ResteasyProviderFactory`** and **`Registry`** instances from the **`ServletContext`** attributes **`org.jboss.resteasy.spi.ResteasyProviderFactory`** and **`org.jboss.resteasy.spi.Registry`**.

2.2. JAVAX.WS.RS.CORE.APPLICATION

`javax.ws.rs.core.Application` is a standard JAX-RS class that can be implemented to provide information about your deployment. It is a class that lists all JAX-RS root resources and providers.

```
/**
 * Defines the components of a JAX-RS application and supplies additional
 * metadata. A JAX-RS application or implementation supplies a concrete
 * subclass of this abstract class.
 */
public abstract class Application
{
   private static final Set<Object> emptySet = Collections.emptySet();

   /**
    * Get a set of root resource and provider classes. The default
lifecycle
    * for resource class instances is per-request. The default lifecycle
for
    * providers is singleton.
    * <p/>
    * <p>Implementations should warn about and ignore classes that do not
    * conform to the requirements of root resource or provider classes.
    * Implementations should warn about and ignore classes for which
    * {@link #getSingletons()} returns an instance. Implementations MUST
    * NOT modify the returned set.</p>
    *
    * @return a set of root resource and provider classes. Returning null
    *         is equivalent to returning an empty set.
    */
   public abstract Set<Class<?>> getClasses();

   /**
    * Get a set of root resource and provider instances. Fields and
properties
    * of returned instances are injected with their declared dependencies
    * (see {@link Context}) by the runtime prior to use.
    * <p/>
    * <p>Implementations should warn about and ignore classes that do not
```

```
     * conform to the requirements of root resource or provider classes.
     * Implementations should flag an error if the returned set includes
     * more than one instance of the same class. Implementations MUST
     * NOT modify the returned set.</p>
     * <p/>
     * <p>The default implementation returns an empty set.</p>
     *
     * @return a set of root resource and provider instances. Returning
 null
     *          is equivalent to returning an empty set.
     */
    public Set<Object> getSingletons()
    {
       return emptySet;
    }

  }
```

To use Application you must set the Servlet `context-param`, `javax.ws.rs.core.Application`, with a fully-qualified class that implements Application. For example:

```
<context-param>
    <param-name>javax.ws.rs.core.Application</param-name>
    <param-value>com.mycom.MyApplicationConfig</param-value>
</context-param>
```

If you have this set, you should probably turn off automatic scanning as this will probably result in duplicate classes being registered.

2.3. RESTEASYLOGGING

RESTEasy logs various events using **slf4j**.

The slf4j API is intended to serve as a simple facade for various logging APIs, allowing you to plug in the desired implementation at deployment time. By default, RESTEasy is configured to use **Apache log4j**, but you can use any logging provider supported by slf4j.

The initial set of logging categories defined in the framework is listed below. Further logging categories are being added, but these should make it easier to troubleshoot issues.

Table 2.1. Logging Categories

Category	Function
`org.jboss.resteasy.core`	Logs all activity by the core RESTEasy implementation.
`org.jboss.resteasy.plugins.providers`	Logs all activity by RESTEasy entity providers.

Category	Function
`org.jboss.resteasy.plugins.server`	Logs all activity by the RESTEasy server implementation.
`org.jboss.resteasy.specimpl`	Logs all activity by JAX-RS implementing classes.
`org.jboss.resteasy.mock`	Logs all activity by the RESTEasy mock framework.

If you are developing RESTEasy code, the **LoggerCategories** class provides easy access to category names and the various loggers.

CHAPTER 3. USING @PATH AND @GET, @POST, ETC.

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

    @GET
    @Path("/books")
    public String getBooks() {...}

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

    @PUT
    @Path("/book/{isbn}")
    public void addBook(@PathParam("isbn") String id, @QueryParam("name")
String name) {...}

    @DELETE
    @Path("/book/{id}")
    public void removeBook(@PathParam("id") String id {...}


}
```

If you have the RESTEasy Servlet configured and reachable at a root path of
http://myhost.com/services, the requests would be handled by the **Library** class:

- GET http://myhost.com/services/library/books

- GET http://myhost.com/services/library/book/333

- PUT http://myhost.com/services/library/book/333

- DELETE http://myhost.com/services/library/book/333

The @**javax.ws.rs.Path** annotation must exist on either the class or a resource method, or both. If it
exists on both the class and method, the relative path to the resource method is a concatenation of the
class and method.

The @**javax.ws.rs package** contains annotations for each HTTP method. @**GET**, @**POST**, @**PUT**,
@**DELETE**, and @**HEAD**. Place these annotations on public methods that you want to map to the
annotation's HTTP method. If a @**Path** annotation exists on the class, you do not need to annotate the
method you wish to map with @**Path**. Multiple HTTP methods can be used, as long as they can be
distinguished from other methods.

When a method is annotated with @**Path** without a HTTP method being applied, the annotated method
is referred to as a **JAXRSResourceLocator**.

3.1. @PATH AND REGULAR EXPRESSION MAPPINGS

The @**Path** annotation is not limited to simple path expressions. You can also insert regular
expressions into the value of @**Path**. For example:

```
@Path("/resources)
public class MyResource {

    @GET
    @Path("{var:.*}/stuff")
    public String get() {...}
}
```

The following GETs will route to the **getResource()** method:

```
GET /resources/stuff
GET /resources/foo/stuff
GET /resources/on/and/on/stuff
```

The format of the expression is:

```
"{" variable-name [ ":" regular-expression ] "}"
```

Here, **regular-expression** is optional. Where this is not provided, the expression defaults to a wildcard matching of one particular segment, like so:

```
"([]*)"
```

For example:

```
@Path("/resources/{var}/stuff")
```

will match these:

```
GET /resources/foo/stuff
GET /resources/bar/stuff
```

but will not match:

```
GET /resources/a/bunch/of/stuff
```

CHAPTER 4. @PATHPARAM

@PathParam is a parameter annotation which allows you to map variable URI path fragments into your method call.

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

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

This lets you embed variable identification in the URIs of your resources. The previous example shows an **isbn** URI parameter passing information about a particular book we want to access. You can inject into any primitive parameter type, a String, any Java object that takes a String parameter, or a static **valueOf** method that takes a String parameter. For example, if we wanted **isbn** to be a real object, we could write:

```
@GET
@Path("/book/{isbn}")
public String getBook(@PathParam("isbn") ISBN id) {...}


public class ISBN {
    public ISBN(String str) {...}
}
```

Or instead of a public String constructor, we could have a **valueOf** method:

```
public class ISBN {

    public static ISBN valueOf(String isbn) {...}
}
```

4.1. ADVANCED @PATHPARAM AND REGULAR EXPRESSIONS

There are several more complicated uses of @**PathParam**s.

You are allowed to specify one or more @**PathParam**s embedded in one URI segment. For example:

```
1. @Path("/aaa{param}bbb")
2. @Path("/{name}-{zip}")
3. @Path("/foo{name}-{zip}bar")
```

So, a URI of the form "/aaa111bbb" would match the first specified parameter. "/bill-02115" would match the second, and "foobill-02115bar" would match the third.

In Section 3.1, "@Path and regular expression mappings", we mentioned that regular expressions can be used within **@Path** values, like so:

```
@GET
@Path("/aaa{param:b+}/{many:.*}/stuff")
public String getIt(@PathParam("param") String bs, @PathParam("many")
String many) {...}
```

With the **@Path** defined here, the request **GET /aaabb/some/stuff** would have a **"param"** value of **bb**, and a **"many"** value of **some**. The request **GET /aaab/a/lot/of/stuff** would have a **"param"** value of **b**, and a **"many"** value of **a/lot/of**.

4.2. @PATHPARAM AND PATHSEGMENT

The specification has a very simple abstraction for examining a fragment of the URI path being invoked on **javax.ws.rs.core.PathSegment**:

```
public interface PathSegment {

    /**
     * Get the path segment.
     * <p>
     * @return the path segment
     */
    String getPath();
    /**
     * Get a map of the matrix parameters associated with the path segment
     * @return the map of matrix parameters
     */
    MultivaluedMap<String, String> getMatrixParameters();

}
```

RESTEasy can inject a **PathSegment** instead of a value with your **@PathParam**.

```
@GET
@Path("/book/{id}")
public String getBook(@PathParam("id") PathSegment id) {...}
```

This is particularly useful when you have multiple **@PathParam**s that use *matrix parameters*. Matrix parameters are an arbitrary set of name-value pairs embedded in a URI path segment. The **PathSegment** object fives you access to these parameters. See Chapter 7, *@MatrixParam* for further information.

An example of a matrix parameter:

```
GET http://host.com/library/book;name=EJB 3.0;author=Bill Burke
```

A matrix parameter represents a resource that can be addressed by its attributes as well as its raw ID.

CHAPTER 5. @QUERYPARAM

The @QueryParam annotation allows you to map a URI query string parameter or URL form-encoded parameter onto your method invocation.

```
GET /books?num=5
```

```
@GET
public String getBooks(@QueryParam("num") int num) {
...
}
```

Since RESTEasy is built on a Servlet, it cannot distinguish between URL query strings and URL form-encoded parameters. As with @**PathParam**, the type of parameter can be a primitive, a String, or a class with a String constructor or static **valueOf()** method.

CHAPTER 6. @HEADERPARAM

The @**HeaderParam** annotation allows you to map a request HTTP header onto your method invocation.

```
GET /books?num=5
```

```
@GET
public String getBooks(@HeaderParam("From") String from) {
...
}
```

As with @**PathParam**, the type of parameter can be a primitive, a String, or a class with a String constructor or static **valueOf()** method. For example, **MediaType** has a **valueOf()** method, so you could do the following:

```
@PUT
public void put(@HeaderParam("Content-Type") MediaType contentType,
...)
```

CHAPTER 7. @MATRIXPARAM

Matrix parameters are an arbitrary set of name-value pairs embedded in a URI path segment. An example of a matrix parameter is:

```
GET http://host.com/library/book;name=EJB 3.0;author=Bill Burke
```

Matrix parameters represent resources that can be addressed by their attributes as well as their raw ID. The @**MatrixParam** annotation lets you injext URI matrix parameters into your method invocation.

```
    @GET
    public String getBook(@MatrixParam("name") String name,
@MatrixParam("author") String author) {...}
```

One limitation with @**MatrixParam** is that the current version of the specification does not resolve. If, for example, the same **MatrixParam** exists simultaneously in different path segments, at present we recommend using **PathParam** combined with **PathSegment**.

CHAPTER 8. @COOKIEPARAM

The **@CookieParam** annotation allows you to inject the value of a cookie, or an object representation of an HTTP request cookie, into your method invocation.

```
GET /books?num=5
```

```
@GET
public String getBooks(@CookieParam("sessionid") int id) {
...
}

@GET
public String getBooks(@CookieParam("sessionid")
javax.ws.rs.core.Cookie id) {...}
```

As with **@PathParam,** the type of parameter can be a primitive, a String, or a class with a String constructor or static **valueOf()** method. You can also get an object representation of the cookie via the **javax.ws.rs.core.Cookie** class.

CHAPTER 9. @FORMPARAM

When the input request body is of the type **application/x-www-form-urlencoded** (that is, an HTML form), you can inject individual form parameters from the request body into method parameter values.

```
<form method="POST" action="/resources/service">
First name:
<input type="text" name="firstname">
<br>
Last name:
<input type="text" name="lastname">
</form>
```

If you post using this form, the service would look as follows:

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

    @Path("/resources/service")
    @POST
    public void addName(@FormParam("firstname") String first,
@FormParam("lastname") String last) {...}
```

You cannot combine **@FormParam** with the default **application/x-www-form-urlencoded** that unmarshalls to a **MultivaluedMap<String, String>** That is, the following would be illegal:

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

    @Path("/resources/service")
    @POST
    @Consumes("application/x-www-form-urlencoded")
    public void addName(@FormParam("firstname") String first,
MultivaluedMap<String, String> form) {...}
```

CHAPTER 10. @FORM

This is a RESTEasy-specific annotation that allows you to reuse any @*Param annotation within an injected class. RESTEasy instantiates the class and injects values into any annotated @*Param or @Context property. This is useful if you have many parameters on your method and you want to condense them into a value object.

```
public class MyForm {

    @FormParam("stuff")
    private int stuff;


    @HeaderParam("myHeader")
    private String header;


    @PathParam("foo")
    public void setFoo(String foo) {...}
}



@POST
@Path("/myservice")
public void post(@Form MyForm form) {...}
```

When someone posts to /myservice, RESTEasy instantiates an instance of MyForm and injects the form parameter stuff into the stuff field, the header myheader into the header field, and call the setFoo method with the @PathParam variable of foo.

CHAPTER 11. @DEFAULTVALUE

@DefaultValue is a parameter annotation that can be combined with any other **@*Param** annotations to define a default value where the HTTP request item does not exist.

```
@GET
public String getBooks(@QueryParam("num") @DefaultValue("10") int num)
{...}
```

CHAPTER 12. @ENCODED AND ENCODING

JAX-RS allows you to get encoded or decoded @***Param**s and specify path definitions and parameter names using encoded or decoded strings.

The @`javax.ws.rs.Encoded` annotation can be used on a class, method, or parameter. By default, injected @**PathParam** and @**QueryParam** are decoded. Adding the @**Encoded** annotation means that the value of these parameters will be provided in encoded form.

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

   @Path("/{param}")
   @GET
   public String get(@PathParam("param") @Encoded String param) {...}
```

In the previous example, the value of the @**PathParam** injected into the **param** of the **get()** method will be URL encoded. Adding the @**Encoded** annotation as a parameter annotation triggers this effect.

You can also use the @**Encoded** annotation on the entire method and any combination of @**QueryParam** or @**PathParam**'s values will be encoded.

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

   @Path("/{param}")
   @GET
   @Encoded
   public String get(@QueryParam("foo") String foo, @PathParam("param")
String param) {}
}
```

In this example, the values of the **foo** query parameter and the **param** path parameter will be injected as encoded values.

You can also set the default to be encoded for the entire class.

```
@Path("/")
@Encoded
public class ClassEncoded {

   @GET
   public String get(@QueryParam("foo") String foo) {}
}
```

The @**Path** annotation has an attribute called *encode*. This controls whether the literal part of the value supplied (that is, the characters that are not part of a template variable) are URL-encoded. If **true**, any characters in the URI template that are not valid will be automatically encoded. If **false**, then all characters must be valid URI characters. By default, the **encode** attribute is set to **true**. (You can also encode the characters by hand.)

```
@Path(value="hello%20world", encode=false)
```

As with @**Path.encode()**, this controls whether the specified query parameter name should be

encoded by the container before it tries to find the query parameter in the request.

```
@QueryParam(value="hello%20world", encode=false)
```

CHAPTER 13. @CONTEXT

The **@Context** annotation allows you to inject instances of **javax.ws.rs.core.HttpHeaders**, **javax.ws.rs.core.UriInfo**, **javax.ws.rs.core.Request**, **javax.servlet.HttpServletRequest**, **javax.servlet.HttpServletResponse**, **javax.servlet.ServletConfig**, **javax.servlet.ServletContext**, and **javax.ws.rs.core.SecurityContext** objects.

CHAPTER 14. JAX-RS RESOURCE LOCATORS AND SUB RESOURCES

Resource classes can partially process a request and then provide another *sub-resource* object to process the remainder of the request. For example:

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

   @Path("/customers/{id}")
   public Customer getCustomer(@PathParam("id") int id) {
      Customer cust = ...; // Find a customer object
      return cust;
   }
}


public class Customer {

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

   @Path("/address")
   public String getAddress() {...}

}
```

Resource methods with a **@Path** annotation and no HTTP method are considered *sub-resource locators*. They provide an object that can process the request. In the previous example code, **ShoppingStore** is a root resource because its class is annotated with @**Path**. The **getCustomer()** is a sub-resource locator method.

If the client invoked the following:

```
GET /customer/123
```

Then the **ShoppingStore.getCustomer()** method would be invoked first. This method provides a **Customer** object that can service the request. The HTTP request will be dispatched to the **Customer.get()** method. Another example is:

```
GET /customer/123/address
```

In this request, again, first the **ShoppingStore.getCustomer()** method is invoked. A **Customer** object is returned, and the rest of the request is dispatched to the **Customer.getAddress()** method.

Another interesting feature of sub-resource locators is that the locator method result is dynamically processed at runtime in order to determine how the request should be dispatched. This means that the **ShoppingStore.getCustomer()** method does not have to declare any specific type.

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

```
    @Path("/customers/{id}")
    public java.lang.Object getCustomer(@PathParam("id") int id) {
        Customer cust = ...; // Find a customer object
        return cust;
    }
}


public class Customer {

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

    @Path("/address")
    public String getAddress() {...}

}
```

In the previous example, **getCustomer()** returns a **java.lang.Object**. Per request, at runtime, the JAX-RS server will determine how to dispatch the request based on the object returned by **getCustomer()**. This can be useful in certain situations.

For example, say you have a class hierarchy for your customers. **Customer** is the abstract base, and **CorporateCustomer** and **IndividualCustomer** are subclasses. In this case, your **getCustomer()** method might perform a Hibernate polymorphic query without requiring any understanding of the concrete class it queries, or the content returned.

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

    @Path("/customers/{id}")
    public java.lang.Object getCustomer(@PathParam("id") int id) {
        Customer cust = entityManager.find(Customer.class, id);
        return cust;
    }
}


public class Customer {

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

    @Path("/address")
    public String getAddress() {...}

}

public class CorporateCustomer extendsCustomer {

    @Path("/businessAddress")
    public String getAddress() {...}

}
```

-

CHAPTER 15. JAX-RS CONTENT NEGOTIATION

The HTTP protocol has built-in content negotiation headers that allow the client and server to specify the type of content that they transfer, and the type of content they prefer to receive. The server declares content preferences via the @**Produces** and @**Consumes** headers.

@**Consumes** is an array of media types that a particular resource or resource method consumes. For example:

```
@Consumes("text/*")
@Path("/library")
public class Library {

@POST
public String stringBook(String book) {...}


@Consumes("text/xml")
@POST
public String jaxbBook(Book book) {...}
```

When a client makes a request, JAX-RS first locates all methods that match the path. It then sorts objects based on the content-type header sent by the client. If a client sends the following:

```
POST /library
content-type: text/plain

this is a nice book
```

The **stringBook()** method is invoked, because it matches the default **text/*** media type. If the client sends XML, like so:

```
POST /library
content-type: text/xml

<book name="EJB 3.0" author="Bill Burke"/>
```

Then the **jaxbBook()** method is invoked.

@**Produces** is used to map a client request and match it with the client's **Accept** header. The *Accept HTTP* header is sent by the client, and defines the media types that the client prefers to receive from the server.

```
@Produces("text/*")
@Path("/library")
public class Library {

@GET
@Produces("application/json")
public String getJSON() {...}


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

So, if the client sends:

```
GET /library
Accept: application/json
```

The **getJSON()** method would be invoked.

@Consumes and **@Produces** can support multiple media types by presenting them in a list. The client's **Accept** header can also list multiple media types to receive. More specific media types are selected first. The **Accept** header (or **@Produces** or **@Consumes**) can also specify weighted preferences that will match requests with resource methods. (This is best explained in Section 14.1 of RFC 2616.) RESTEasy provides support for this more complex method of content negotiation.

An alternative method used by JAX-RS is a combination of media-type, content-language, and content encoding, in addition to etags, last modified headers, and other pre-conditions. This is a more complex form of content negotiation, performed programmatically by the application developer via the **javax.ws.rs.Variant**, **VarianListBuilder**, and **Request** objects. **Request** is injected using the **@Context** annotation. (For more information, read the JavaDoc.)

CHAPTER 16. CONTENT MARSHALLING/PROVIDERS

16.1. DEFAULT PROVIDERS AND DEFAULT JAX-RS CONTENT MARSHALLING

RESTEasy can automatically marshal and unmarshal several different message body types.

Table 16.1. Message Body Types

Media Types	Java Type
application/*+xml, text/*+xml, application/*+json, application/*+fastinfoset, application/atom+*	JaxB annotated classes
/	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

16.2. CONTENT MARSHALLING WITH @PROVIDER CLASSES

The JAX-RS specification lets you plug in your own readers and writers for request or response bodies. To do so, annotate a class with **@Provider** and specify the **@Produces** types for a reader. You must also implement a **MessageBodyReader** and a **MessageBodyWriter** interface.

16.3. PROVIDERS UTILITY CLASS

javax.ws.rs.ext.Providers is a simple injectable interface that lets you locate **MessageBodyReaders**, **MessageBodyWriters**, **ContextResolvers** and **ExceptionMappers**. It also lets you implement multi-part providers (content types that embed other content types).

```
public interface Providers
{

   /**
    * Get a message body reader that matches a set of criteria. The set of
    * readers is first filtered by comparing the supplied value of
    * {@code mediaType} with the value of each reader's
```

```
     * {@link javax.ws.rs.Consumes}, ensuring the supplied value of
     * {@code type} is assignable to the generic type of the reader, and
     * eliminating those that do not match.
     * The list of matching readers is then ordered with those with the
best
     * matching values of {@link javax.ws.rs.Consumes} (x/y > x/* > */*)
     * sorted first. Finally, the
     * {@link MessageBodyReader#isReadable}
     * method is called on each reader in order using the supplied criteria
and
     * the first reader that returns {@code true} is selected and returned.
     *
     * @param type        the class of object that is to be written.
     * @param mediaType   the media type of the data that will be read.
     * @param genericType the type of object to be produced. E.g. if the
     *                    message body is to be converted into a method
parameter, this will be
     *                    the formal type of the method parameter as
returned by
     *                    <code>Class.getGenericParameterTypes</code>.
     * @param annotations an array of the annotations on the declaration of
the
     *                    artifact that will be initialized with the
produced instance. E.g. if the
     *                    message body is to be converted into a method
parameter, this will be
     *                    the annotations on that parameter returned by
     *                    <code>Class.getParameterAnnotations</code>.
     * @return a MessageBodyReader that matches the supplied criteria or
null
     *         if none is found.
     */
    <T> MessageBodyReader<T> getMessageBodyReader(Class<T> type,
                                                  Type genericType,
Annotation annotations[], MediaType mediaType);

    /**
     * Get a message body writer that matches a set of criteria. The set of
     * writers is first filtered by comparing the supplied value of
     * {@code mediaType} with the value of each writer's
     * {@link javax.ws.rs.Produces}, ensuring the supplied value of
     * {@code type} is assignable to the generic type of the reader, and
     * eliminating those that do not match.
     * The list of matching writers is then ordered with those with the
best
     * matching values of {@link javax.ws.rs.Produces} (x/y > x/* > */*)
     * sorted first. Finally, the
     * {@link MessageBodyWriter#isWriteable}
     * method is called on each writer in order using the supplied criteria
and
     * the first writer that returns {@code true} is selected and returned.
     *
     * @param mediaType   the media type of the data that will be written.
     * @param type        the class of object that is to be written.
     * @param genericType the type of object to be written. E.g. if the
     *                    message body is to be produced from a field,
```

```
this will be
     *                      the declared type of the field as returned by
     *                      <code>Field.getGenericType</code>.
     * @param annotations an array of the annotations on the declaration of
the
     *                      artifact that will be written. E.g. if the
     *                      message body is to be produced from a field,
this will be
     *                      the annotations on that field returned by
     *                      <code>Field.getDeclaredAnnotations</code>.
     * @return a MessageBodyReader that matches the supplied criteria or
null
     *           if none is found.
     */
    <T> MessageBodyWriter<T> getMessageBodyWriter(Class<T> type,
                                                  Type genericType,
Annotation annotations[], MediaType mediaType);

    /**
     * Get an exception mapping provider for a particular class of
exception.
     * Returns the provider whose generic type is the nearest superclass of
     * {@code type}.
     *
     * @param type the class of exception
     * @return an {@link ExceptionMapper} for the supplied type or null if
none
     *           is found.
     */
    <T extends Throwable> ExceptionMapper<T> getExceptionMapper(Class<T>
type);

    /**
     * Get a context resolver for a particular type of context and media
type.
     * The set of resolvers is first filtered by comparing the supplied
value of
     * {@code mediaType} with the value of each resolver's
     * {@link javax.ws.rs.Produces}, ensuring the generic type of the
context
     * resolver is assignable to the supplied value of {@code contextType},
and
     * eliminating those that do not match. If only one resolver matches
the
     * criteria then it is returned. If more than one resolver matches then
the
     * list of matching resolvers is ordered with those with the best
     * matching values of {@link javax.ws.rs.Produces} (x/y > x/* > */*)
     * sorted first. A proxy is returned that delegates calls to
     * {@link ContextResolver#getContext(java.lang.Class)} to each matching
context
     * resolver in order and returns the first non-null value it obtains or
null
     * if all matching context resolvers return null.
     *
     * @param contextType the class of context desired
```

```
    * @param mediaType    the media type of data for which a context is
required.
    * @return a matching context resolver instance or null if no matching
    *         context providers are found.
    */
   <T> ContextResolver<T> getContextResolver(Class<T> contextType,
                                             MediaType mediaType);
}
```

You can inject an instance of **Providers** into **MessageBodyReader** or **MessageBodyWriter** like so:

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

private @Context Providers providers;

...

}
```

CHAPTER 17. JAXB PROVIDERS

RESTEasy includes support for marshaling and unmarshaling JAXB annotated classes. Multiple JAXB Providers are included with RESTEasy to address the subtle differences between classes generated by XJC and classes that are annotated with **@XmlRootElement**, or work with **JAXBElement** classes directly.

When using the JAX-RS API in development, the provider to be invoked is selected transparently. This chapter describes the providers best-suited for a variety of configurations if you want to access the providers *directly*.

RESTEasy selects a JAXB Provider when a parameter type (return type) is an object annotated with JAXB annotations (for example, **@XmlRootEntity** or **@XmlType**), or a **JAXBElement**. The resource class (resource method) will be annotated with either **@Consumes** or **@Produces**, and contain one or more of the following values:

- **text/*+xml**

- **application/*+xml**

- **application/*+fastinfoset**

- **application/*+json**

RESTEasy selects different providers based on the return type used in the resource. This section describes the workings of the selection process.

Classes annotated with **@XmlRootElement** are handled with the **JAXBXmlRootElementProvider**. This provider handles basic marshaling and unmarshaling of custom JAXB entities.

Classes that are generated by XJC do not usually contain an **@XmlRootElement** annotation. To be marshaled, they must be wrapped in an instance of **JAXBElement**. This usually involves invoking a method named **ObjectFactory** on the class, which serves as the **XmlRegistry**.

The **JAXBXmlTypeProvider** provider is selected when the class is annotated with an **XmlType** annotation and not an **XmlRootElement** annotation. The provider attempts to locate the **XmlRegistry** for the target class. By default, a JAXB implementation creates a class called **ObjectFactory** and is located in the same package as the target class. **ObjectFactory** contains a **create** method that takes the object instance as a parameter. For example, if the target type is called **Contact**, then the **ObjectFactory** class will have a method:

```
public JAXBElement createContact(Contact value) {..
```

If your resource works with the **JAXBElement** class directly, the RESTEasy runtime will select the **JAXBElementProvider**. This provider examines the **ParameterizedType** value of the **JAXBElement** in order to select the appropriate **JAXBContext**.

17.1. JAXB DECORATORS

RESTEasy's JAXB providers can decorate **marshaler** and **Unmarshaler** instances. Add an annotation that triggers the decoration **marshaler** or **Unmarshaler**. The decorators can perform tasks such as setting **marshaler** or **Unmarshaler** properties and setting up validation.

As an example, say you want to create an annotation that will trigger *pretty-printing* of an XML document. In raw JAXB, we would set a property on the `marshaler` of `marshaler.JAXB_FORMATTED_OUTPUT`. Instead, let us write a *marshaler decorator*.

First, define an annotation:

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

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

For this to work, you must annotate the @`Pretty` annotation with a meta-annotation named @`Decorator`. The `target()` attribute must be the JAXB `marshaler` class. Next, we will write the `processor()` attribute class.

```
import org.jboss.resteasy.core.interception.DecoratorProcessor;
import org.jboss.resteasy.annotations.DecorateTypes;

import javax.xml.bind.marshaler;
import javax.xml.bind.PropertyException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.Produces;
import java.lang.annotation.Annotation;

/**
 * @author <a href="mailto:bill@burkecentral.com">Bill Burke</a>
 * @version $Revision: 1 $
 */
@DecorateTypes({"text/*+xml", "application/*+xml"})
public class PrettyProcessor implements DecoratorProcessor<marshaler,
Pretty>
{
   public marshaler decorate(marshaler target, Pretty annotation,
                 Class type, Annotation[] annotations, MediaType
mediaType)
   {
      target.setProperty(marshaler.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
   }
}
```

The `processor` implementation must implement the `DecoratorProcessor` interface, and should also be annotated with @`DecorateTypes`. This annotation specifies the media types that the processor can work with.

Now that we have defined our annotation and our `Processor`, we can use it on our JAX-RS resource methods or JAXB types like so:

```
@GET
@Pretty
@Produces("application/xml")
public SomeJAXBObject get() {...}
```

If this is confusing, check the RESTEasy source code for information about implementing **@XmlHeader**.

17.2. PLUGGABLE JAXBCONTEXTS WITH CONTEXTRESOLVERS

We do not recommend using this feature unless you are familiar with the principles involved.

By default, RESTEasy creates and caches **JAXBContext** instances per class type depending on the class you are marshaling or unmarshaling. If you do not want RESTEasy to create **JAXBContext**s, you can plug in your own by implementing an instance of **javax.ws.rs.ext.ContextResolver**.

```
  public interface ContextResolver<T>
  {
T getContext(Class<?> type);
  }

  @Provider
  @Produces("application/xml")
  public class MyJAXBContextResolver implements
ContextResolver<JAXBContext>
  {
JAXBContext getContext(Class<?> type)
{
    if (type.equals(WhateverClassIsOverridedFor.class)) return
JAXBContext.newInstance()...;
}
  }
```

You must provide a **@Produces** annotation to specify the types of media intended for the context. You must also implement **ContextResolver<JAXBContext>**. This helps the runtime match the correct context resolver. You must also annotate the **ContextResolver** class with **@Provider**.

There are several ways to make this **ContextResolver** available.

1. return it as a class or instance from a **javax.ws.rs.core.Application** implementation.

2. list it as a provider with **resteasy.providers**.

3. let RESTEasy automatically scan for it within your **WAR** file. (See the Configuration Guide for more details.)

4. add it manually via **ResteasyProviderFactory.getInstance().registerProvider(Class)** or **registerProviderInstance(Object)**.

17.3. JAXB AND XML PROVIDER

RESTEasy provides the required JAXB provider support for XML. It has several additional annotations to make application coding simpler.

17.3.1. @XmlHeader and @Stylesheet

To set an XML header when you output XML documents, use the **@org.jboss.resteasy.annotations.providers.jaxb.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;
   }
}
```

Here, the **@XmlHeader** forces an **xml-stylesheet** header on the XML output. The same result can be obtained by placing the header on the **Thing** class. Read the JavaDocs for further information regarding the substitution values provided by RESTEasy.

RESTEasy also has a convenient annotation for stylesheet headers. For example:

```
@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;
    }
}
```

17.4. JAXB AND JSON PROVIDER

RESTEasy lets you marshal JAXB annotated POJOs to and from JSON with the Jettison JSON library. You can find more information about Jettison at http://jettison.codehaus.org/.

Jettison has two mapping formats: the default *Jettison Mapped Convention* format, and BadgerFish.

For example, consider this JAXB class:

```
@XmlRootElement(name = "book")
public class Book {

    private String author;
    private String ISBN;
    private String title;

    public Book() {
    }

    public Book(String author, String ISBN, String title) {
        this.author = author;
        this.ISBN = ISBN;
        this.title = title;
    }

    @XmlElement
    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    @XmlElement
    public String getISBN() {
        return ISBN;
    }
}
```

```
    public void setISBN(String ISBN) {
        this.ISBN = ISBN;
    }

    @XmlAttribute
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

The JAXB **Book** class would be marshaled to JSON using the BadgerFish Convention:

```
{"book":
   {
      "@title":"EJB 3.0",
      "author":{"$":"Bill Burke"},
      "ISBN":{"$":"596529260"}
   }
}
```

Element values are associated with a map. To find the value of the element, you must access the $ variable. You could access the book like this, in JavaScript:

```
var data = eval("(" + xhr.responseText + ")");
document.getElementById("zone").innerHTML = data.book.@title;
document.getElementById("zone").innerHTML += data.book.author.$;
```

To use the BadgerFish Convention you must use the `@org.jboss.resteasy.annotations.providers.jaxb.json.BadgerFish` annotation either on the JAXB class you are marshaling or unmarshaling, or on the JAX-RS resource method or parameter:

```
@BadgerFish
@XmlRootElement(name = "book")
public class Book {...}
```

To return a **book** on the JAX-RS method without polluting your JAXB classes with RESTEasy annotations, you can add the annotation to the JAX-RS method instead:

```
@BadgerFish
@GET
public Book getBook(...) {...}
```

If your input is a **Book**, place it on the parameter:

```
@POST
public void newBook(@BadgerFish Book book) {...}
```

–

The default Jettison Mapped Convention returns the following JSON:

```
{ "book" :
    {
       "@title":"EJB 3.0",
       "author":"Bill Burke",
       "ISBN":596529260
    }
}
```

Note that `title` is prefixed with the @ character. Unlike the BadgerFish convention, this does not represent the value of element text, which makes it simpler (and a sensible default). To access this in JavaScript:

```
var data = eval("(" + xhr.responseText + ")");
document.getElementById("zone").innerHTML = data.book.@title;
document.getElementById("zone").innerHTML += data.book.author;
```

The Mapped Convention lets you adjust the JAXB mapping with the `@org.jboss.resteasy.annotations.providers.jaxb.json.Mapped` annotation. With this, you can provide an XML namespace to JSON namespace mapping. For example, if you define your JAXB namespace within your `package-info.java` class like so:

```
@javax.xml.bind.annotation.XmlSchema(namespace="http://jboss.org/books")
package org.jboss.resteasy.test.books;
```

You must define a JSON-to-XML namespace mapping, or you will receive an exception:

```
java.lang.IllegalStateException: Invalid JSON namespace:
  http://jboss.org/books
at org.codehaus.jettison.mapped.MappedNamespaceConvention
  .getJSONNamespace(MappedNamespaceConvention.java:151)
at org.codehaus.jettison.mapped.MappedNamespaceConvention
  .createKey(MappedNamespaceConvention.java:158)
at org.codehaus.jettison.mapped.MappedXMLStreamWriter
  .writeStartElement(MappedXMLStreamWriter.java:241)
```

The @**Mapped** annotation fixes this problem. Place the @**Mapped** annotation on your JAXB classes, your JAX-RS resource method, or on the parameter that you are unmarshaling.

```
import org.jboss.resteasy.annotations.providers.jaxb.json.Mapped;
import org.jboss.resteasy.annotations.providers.jaxb.json.XmlNsMap;

...

@GET
@Produces("application/json")
@Mapped(namespaceMap = {
       @XmlNsMap(namespace = "http://jboss.org/books", jsonName =
"books")
})
public Book get() {...}
```

You can also force **@XmlAttributes** to be marshaled as **XMLElements**.

```
          @Mapped(attributeAsElements={"title"})
          @XmlRootElement(name = "book")
          public class Book {...}
```

To return a **book** on the JAX-RS method without polluting your JAXB classes with RESTEasy annotations, add the annotation to the JAX-RS method:

```
          @Mapped(attributeAsElements={"title"})
          @GET
          public Book getBook(...) {...}
```

If your input is a **Book**, place it on the parameter:

```
 @POST
 public void newBook(@Mapped(attributeAsElements={"title"}) Book book)
{...}
```

17.5. JAXB AND FASTINFOSET PROVIDER

RESTEasy supports the **Fastinfoset** MIME type through the use of JAXB annotated classes. **Fastinfoset** documents serialize and parse more quickly, and are smaller in size, than logically-equivalent XML documents, so they can be used where size and processing time of XML documents is problematic. It is configured in the same way as the XML JAXB provider.

17.6. ARRAYS AND COLLECTIONS OF JAXB OBJECTS

RESTEasy automatically marshals arrays, **java.util.Sets**, and **java.util.Lists** of JAXB objects to and from XML, JSON, **Fastinfoset**, and other RESTEasy JAXB mappers.

```
 @XmlRootElement(name = "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("/")
public class MyResource
{
   @PUT
   @Path("array")
   @Consumes("application/xml")
   public void putCustomers(Customer[] customers)
   {
      Assert.assertEquals("bill", customers[0].getName());
      Assert.assertEquals("monica", customers[1].getName());
   }

   @GET
   @Path("set")
   @Produces("application/xml")
   public Set<Customer> getCustomerSet()
   {
      HashSet<Customer> set = new HashSet<Customer>();
      set.add(new Customer("bill"));
      set.add(new Customer("monica"));

      return set;
   }

   @PUT
   @Path("list")
   @Consumes("application/xml")
   public void putCustomers(List<Customer> customers)
   {
      Assert.assertEquals("bill", customers.get(0).getName());
      Assert.assertEquals("monica", customers.get(1).getName());
   }
}
```

The resource above publishes and receives JAXB objects. We assume that these are wrapped in a collection element like the following:

```
 <collection>
<customer><name>bill</name></customer>
<customer><name>monica</name></customer>
 <collection>
```

You can change the namespace URI, namespace tag, and collection element name by using the **@org.jboss.resteasy.annotations.providers.jaxb.Wrapped** annotation on a parameter or method:

```
@Target({ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Wrapped
{
String element() default "collection";

String namespace() default "http://jboss.org/resteasy";
```

```
String prefix() default "resteasy";
 }
```

So, if we wanted to output the following XML:

```
 <foo:list xmlns:foo="http://foo.org">
<customer><name>bill</name></customer>
<customer><name>monica</name></customer>
 </foo:list>
```

We would use the **@Wrapped** annotation as follows:

```
@GET
@Path("list")
@Produces("application/xml")
@Wrapped(element="list", namespace="http://foo.org", prefix="foo")
public List<Customer> getCustomerSet()
{
   List<Customer> list = new ArrayList<Customer>();
   list.add(new Customer("bill"));
   list.add(new Customer("monica"));

   return list;
}
```

17.6.1. JSON and JAXB Collections/Arrays

RESTEasy supports using collections with JSON. It encloses lists, sets, or arrays of returned JAXB objects in a simple JSON array. For example:

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public static class Foo
{
   @XmlAttribute
   private String test;

   public Foo()
   {
   }

   public Foo(String test)
   {
      this.test = test;
   }

   public String getTest()
   {
      return test;
   }

   public void setTest(String test)
```

```
    {
        this.test = test;
    }
}
```

A List or Array of the **Foo** class would be represented in JSON like so:

```
[{"foo":{"@test":"bill"}},{"foo":{"@test":"monica}"}}]
```

It would also expect this format when receiving input.

17.7. MAPS OF JAXB OBJECTS

RESTEasy automatically marshals maps of JAXB objects to and from XML, JSON, **Fastinfoset**, and other JAXB mappers. Your parameter or method return type must be generic, with a String as the key and the JAXB object's type.

```
@XmlRootElement(namespace = "http://foo.com")
public static class Foo
{
    @XmlAttribute
    private String name;

    public Foo()
    {
    }

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

    public String getName()
    {
        return name;
    }
}

@Path("/map")
public static class MyResource
{
    @POST
    @Produces("application/xml")
    @Consumes("application/xml")
    public Map<String, Foo> post(Map<String, Foo> map)
    {
        Assert.assertEquals(2, map.size());
        Assert.assertNotNull(map.get("bill"));
        Assert.assertNotNull(map.get("monica"));
        Assert.assertEquals(map.get("bill").getName(), "bill");
        Assert.assertEquals(map.get("monica").getName(), "monica");
        return map;
    }
}
```

This resource publishes and receives JAXB objects within a map. By default, they are wrapped in a **map** element in the default namespace. Each **map** element has zero or more **entry** elements with a **key** attribute.

```
  <map>
<entry key="bill" xmlns="http://foo.com">
    <foo name="bill"/>
</entry>
<entry key="monica" xmlns="http://foo.com">
    <foo name="monica"/>
</entry>
  </map>
```

You can change the namespace URI, namespace prefix and map, entry, and key element and attribute names by using the **@org.jboss.resteasy.annotations.providers.jaxb.WrappedMap** annotation on a parameter or method.

```
@Target({ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface WrappedMap
{
/**
 * map element name
 */
String map() default "map";

/**
 * entry element name *
 */
String entry() default "entry";

/**
 * entry's key attribute name
 */
String key() default "key";

String namespace() default "";

String prefix() default "";
}
```

So, to output the following XML:

```
  <hashmap>
<hashentry hashkey="bill" xmlns:foo="http://foo.com">
    <foo:foo name="bill"/>
</hashentry>
  </map>
```

We would use the @WrappedMap annotation as follows:

```
@Path("/map")
public static class MyResource
{
    @GET
```

```
    @Produces("application/xml")
    @WrappedMap(map="hashmap", entry="hashentry", key="hashkey")
    public Map<String, Foo> get()
    {
        ...
        return map;
    }
```

17.7.1. JSON and JAXB maps

RESTEasy supports the use of maps with JSON. It encloses returned JAXB objects within simple JSON maps. For example:

```
    @XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public static class Foo
{
    @XmlAttribute
    private String test;

    public Foo()
    {
    }

    public Foo(String test)
    {
        this.test = test;
    }

    public String getTest()
    {
        return test;
    }

    public void setTest(String test)
    {
        this.test = test;
    }
}
```

This a List or array of this Foo class would be represented in JSON like this:

```
  { "entry1" : {"foo":{"@test":"bill"}}, "entry2" : {"foo":
{"@test":"monica}"}}}
```

It also expects this format for input

17.7.2. Possible Problems with Jettison Provider

If you have the `resteasy-jackson-provider-xxx.jar` in your classpath, the Jackson JSON provider will be triggered. This is problematic for code that depends upon the Jettison JAXB or JSON provider. To correct this, you must either remove Jackson from your `WEB-INF/lib` or classpath, or use the `@NoJackson` annotation on your JAXB classes.

17.8. INTERFACES, ABSTRACT CLASSES, AND JAXB

Some object models use abstract classes and interfaces heavily. JAXB does not work with interfaces that are root elements, and RESTEasy cannot unmarshal parameters that are interfaces or raw abstract classes because it lacks the information required to create a **JAXBContext**. For example:

```
public interface IFoo {}

@XmlRootElement
public class RealFoo implements IFoo {}

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

@PUT
@Consumes("application/xml")
public void put(IFoo foo) {...}
 }
```

In this example, RESTEasy would display an error ("Cannot find MessageBodyReader for..." or similar) because RESTEasy does not know that implementations of **IFoo** are JAXB classes, so it cannot create a **JAXBContext** for **IFoo**. As a workaround, you can annotate the interface with **@XmlSeeAlso** to correct the issue.
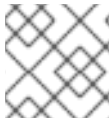
> **NOTE**
>
> This will not work with manual, hand-coded JAXB.

```
@XmlSeeAlso(RealFoo.class)
public interface IFoo {}
```

The extra **@XmlSeeAlso** on **IFoo** allows RESTEasy to create a **JAXBContext** that knows how to unmarshal **RealFoo** instances.

CHAPTER 18. RESTEASY ATOM SUPPORT

Atom is an XML-based document format that compiles lists of related information, known as *feeds*. Feeds are composed of a number of items, known as *entries*, each of which includes an extensible set of metadata (a title, for example).

Atom is primarily used to syndicate web content (such as weblogs and news headlines) to websites, and directly to user agents.

Atom is the RSS feed of the next generation. Although used primarily to syndicate weblogs and news, the format is starting to be used as the envelope for Web services such as distributed notifications and job queues, or simply to send or receive data in bulk to or from a service.

18.1. RESTEASY ATOM API AND PROVIDER

RESTEasy has defined a simple object model to represent Atom in Java, and uses JAXB to marshal and unmarshal it. The **org.jboss.resteasy.plugins.providers.atom** package contains the main classes: **Feed**, **Entry**, **Content**, and **Link**. Each class is annotated with JAXB annotations. The distribution also contains the JavaDocs for this project, which are very useful in learning the model. The following code is a simple example of sending an Atom feed with the RESTEasy API:

```
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("Bill Burke"));
      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;
   }
}
```

RESTEasy's Atom provider is JAXB-based, so you are not limited to sending Atom objects with XML. You can automatically re-use RESTEasy's other JAXB providers (JSON and FastinfoSet). All you need to do is add **+atom** in front of the main subtype (that is, **@Produces("application/atom+json")** or **@Consumes("application/atom+fastinfoset")**).

18.2. USING JAXB WITH THE ATOM PROVIDER

The **org.jboss.resteasy.plugins.providers.atom.Content** class lets you marshal and unmarshal JAXB-annotated objects that form the body of an entry's content. The following code is an example of sending an **Entry** with a **Customer** object attached as the body of the entry's content.

```
@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 tells the content object that you are returning a Java JAXB object to be marshaled. If you use a base format other than XML (for example, **application/atom+json**), the attached JAXB object will be marshaled into that format.

If your input is an Atom document, you can also extract JAXB objects from **Content** by using **Content.getJAXBObject(Class class)**. The code that follows is an example of extracting a **Customer** object from the **Content**:

```
@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);
    }
}
```

18.3. ATOM SUPPORT THROUGH APACHE ABDERA

RESTEasy supports Apache Abdera, an implementation of the Atom protocol and data format. You can find Abdera at the Apache web site.

Abdera is a fully-fledged Atom server, but RESTEasy only supports integration with JAX-RS for marshaling and unmarshaling the Atom data format to and from the **Feed** and **Entry** interface types in Abdera.

18.3.1. Abdera and Maven

The Abdera provider is not included with the RESTEasy distribution. To include the Abdera provider in your WAR archive's **pom** files, add the following. Remember to change the version in the code to the version of RESTEasy that you are working with.

> ⚠️ **WARNING**
>
> RESTEasy may not pick up the latest version of Abdera.

```
<repository>
    <id>jboss</id>
    <url>http://repository.jboss.org/maven2</url>
</repository>

    ...
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>abdera-atom-provider</artifactId>
    <version>...version...</version>
</dependency>
```

18.3.2. Using the Abdera Provider

```java
import org.apache.abdera.Abdera;
import org.apache.abdera.factory.Factory;
import org.apache.abdera.model.Entry;
import org.apache.abdera.model.Feed;
import org.apache.commons.httpclient.HttpClient;
import org.apache.commons.httpclient.methods.GetMethod;
import org.apache.commons.httpclient.methods.PutMethod;
import org.apache.commons.httpclient.methods.StringRequestEntity;
import org.jboss.resteasy.plugins.providers.atom.AbderaEntryProvider;
import org.jboss.resteasy.plugins.providers.atom.AbderaFeedProvider;
import org.jboss.resteasy.test.BaseResourceTest;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.UriInfo;
import javax.xml.bind.JAXBContext;
import java.io.StringReader;
import java.io.StringWriter;
import java.util.Date;

/**
 * @author <a href="mailto:bill@burkecentral.com">Bill Burke</a>
 * @version $Revision: 1 $
 */
public class AbderaTest extends BaseResourceTest
{

   @Path("atom")
   public static class MyResource
   {
      private static final Abdera abdera = new Abdera();

      @GET
      @Path("feed")
      @Produces(MediaType.APPLICATION_ATOM_XML)
      public Feed getFeed(@Context UriInfo uri) throws Exception
      {
         Factory factory = abdera.getFactory();
         Assert.assertNotNull(factory);
         Feed feed = abdera.getFactory().newFeed();
         feed.setId("tag:example.org,2007:/foo");
         feed.setTitle("Test Feed");
         feed.setSubtitle("Feed subtitle");
         feed.setUpdated(new Date());
         feed.addAuthor("James Snell");
         feed.addLink("http://example.com");
```

```
        Entry entry = feed.addEntry();
        entry.setId("tag:example.org,2007:/foo/entries/1");
        entry.setTitle("Entry title");
        entry.setUpdated(new Date());
        entry.setPublished(new Date());
        entry.addLink(uri.getRequestUri().toString());

        Customer cust = new Customer("bill");

        JAXBContext ctx = JAXBContext.newInstance(Customer.class);
        StringWriter writer = new StringWriter();
        ctx.createmarshaler().marshal(cust, writer);
        entry.setContent(writer.toString(), "application/xml");
        return feed;

    }

    @PUT
    @Path("feed")
    @Consumes(MediaType.APPLICATION_ATOM_XML)
    public void putFeed(Feed feed) throws Exception
    {
        String content = feed.getEntries().get(0).getContent();
        JAXBContext ctx = JAXBContext.newInstance(Customer.class);
        Customer cust = (Customer) ctx.createUnmarshaler().unmarshal(new
StringReader(content));
        Assert.assertEquals("bill", cust.getName());

    }

    @GET
    @Path("entry")
    @Produces(MediaType.APPLICATION_ATOM_XML)
    public Entry getEntry(@Context UriInfo uri) throws Exception
    {
        Entry entry = abdera.getFactory().newEntry();
        entry.setId("tag:example.org,2007:/foo/entries/1");
        entry.setTitle("Entry title");
        entry.setUpdated(new Date());
        entry.setPublished(new Date());
        entry.addLink(uri.getRequestUri().toString());

        Customer cust = new Customer("bill");

        JAXBContext ctx = JAXBContext.newInstance(Customer.class);
        StringWriter writer = new StringWriter();
        ctx.createmarshaler().marshal(cust, writer);
        entry.setContent(writer.toString(), "application/xml");
        return entry;

    }

    @PUT
    @Path("entry")
```

```
    @Consumes(MediaType.APPLICATION_ATOM_XML)
    public void putFeed(Entry entry) throws Exception
    {
        String content = entry.getContent();
        JAXBContext ctx = JAXBContext.newInstance(Customer.class);
        Customer cust = (Customer) ctx.createUnmarshaler().unmarshal(new
StringReader(content));
        Assert.assertEquals("bill", cust.getName());


    }
}

    @Before
    public void setUp() throws Exception
    {

dispatcher.getProviderFactory().registerProvider(AbderaFeedProvider.class)
;

dispatcher.getProviderFactory().registerProvider(AbderaEntryProvider.class
);
        dispatcher.getRegistry().addPerRequestResource(MyResource.class);
    }

    @Test
    public void testAbderaFeed() throws Exception
    {
        HttpClient client = new HttpClient();
        GetMethod method = new GetMethod("http://localhost:8081/atom/feed");
        int status = client.executeMethod(method);
        Assert.assertEquals(200, status);
        String str = method.getResponseBodyAsString();

        PutMethod put = new PutMethod("http://localhost:8081/atom/feed");
        put.setRequestEntity(new StringRequestEntity(str,
MediaType.APPLICATION_ATOM_XML, null));
        status = client.executeMethod(put);
        Assert.assertEquals(200, status);

    }

    @Test
    public void testAbderaEntry() throws Exception
    {
        HttpClient client = new HttpClient();
        GetMethod method = new
GetMethod("http://localhost:8081/atom/entry");
        int status = client.executeMethod(method);
        Assert.assertEquals(200, status);
        String str = method.getResponseBodyAsString();

        PutMethod put = new PutMethod("http://localhost:8081/atom/entry");
        put.setRequestEntity(new StringRequestEntity(str,
MediaType.APPLICATION_ATOM_XML, null));
        status = client.executeMethod(put);
```

```
        Assert.assertEquals(200, status);
    }
}
```

CHAPTER 19. JSON SUPPORT VIA JACKSON

Apart from the Jettison JAXB adapter for JSON, RESTEasy also supports integration with the Jackson project. Many users find Jackson's output format more intuitive than the format provided by either BadgerFish or Jettison.

Jackson is available from http://jackson.codehaus.org. It lets you easily marshal Java objects to and from JSON. Jackson has a JavaBean-based model and JAXB-like APIs. RESTEasy integrates with the JavaBean model as described in the Jackson Tutorial.

To include Jackson in your project, add the following Maven dependency to your build:

```
<repository>
    <id>jboss</id>
    <url>http://repository.jboss.org/maven2</url>
</repository>

    ...
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jackson-provider</artifactId>
    <version>1.1.GA</version>
</dependency>
```

RESTEasy expands the JAX-RS integration built into Jackson in several ways. The first expansion provided support for **application/*+json**. Previously, Jackson accepted only **application/json** and **text/json** as valid media types. **application/*+json** support lets you marshal your JSON-based media types with Jackson. For example:

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

    @GET
    @Produces("application/vnd.customer+json")
    public Customer[] getCustomers() {}
}
```

Using RESTEasy JAXB providers alongside Jackson is also problematic. Rather than use Jackson to output your JSON, you can use Jettison and JAXB. To do so, you must either not install the Jackson provider, or use the **@org.jboss.resteasy.annotations.providers.NoJackson** annotation on your JAXB annotated classes, like so:

```
@XmlRootElement
@NoJackson
public class Customer {...}

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

    @GET
    @Produces("application/vnd.customer+json")
    public Customer[] getCustomers() {}
}
```

If you cannot annotate the JAXB class with **@NoJackson**, then you can annotate a method parameter

instead:

```
@XmlRootElement
public class Customer {...}

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

    @GET
    @Produces("application/vnd.customer+json")
    @NoJackson
    public Customer[] getCustomers() {}

    @POST
    @Consumes("application/vnd.customer+json")
    public void createCustomer(@NoJackson Customer[] customers)
{...}
}
```

19.1. POSSIBLE CONFLICT WITH JAXB PROVIDER

If your Jackson classes are annotated with JAXB annotations and the **resteasy-jaxb-provider** is on your classpath, you can trigger the Jettison JAXB marshalling code. To disable the JAXB JSON Marshaller, annotate your classes with **@org.jboss.resteasy.annotations.providers.jaxb.IgnoreMediaTypes("application/ *+json")**.

CHAPTER 20. MULTIPART PROVIDERS

RESTEasy has rich support for the **multipart/*** and **multipart/form-data** MIME (Multipurpose Internet Mail Extension) types. The **multipart** MIME format passes lists of content bodies. Multiple content bodies are embedded in the one message. **multipart/form-data** is often found in web application HTML Form documents, and is generally used to upload files. The **form-data** format works like other **multipart** formats, except that each inlined piece of content has a name associated with it.

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

20.1. INPUT WITH MULTIPART/MIXED

When you write a JAX-RS service, RESTEasy provides an interface to let you read any **multipart** MIME type: **org.jboss.resteasy.plugins.providers.multipart.MultipartInput**.

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

public interface MultipartInput
{
    List<InputPart> getParts();

    String getPreamble();
}

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();
}
```

MultipartInput is a simple interface that lets you access each part of the **multipart** message. Each part is represented by an **InputPart** interface, and is associated with a set of headers. You can unmarshal a part by calling one of the **getBody()** methods. The **Type genericType** parameter can be null, but the **Class type** parameter must be set. RESTEasy locates a **MessageBodyReader** based on the media type of the part, and the type information you pass in. The following piece of code unmarshals XML parts into a JAXB annotated class called **Customer**.

```
    @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);
        }
    }
}
```

If you want to unmarshal a body part that is sensitive to generic type metadata, you can use the **org.jboss.resteasy.util.GenericType** class, like so:

```
    @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<<() {});
            }
        }
    }
```

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

20.2. JAVA.UTIL.LIST WITH MULTIPART DATA

If the body parts are uniform, you can provide a **java.util.List** as your input parameter and avoid unmarshaling each part manually. As you can see in the example code below, this must include the type being unmarshaled with the generic parameter of the List type declaration.

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

20.3. INPUT WITH MULTIPART/FORM-DATA

When you write a JAX-RS service, RESTEasy provides an interface that lets you read the **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 like other

multipart formats except that each inlined piece of content is associated with a name. The interface for **form-data** input is
org.jboss.resteasy.plugins.providers.multipart.MultipartFormDataInput.

```
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;
}
```

This works similarly to **MultipartInput**, as described earlier in this chapter.

20.4. JAVA.UTIL.MAP WITH MULTIPART/FORM-DATA

With **form-data**, if the body parts are uniform, you can provide a **java.util.Map** as your input parameter and avoid unmarshaling each part manually. As you can see in the example code below, this must include the type being unmarshaled with the generic parameter of the List type declaration.

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

20.5. INPUT WITH MULTIPART/RELATED

When you write a JAX-RS service, RESTEasy provides an interface that lets you read the **multipart/related** MIME type. **multipart/related** indicates that message parts should be considered as parts of a whole, and not individually. You can use **multipart/related** to perform tasks like sending a web page complete with images in a single message.

Every **multipart/related** message has a **root/start** part that references other parts of the message. Parts are identified by their **Content-ID** headers. **multipart/related** is defined by RFC 2387. The interface for **related** input is
org.jboss.resteasy.plugins.providers.multipart.MultipartRelatedInput.

```
public interface MultipartRelatedInput extends MultipartInput
{
    String getType();
```

```
    String getStart();

    String getStartInfo();

    InputPart getRootPart();

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

It works similarly to `MultipartInput`, as described earlier in this chapter.

20.6. OUTPUT WITH MULTIPART

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, create a `MultipartOutput` object and call `addPart()` methods. RESTEasy automatically finds a `MessageBodyWriter` to marshal your entity objects. As with `MultipartInput`, your marshaling may be sensitive to generic type metadata. In this case, use `GenericType`. The following example returns a `multipart/mixed` format to the calling client. The parts are JAXB-annotated `Customer` objects that will marshal into `application/xml`.

```
    @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;
        }
```

20.7. MULTIPART OUTPUT WITH JAVA.UTIL.LIST

If the body parts are uniform, you can provide a `java.util.Map` as your input parameter and avoid unmarshaling each part manually or using a `MultipartOutput` object. As you can see in the example code below, this must include the type being unmarshaled with the generic parameter of the List type declaration. You must also annotate the method with @**PartType** to specify each part's media type. The following example returns a customer list to a client, where each customer is a JAXB object:

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

20.8. 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**, create a **MultipartFormDataOutput** object and call **addFormData()** methods. RESTEasy automatically locates a **MessageBodyWriter** to marshal your entity objects. As with **MultipartInput**, your marshaling may be sensitive to generic type metadata. In this case, use **GenericType**. The example below returns a **multipart/form-data** format to a calling client. The parts are JAXB-annotated **Customer** objects, which will be marshaled into **application/xml**.

```
    @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;
        }
```

20.9. MULTIPART FORMDATA OUTPUT WITH JAVA.UTIL.MAP

If the body parts are uniform, you can provide a **java.util.Map** as your input parameter and avoid unmarshaling each part manually or using a **MultipartFormDataOutput** object. As you can see in the example code below, this must include the type being unmarshaled with the generic parameter of the List type declaration. You must also annotate the method with @**PartType** to specify each part's media type. This example returns a customer list to a client, where each customer is a JAXB object.

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

20.10. 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**, create a **MultipartRelatedOutput** object and call **addPart()** methods. The first added part is used as the root part of the **multipart/related** message. RESTEasy automatically locates a **MessageBodyWriter** to marshal your entity objects. As with **MultipartInput**, your marshaling may be sensitive to generic type metadata. In this case, use **GenericType**. The example below returns a **multipart/related** format to the calling client — a HTML file with 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;
   }
}
```

20.11. @MULTIPARTFORM AND POJOS

If you are familiar with your **multipart/form-data** packets, you can map them to and from a POJO class with the @**org.jboss.resteasy.annotations.providers.multipart.MultipartForm** annotation and the @**FormParam** JAX-RS annotation. To do so, define a POJO with a default constructor (at least), and annotate its fields, properties, or both, with @**FormParam**s. These @**FormParam**s must also be annotated with @**org.jboss.resteasy.annotations.providers.multipart.PartType** to be output. For example:

```
public class CustomerProblemForm {
    @FormData("customer")
    @PartType("application/xml")
    private Customer customer;

    @FormData("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; }
}
```

Once you have defined your POJO class, you can use it to represent **multipart/form-data**. The following code sends a **CustomerProblemForm** using the RESTEasy client framework:

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

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

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

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

Note that the @**MultipartForm** annotation tells RESTEasy that the object has @**FormParam**, and that it should be marshaled from that parameter. You can use the same object to receive **multipart** data. Here is an example of the server-side counterpart to the customer portal.

```
@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...
        }
     }
```

20.12. XML-BINARY OPTIMIZED PACKAGING (XOP)

RESTEasy supports packaging XOP (XML-binary Optimized Packaging) messages as
multipart/related. This means that if you have a JAXB-annotated POJO that also holds some
binary content, you can send it without needing to encode the binary in any other way. This results in
faster transport while retaining the convenience of the POJO. (You can read more about XOP at the
W3C web page.)

Take the following JAXB-annotated POJO as an example. **@XmlMimeType** tells JAXB the MIME type of
the binary content. (This is not required, but it is recommended.)

```
@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 ...
}
```

Here, **myBinary** and **myDataHandler** are processed as binary attachments, while the XOP object will
be sent as XML. **javax.activation.DataHandler** is the most common supported type, so if you
need a **java.io.InputStream** or a **javax.activation.DataSource**, you must use the
DataHandler. java.awt.Image and **javax.xml.transform.Source** are also supported. We
assume, in the previous example, that **Customer** is a JAXB-friendly POJO. The following is an example
Java client to send the previous example:

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

@Consumes(MediaType.MULTIPART_RELATED) tells RESTEasy that we want to send
multipart/related packages, a format that will hold our XOP message.
@XopWithMultipartRelated tells RESTEasy that we want to create XOP messages. Now that we
have a POJO and a client service capable of sending it, we need a server capable of reading it:

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

@Consumes(MediaType.MULTIPART_RELATED) tells RESTEasy that we want to read
multipart/related packages. **@XopWithMultipartRelated** tells RESTEasy that we want to read
XOP messages. We could also produce XOP return values by annotating them with **@Produce**.

CHAPTER 21. YAML PROVIDER

Since Beta 6, RESTEasy includes built-in support for YAML with the JYAML library. To enable YAML support, add the `jyaml-1.3.jar` to RESTEasy's classpath.

The JYAML **JAR** can be downloaded from SourceForge.

If you use Maven, the JYAML **JAR** is available through the main repositories, and included with the following dependency:

```
 <dependency>
<groupId>org.jyaml</groupId>
<artifactId>jyaml</artifactId>
<version>1.3</version>
 </dependency>
```

When starting up RESTEasy, watch the logs for a line stating that the **YamlProvider** has been added — this indicates that RESTEasy has located the JYAML **JAR**:

```
2877 Main INFO org.jboss.resteasy.plugins.providers.RegisterBuiltin -
Adding YamlProvider
```

The YAML provider recognizes three MIME types:

- **text/x-yaml**

- **text/yaml**

- **application/x-yaml**

You can use YAML in a resource method like so:

```
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();
}
...
 }
```

CHAPTER 22. STRING MARSHALLING FOR STRING BASED @*PARAM

@PathParam, **@QueryParam**, **@MatrixParam**, **@FormParam**, and **@HeaderParam** are represented as Strings in a raw HTTP request. These injected parameter types can be converted into objects if they have a **valueOf(String)** static method, or a constructor that takes one String parameter. If you have a class with **valueOf()**, or the String constructor is inappropriate for your HTTP request, you can plug in RESTEasy's proprietary @**Provider**:

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

public interface StringConverter<T>
{
   T fromString(String str);
   String toString(T value);
}
```

This interface lets you use your own customized String marshalling. It is registered in **web.xml** under the **resteasy.providers context-param**. (See the Installation and Configuration chapter for details.) You can register it manually by calling the **ResteasyProviderFactory.addStringConverter()** method. A simple example of using the **StringConverter** follows:

```
   import org.jboss.resteasy.client.ProxyFactory;
   import org.jboss.resteasy.spi.StringConverter;
   import org.jboss.resteasy.test.BaseResourceTest;
   import org.junit.Assert;
   import org.junit.Before;
   import org.junit.Test;

   import javax.ws.rs.HeaderParam;
   import javax.ws.rs.MatrixParam;
   import javax.ws.rs.PUT;
   import javax.ws.rs.Path;
   import javax.ws.rs.PathParam;
   import javax.ws.rs.QueryParam;
   import javax.ws.rs.ext.Provider;

   public class StringConverterTest extends BaseResourceTest
   {
      public static class POJO
      {
         private String name;

         public String getName()
         {
            return name;
         }

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

```
    @Provider
    public static class POJOConverter implements StringConverter<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();
       }
    }

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

    @Before
    public void setUp() throws Exception
    {

dispatcher.getProviderFactory().addStringConverter(POJOConverter.class);
       dispatcher.getRegistry().addPerRequestResource(MyResource.class);
    }

    @Path("/")
    public static interface MyClient
    {
       @Path("{pojo}")
       @PUT
       void put(@QueryParam("pojo")POJO q, @PathParam("pojo")POJO pp,
                @MatrixParam("pojo")POJO mp, @HeaderParam("pojo")POJO
hp);
    }

    @Test
    public void testIt() throws Exception
    {
```

```
        MyClient client = ProxyFactory.create(MyClient.class,
"http://localhost:8081");
        POJO pojo = new POJO();
        pojo.setName("pojo");
        client.put(pojo, pojo, pojo, pojo);
    }
  }
```

CHAPTER 23. RESPONSES USING JAVAX.WS.RS.CORE.RESPONSE

You can build custom responses by using the `javax.ws.rs.core.Response` and `ResponseBuilder` classes. To perform your own streaming, your entity response must be an implementation of `javax.ws.rs.core.StreamingOutput`. See the Java Documentation for further information.

CHAPTER 24. EXCEPTION HANDLING

24.1. EXCEPTION MAPPERS

ExceptionMappers are custom, application-provided components that can catch application exceptions and write specific HTTP responses. They are associated with @**Provider**, and implement the following interface:

```
package javax.ws.rs.ext;

import javax.ws.rs.core.Response;

/**
* Contract for a provider that maps Java exceptions to
* {@link javax.ws.rs.core.Response}. An implementation of this
interface must
* be annotated with {@link Provider}.
*
* @see Provider
* @see javax.ws.rs.core.Response
*/
public interface ExceptionMapper<E>
{

/**
* Map an exception to a {@link javax.ws.rs.core.Response}.
*
* @param exception the exception to map to a response
* @return a response mapped from the supplied exception
*/
Response toResponse(E exception);
}
```

When an application throws an exception, the exception is caught by the JAX-RS runtime. JAX-RS then scans registered **ExceptionMapper**s to locate one which supports marshalling the exception type thrown. An example **ExceptionMapper** follows:

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

Response toResponse(EJBException exception) {
return Response.status(500).build();
}

}
```

ExceptionMappers are registered in the same way as **MessageBodyReader**s and **MessageBodyWriter**s: by scanning through the RESTEasy provider **context-param** (if you are deploying in a **WAR** file), or programmatically through the **ResteasyProviderFactory** class.

24.2. RESTEASY BUILT-IN INTERNALLY-THROWN EXCEPTIONS

RESTEasy has a set of built-in exceptions that are thrown when an error is encountered during dispatching or marshalling. Each exception matches a specific HTTP error code. The full list is available in the RESTEasy Java Documentation, under the `org.jboss.resteasy.spi` package. The following table lists the most common exceptions:

Table 24.1. Common Exceptions

Exception	HTTP Code	Description
`BadRequestException`	400	Bad Request. Request was not formatted correctly or there was a problem processing request input.
`UnauthorizedException`	401	Unauthorized. Security exception thrown if you use RESTEasy's simple annotation- and role-based security.
`InternalServerError Exception`	500	Internal Server Error.
`MethodNotAllowed Exception`	405	Method Not Allowed. There is no JAX-RS method for the resource that can handle the invoked HTTP operation.
`NotAcceptableException`	406	Not Acceptable. There is no JAX-RS method that can produce the media types listed in the Accept header.
`NotFoundException`	404	Not Found. There is no JAX-RS method that serves the request path/resource.
`Failure`	N/A	Internal RESTEasy. Not logged.
`LoggableFailure`	N/A	Internal RESTEasy error. Logged.
`DefaultOptionsMethod Exception`	N/A	If the user invokes HTTP OPTIONS without a JAX-RS method, RESTEasy provides a default behavior by throwing this exception.

24.3. OVERRIDING RESTEASY BUILTIN EXCEPTIONS

You can override RESTEasy built-in exceptions by writing an **ExceptionMapper** for the exception. You can also write an **ExceptionMapper** for any exception thrown, including **WebApplicationException**.

CHAPTER 25. CONFIGURING INDIVIDUAL JAX-RS RESOURCE BEANS

When you scan your path for JAX-RS annotated resource beans, your beans are registered in *per-request mode*. This means that an instance will be created for every HTTP request served. You will usually require information from your environment. If you run a **WAR** in a Servlet container with Beta 2 or lower, you can only use JNDI lookups to obtain references to Java EE resources and configuration information. In this case, define your EE configuration (that is, `ejb-ref`, `env-entry`, `persistence-context-ref`, etc.) in the `web.xml` of your RESTEasy **WAR** file. Then, within your code, perform JNDI lookups in the `java:comp` namespace. For example:

web.xml

```
<ejb-ref>
  <ejb-ref-name>ejb/foo</ejb-ref-name>
  ...
</ejb-ref>
```

Resource Code:

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

    public Object getSomethingFromJndi() {
        new InitialContext.lookup("java:comp/ejb/foo");
    }
...
}
```

You can also configure and register your beans manually through the Registry. In a **WAR**-based deployment, you must write a specific **ServletContextListener** to do this. The listener lets you obtain a reference to the Registry, like so:

```
public class MyManualConfig implements ServletContextListener
{
    public void contextInitialized(ServletContextEvent event)
    {

        Registry registry = (Registry)
event.getServletContext().getAttribute(Registry.class.getName());

    }
...
}
```

We recommend investigating Spring Integration and the Embedded Container's Spring Integration for a full understanding of this process.

CHAPTER 26. GZIP COMPRESSION/DECOMPRESSION

RESTEasy has automatic GZIP decompression support. If the client framework or a JAX-RS service receives a message body with a `Content-Encoding` of `gzip`, the message will be automatically decompressed. The client framework also sets the `Accept-Encoding` header to `gzip, deflate` automatically.

RESTEasy also supports automatic compression. If a request or response with a `Content-Encoding` header of `gzip` is sent or received, RESTEasy performs the compression. You can use the `@org.jboss.resteasy.annotation.GZIP` annotation if you do not want to set each `Content-Encoding` manually.

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

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

Here, the **order** message body is tagged for GZIP compression. You can use the same annotation to tag server responses:

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

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

CHAPTER 27. RESTEASY CACHING FEATURES

RESTEasy provides a number of annotations to support HTTP caching semantics, to simplify processes such as setting Cache-Control headers, and to make both server-side and client-side in-memory caches available.

27.1. @CACHE AND @NOCACHE ANNOTATIONS

RESTEasy provides an extension to JAX-RS that lets you set **Cache-Control** headers on successful GET requests. It can only be used on methods annotated with **@GET**. Successful get requests return a response of **200 OK**.

```
package org.jboss.resteasy.annotations.cache;

public @interface Cache
{
   int maxAge() default -1;
   int sMaxAge() default -1;
   boolean noStore() default false;
   boolean noTransform() default false;
   boolean mustRevalidate() default false;
   boolean proxyRevalidate() default false;
   boolean isPrivate() default false;
}

public @interface NoCache
{
   String[] fields() default {};
}
```

@Cache builds a complex **Cache-Control** header; **@NoCache** specifies that you do not want anything to be cached. (That is, **Cache-Control: nocache**.)

You can place these annotations on the resource class or interface, or place them on each individual **@GET** resource method. They specify a default cache value for each **@GET** resource method.

27.2. CLIENT "BROWSER" CACHE

RESTEasy can create a client-side, browser-like cache for use with the Client Proxy Framework or with raw **ClientRequests**. This cache locates **Cache-Control** headers that are returned with a server response. If the **Cache-Control** headers specify that the client may cache the response, RESTEasy caches it within local memory. This cache obeys **max-age** requirements, and automatically performs HTTP 1.1 cache revalidation if either or both of the **Last-Modified** or **ETag** headers are returned with the original response. (See the HTTP 1.1 specification for details about **Cache-Control** or cache revalidation.)

Enabling RESTEasy caching is simple. The following shows the client cache being used with the Client Proxy Framework:

```
@Path("/orders")
public interface OrderServiceClient {
```

```
    @Path("{id}")
    @GET
    @Produces("application/xml")
    public Order getOrder(@PathParam("id") String id);
}
```

You can create a proxy for this interface and enable caching for that proxy like so:

```
import org.jboss.resteasy.client.ProxyFactory;
import org.jboss.resteasy.client.cache.CacheFactory;
import org.jboss.resteasy.client.cache.LightweightBrowserCache;

public static void main(String[] args) throws Exception
{
     RegisterBuiltin.register(ResteasyProviderFactory.getInstance());
     OrderServiceClient proxy =
ProxyFactory.create(OrderServiceClient.class, generateBaseUrl());

     // This line enables caching
     LightweightBrowserCache cache = CacheFactory.makeCacheable(proxy);
}
```

If you are using the **ClientRequest** class instead of the proxy server to perform invocations, you can enable the cache like so:

```
import org.jboss.resteasy.client.ProxyFactory;
import org.jboss.resteasy.client.cache.CacheFactory;
import org.jboss.resteasy.client.cache.LightweightBrowserCache;

public static void main(String[] args) throws Exception
{
     RegisterBuiltin.register(ResteasyProviderFactory.getInstance());

     // This line enables caching
     LightweightBrowserCache cache = new LightweightBrowserCache();

     ClientRequest request = new
ClientRequest("http://example.com/orders/333");
     CacheFactory.makeCacheable(request, cache);
}
```

By default, the **LightweightBrowserCache** has a maximum caching space of two megabytes. You can change this programmatically by calling the **setMaxBytes()** method. *If the cache becomes full, all cached data will be deleted automatically.* For more complex caching solutions, or support for third-party cache options, contact the resteasy-development list and discuss your ideas with the community.

27.3. LOCAL SERVER-SIDE RESPONSE CACHE

RESTEasy has a local, server-side, in-memory cache for your JAX-RS services. It automatically caches marshaled responses from HTTP GET JAX-RS invocations if your JAX-RS resource method sets a **Cache-Control** header. When a GET is received, the RESTEasy Server Cache checks whether the URI is stored in the cache. If true, the marshaled response is returned without invoking your JAX-RS method. Each cache entry has a *maximum age* for which the specifications in the **Cache-Control** header of the initial request are valid. The cache also automatically generates an **ETag** using an MD5

hash on the response body. This lets the client perform HTTP 1.1 cache revalidation with the **IF-NONE-MATCH** header. The cache will also perform revalidation if there is no initial cache hit, but the JAX-RS method returns a body with the same **ETag**.

To set up the server-side cache with Maven, you must use the **resteasy-cache-core** artifact:

```
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-cache-core</artifactId>
    <version>1.1.GA</version>
</dependency>
```

Next, add a **ServletContextListener**:
org.jboss.resteasy.plugins.cache.server.ServletServerCache. You must specify this after the **ResteasyBootstrap** listener in your **web.xml** file.

```
<web-app>
    <listener>
        <listener-class>
            org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap
        </listener-class>
    </listener>

    <context-param>
        <param-name>resteasy.server.cache.maxsize</param-name>
        <param-value>1000</param-value>
    </context-param>

    <context-param>
        <param-name>resteasy.server.cache.eviction.wakeup.interval</param-name>
        <param-value>5000</param-value>
    </context-param>

    <listener>
        <listener-class>
            org.jboss.resteasy.plugins.cache.server.ServletServerCache
        </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>/rest-services/*</url-pattern>
    </servlet-mapping>

</web-app>
```

The cache implementation is based on the JBoss Cache project. You can set two `context-param` configuration variables: `resteasy.server.cache.maxsize` sets the number of elements that can be cached, and `resteasy.server.cache.eviction.wakeup.interval` sets the rate at which the background eviction thread runs to purge the cache of stale entries.

CHAPTER 28. INTERCEPTORS

RESTEasy can intercept JAX-RS invocations and route them through listener-like objects called *interceptors*. There are four interception points on the server side:

- wrapping around **MessageBodyWriter** invocations

- wrapping around **MessageBodyReader** invocations

- through *pre-processors*, which intercept the incoming request before unmarshaling occurs

- through *post-processors*, which are invoked immediately after the JAX-RS method finishes

You can also intercept **MessageBodyReader**, **MessageBodyWriter**, and the remote invocation to the server on the client side.

28.1. MESSAGEBODYREADER/WRITER INTERCEPTORS

MessageBodyReader and **MessageBodyWriter** interceptors wrap around the invocation of **MessageBodyReader.readFrom()** or **MessageBodyWriter.writeTo()**. They are used to wrap the **Output** or **InputStream**. For example, RESTEasy GZIP support contains interceptors that create and override the default **Output** and **InputStream** with a **GzipOutputStream** or **GzipInputStream** so that GZIP encoding can work. You can also use interceptors to append headers to the response (or, on the client side, the outgoing request).

To use an interceptor, implement the **org.jbos.resteasy.spi.interception.MessageBodyReaderInterceptor** or **MessageBodyWriterInterceptor**.

```
public interface MessageBodyReaderInterceptor
{
   Object read(MessageBodyReaderContext context) throws IOException,
WebApplicationException;

}

public interface MessageBodyWriterInterceptor
{
   void write(MessageBodyWriterContext context) throws IOException,
WebApplicationException;

}
```

Interceptors are driven by the **MessageBodyWriterContext** or **MessageBodyReaderContext**. They are invoked together in a Java call stack. You must call **MessageBodyReaderContext.proceed()** or **MessageBodyWriterContext.proceed()** to add subsequent interceptors. When there are no more interceptors to invoke, call the **readFrom()** or **writeTo()** method of the **MessageBodyReader** or **MessageBodyWriter**. This wrapping lets you modify objects before they reach the Reader or Writer, and clean up when **proceed()** returns. The **Context** objects also possess methods that modify the parameters sent to the Reader or Writer.

```
public interface MessageBodyReaderContext
{
   Class getType();
```

```java
    void setType(Class type);

    Type getGenericType();

    void setGenericType(Type genericType);

    Annotation[] getAnnotations();

    void setAnnotations(Annotation[] annotations);

    MediaType getMediaType();

    void setMediaType(MediaType mediaType);

    MultivaluedMap<String, String> getHeaders();

    InputStream getInputStream();

    void setInputStream(InputStream is);

    Object proceed() throws IOException, WebApplicationException;
}
public interface MessageBodyWriterContext
{
    Object getEntity();

    void setEntity(Object entity);

    Class getType();

    void setType(Class type);

    Type getGenericType();

    void setGenericType(Type genericType);

    Annotation[] getAnnotations();

    void setAnnotations(Annotation[] annotations);

    MediaType getMediaType();

    void setMediaType(MediaType mediaType);

    MultivaluedMap<String, Object> getHeaders();

    OutputStream getOutputStream();

    public void setOutputStream(OutputStream os);

    void proceed() throws IOException, WebApplicationException;
}
```

MessageBodyReaderInterceptors and **MessageBodyWriterInterceptors** can be use on the

server or the client side. They must be annotated with
@org.jboss.resteasy.annotations.interception.ServerInterceptor or
@org.jboss.resteasy.annotations.interception.ClientInterceptor so that RESTEasy
adds them to the correct interceptor list. If your interceptor classes are not annotated with one or both
of these annotations, a deployment error will occur. Interceptors should also be annotated with
@Provider, like so:

```
@Provider
@ServerInterceptor
public class MyHeaderDecorator implements MessageBodyWriterInterceptor {

    public void write(MessageBodyWriterContext context) throws
IOException, WebApplicationException
    {
       context.getHeaders().add("My-Header", "custom");
       context.proceed();
    }
}
```

This is a server-side interceptor that adds a header value to the response. It is annotated with
@Provider and **@ServerInterceptor**. It must modify the header before calling
context.proceed(), because the response may be committed after the **MessageBodyReader** runs.

> **IMPORTANT**
>
> You must call **context.proceed()**, or your invocation will not occur.

28.2. PREPROCESSINTERCEPTOR

The **org.jboss.resteasy.spi.interception.PreProcessInterceptor** runs after a JAX-RS
resource method is located, but before the method is invoked. They can only be used on the server, but
you must still annotate them with **@ServerInterceptor**. They can be used to implement security
features or to preempt the Java request. The RESTEasy security implementation uses these
interceptors to abort requests prior to invocation if the user does not pass authorization. The
RESTEasy caching framework uses them to return cached responses, to avoid invoking methods
multiple times. The interceptor interface is as follows:

```
public interface PreProcessInterceptor
{
    ServerResponse preProcess(HttpRequest request, ResourceMethod
method) throws Failure, WebApplicationException;
}
```

PreProcessInterceptors run in sequence and do not wrap the actual JAX-RS invocation. To
illustrate:

```
for (PreProcessInterceptor interceptor : preProcessInterceptors) {
   ServerResponse response = interceptor.preProcess(request, method);
   if (response != null) return response;
}
executeJaxrsMethod(...);
```

If the **preProcess()** method returns a **ServerResponse** then the underlying JAX-RS method will not be invoked and the runtime will process the response and return to the client.

28.3. POSTPROCESSINTERCEPTORS

The **org.jboss.resteasy.spi.interception.PostProcessInterceptor** runs after the JAX-RS method is invoked but before **MessageBodyWriter**s are invoked. They can only be used on the server side, and exist to provide symmetry with **PreProcessInterceptor**. They are used to set response headers when there is a possibility that no **MessageBodyWriter** will be invoked. They do not wrap any object, and are invoked in order, like **PreProcessInterceptor**s.

```
public interface PostProcessInterceptor
{
    void postProcess(ServerResponse response);
}
```

28.4. CLIENTEXECUTIONINTERCEPTORS

org.jboss.resteasy.spi.interception.ClientExecutionInterceptor classes are client-side only. They run after the **MessageBodyWriter**, and after the **ClientRequest** has been built on the client side. They wrap the HTTP invocation that is sent to the server. In RESTEasy GZIP support, they set the **Accept** header to contain **gzip, deflate** before the request is sent. In the RESTEasy client cache, they check that the cache contains a resource before attempting to act on a resource. These interceptors must be annotated with both @**ClientInterceptor** and @**Provider**.

```
public interface ClientExecutionInterceptor
{
    ClientResponse execute(ClientExecutionContext ctx) throws
Exception;
}

public interface ClientExecutionContext
{
    ClientRequest getRequest();

    ClientResponse proceed() throws Exception;
}
```

They work similarly to **MessageBodyReader** in that you must call **proceed()** or the invocation will be aborted.

28.5. BINDING INTERCEPTORS

By default, any registered interceptor will be invoked for every request. You can alter this by having your interceptors implement the **org.jboss.resteasy.spi.AcceptedByMethod** interface:

```
public interface AcceptedByMethod
{
    public boolean accept(Class declaring, Method method);
```

```
        }
```

If your interceptor implements this interface, RESTEasy invokes the `accept()` method. If this method returns **true**, RESTEasy adds that interceptor to the JAX-RS method's call chain. If it returns **false**, the interceptor will not be added to the call chain. For example:

```
@Provider
@ServerInterceptor
public class MyHeaderDecorator implements MessageBodyWriterInterceptor,
AcceptedByMethod {

    public boolean accept(Class declaring, Method method) {
        return method.isAnnotationPresent(GET.class);
    }

   public void write(MessageBodyWriterContext context) throws IOException,
WebApplicationException
   {
      context.getHeaders().add("My-Header", "custom");
      context.proceed();
   }
}
```

In this example, the `accept()` method checks whether the `@GET` annotation exists in the JAX-RS method. If it does, the interceptor will be applied to that method's call chain.

28.6. REGISTERING INTERCEPTORS

When your interceptors are annotated as @**Provider**s, they can be listed in the `resteasy.providers context-param` in `web.xml`, or returned as a class or object in the `Application.getClasses()` or `Application.getSingletons()` method.

28.7. INTERCEPTOR ORDERING AND PRECEDENCE

Some interceptors are sensitive to the order in which they are invoked. For example, your security interceptor should always be invoked first. Other interceptors' behavior can be triggered by an interceptor that adds a header. By default, you have no control over the order in which registered interceptors are invoked, but you can specify interceptor *precedence*.

Interceptor precedence is not specified by listing interceptor classes. Instead, a particular interceptor class is associated with a *precedence family* with the `@org.jboss.resteasy.annotations.interception.Precedence` annotation. Specifying precedence through a family structure protects the built-in interceptors that are sensitive to ordering and simplifies configuration.

The families are listed here in execution order:

```
    SECURITY
    HEADER_DECORATOR
    ENCODER
    REDIRECT
    DECODER
```

Any interceptor not associated with a precedence family will be invoked last. **SECURITY** usually includes **PreProcessInterceptor**s. These should be invoked first so that as little as possible occurs prior to authorization. **HEADER_DECORATOR**s are interceptors that add headers to a response or an outgoing request. These are next in precedence because the added headers may affect the behavior of other interceptors. **ENCODER** interceptors change the **OutputStream**. For example, the GZIP interceptor creates a **GZIPOutputStream** to wrap the real **OutputStream** for compression. **REDIRECT** interceptors are usually used in **PreProcessInterceptor**s because they can reroute the request and bypass the JAX-RS method. **DECODER** interceptors wrap the **InputStream**. For example, the GZIP interceptor decoder wraps the **InputStream** in a **GzipInputStream** instance.

To associate your custom interceptors with a particular family, annotate it with **@org.jboss.resteasy.annotations.interception.Precendence annotation**.

```
@Provider
@ServerInterceptor
@ClientInterceptor
@Precedence("ENCODER")
public class MyCompressionInterceptor implements
MessageBodyWriterInterceptor {...}
```

There are convenience annotations in the **org.jboss.resteasy.annotations.interception** package to provide complete type safety: **@DecoredPrecedence**, **@EncoderPrecedence**, **@HeaderDecoratorPrecedence**, **@RedirectPrecedence**, and **@SecurityPrecedence**. Use these instead of the **@Precedence** annotation

28.7.1. Custom Precedence

You can define your own precedence families and apply them with the **@Precedence** annotation.

```
@Provider
@ServerInterceptor
@Precedence("MY_CUSTOM_PRECEDENCE")
public class MyCustomInterceptor implements MessageBodyWriterInterceptor
{...}
```

You can create your own convenience annotation by using **@Precedence** as a meta-annotation.

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Precedence("MY_CUSTOM_PRECEDENCE")
public @interface MyCustomPrecedence {}
```

You must register your custom precedence, or RESTEasy will show an error at deployment time. You can register your custom precendence with the context parameters:

```
resteasy.append.interceptor.precedence
resteasy.interceptor.before.precedence
resteasy.interceptor.after.precedence
```

resteasy.append.interceptor.precedence appends the precedence family to the list. **resteasy.interceptor.before.precedence** lets you specify a family for your precendence to fall ahead of. **resteasy.interceptor.after.precedence** lets you specify a family for your precedence to follow after. For example:

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

    <!-- testing configuration -->
    <context-param>
        <param-name>resteasy.append.interceptor.precedence</param-name>
        <param-value>END</param-value>
    </context-param>
    <context-param>
        <param-name>resteasy.interceptor.before.precedence</param-name>
        <param-value>ENCODER : BEFORE_ENCODER</param-value>
    </context-param>

    <context-param>
        <param-name>resteasy.interceptor.after.precedence</param-name>
        <param-value>ENCODER : AFTER_ENCODER</param-value>
    </context-param>

    <context-param>
        <param-name>resteasy.servlet.mapping.prefix</param-name>
        <param-value>/test</param-value>
    </context-param>

    <listener>
        <listener-
class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listene
r-class>
    </listener>

    <servlet>
        <servlet-name>Resteasy</servlet-name>
        <servlet-
class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</ser
vlet-class>
    </servlet>

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

</web-app>
```

In this **web.xml** file, we have defined three new precedence families: **END, BEFORE_ENCODER,** and
AFTER_ENCODER. With this configuration, the family order would look like this:

```
SECURITY
HEADER_DECORATOR
BEFORE_ENCODER
ENCODER
AFTER_ENCODER
REDIRECT
DECODER
END
```

CHAPTER 29. ASYNCHRONOUS HTTP REQUEST PROCESSING

Asynchronous HTTP Request Processing lets you process a single HTTP request using NIO (Non-blocking Input/Output), in separate threads (if desired). This is also known as *COMET capability*. The primary use case for Asynchronous HTTP is where the client polls the server for a delayed response.

A common example is an AJAX chat client that pushes/pulls from both the client and the server. In this scenario, the client blocks a long period of time on the server's socket while waiting for a new message. In synchronous HTTP (where the server blocks on incoming and outgoing I/O), one thread is consumed per client connection, which consumes both memory and thread resources. When multiple concurrent clients block in this way, resources are not used effectively, and the server does not scale well.

Tomcat, Jetty, and JBoss Web all have similar (proprietary) support for asynchronous HTTP request processing. This functionality is currently being standardized in the Servlet 3.0 specification. RESTEasy provides a simple callback API to provide asynchronous capabilities, and supports integration with Servlet 3.0 (through Jetty 7), Tomcat 6, and JBoss Web 2.1.1.

The RESTEasy asynchronous HTTP support is implemented via two classes: the **@Suspend** annotation and the **AsynchronousResponse** interface.

```
public @interface Suspend
{
    long value() default -1;
}

import javax.ws.rs.core.Response;

public interface AsynchronousResponse
{
    void setResponse(Response response);
}
```

@Suspend tells RESTEasy that the HTTP request/response should be detached from the currently executing thread, and that the current thread should not automatically process the response. The argument to **@Suspend** is the time in milliseconds until the request should be canceled.

AsynchronousResponse is the callback object. It is injected into the method by RESTEasy. Application code moves the **AsynchronousResponse** to a different thread for processing. Calling **setResponse()** returns a response to the client and terminates the HTTP request. The following is an example of asynchronous processing:

```
import org.jboss.resteasy.annotations.Suspend;
import org.jboss.resteasy.spi.AsynchronousResponse;

@Path("/")
public class SimpleResource
{

    @GET
    @Path("basic")
    @Produces("text/plain")
    public void getBasic(final @Suspend(10000) AsynchronousResponse
response) throws Exception
    {
        Thread t = new Thread()
```

```
        {
            @Override
            public void run()
            {
                try
                {
                    Response jaxrs =
Response.ok("basic").type(MediaType.TEXT_PLAIN).build();
                    response.setResponse(jaxrs);
                }
                catch (Exception e)
                {
                    e.printStackTrace();
                }
            }
        };
        t.start();
    }
}
```

29.1. TOMCAT 6 AND JBOSS 4.2.3 SUPPORT

To use RESTEasy's Asynchronous HTTP APIs with Tomcat 6 or JBoss 4.2.3, you must use a special RESTEasy Servlet and configure Tomcat (or JBoss Web in JBoss 4.2.3) to use the NIO transport. First, edit Tomcat's (or JBoss Web's) `server.xml` file. Comment out the **vanilla HTTP adapter** and add the following:

```
<Connector port="8080" address="${jboss.bind.address}"
    emptySessionPath="true"
protocol="org.apache.coyote.http11.Http11NioProtocol"
    enableLookups="false" redirectPort="6443" acceptorThreadCount="2"
pollerThreadCount="10"
/>
```

Your deployed RESTEasy applications must also use a different RESTEasy Servlet: `org.jboss.resteasy.plugins.server.servlet.Tomcat6CometDispatcherServlet`. This class is available within the `async-http-tomcat-xxx.jar` (or within the Maven repository, under the `async-http-tomcat6` artifact ID) in `web.xml`.

```
<servlet>
    <servlet-name>Resteasy</servlet-name>
    <servlet-
class>org.jboss.resteasy.plugins.server.servlet.Tomcat6CometDispatcherServ
let</servlet-class>
</servlet>
```

29.2. SERVLET 3.0 SUPPORT

As of October 20th, 2008, only Jetty 7.0.pre3 (mortbay.org) supported the current draft of the unfinished Servlet 3.0 specification.

Your deployed RESTEasy applications must also use a different RESTEasy Servlet: `org.jboss.resteasy.plugins.server.servlet.HttpServlet30Dispatcher`. This class is

available within the **async-http-servlet-3.0-xxx.jar** (or within the Maven repository under the **async-http-servlet-3.0** artifact ID) in **web.xml**:

```
<servlet>
    <servlet-name>Resteasy</servlet-name>
    <servlet-
class>org.jboss.resteasy.plugins.server.servlet.HttpServlet30Dispatcher</s
ervlet-class>
</servlet>
```

29.3. JBOSSWEB, EAP 5.0.X SUPPORT

The JBossWeb container is shipped with EAP 5.0.x and higher requires the JBoss Native plug-in to enable asynchronous HTTP processing. See the JBoss Web documentation for information about this process.

Your deployed RESTEasy applications must use a different RESTEasy Servlet: **org.jboss.resteasy.plugins.server.servlet.JBossWebDispatcherServlet**. This class is available within the **async-http-jbossweb-xxx.jar** (or within the Maven repository under the **async-http-jbossweb** artifact ID) in **web.xml**:

```
<servlet>
    <servlet-name>Resteasy</servlet-name>
    <servlet-
class>org.jboss.resteasy.plugins.server.servlet.JBossWebDispatcherServlet<
/servlet-class>
</servlet>
```

CHAPTER 30. ASYNCHRONOUS JOB SERVICE

The RESTEasy Asynchronous Job Service is an implementation of the Asynchronous Job pattern defined in O'Reilly's *Restful Web Services*. It is designed to add asynchronicity to a synchronous protocol.

30.1. USING ASYNC JOBS

While HTTP is a synchronous protocol, it is capable of dealing with asynchronous invocations. The HTTP 1.1 response code **202** (Accepted) means that the server has received and accepted the response for processing, but that processing is not yet complete. The RESTEasy Asynchronous Job Service is based on this type of response.

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

For example, if you make the above post with the **asynch** query parameter set to **true**, RESTEasy returns a **202** (Accepted) response code and runs the invocation in the background. It also returns a Location header with a URL pointing to the location of the background method's response.

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

The URI will have the form of:

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

You can perform **GET**, **POST** and **DELETE** operations on this job URL. **GET** returns the response of the JAX-RS resource method, if the job has completed. If the job has not completed, this **GET** returns a response code of **202** (Accepted). Invoking **GET** does not remove the job, so it can be called multiple times. When RESTEasy's job queue becomes full, it will evict the least recently used job from memory. You can clean the queue manually by calling **DELETE** on the URI. **POST** reads the **JOB** response and removes the **JOB** when it has completed.

Both **GET** and **POST** let you specify a maximum wait time in milliseconds — a **wait** query parameter. For example:

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

If you do not specify a **wait** parameter, the **GET** or **POST** will not wait at all if the job is not complete.

NOTE

While you can invoke **GET**, **DELETE**, and **PUT** methods asynchronously, this breaks the HTTP 1.1 contract of these methods. These invocations may not change the state of the resource if invoked more than once, but they do change the state of the server. Try to invoke POST methods asynchronously.
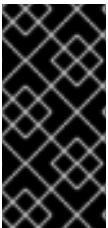
> **IMPORTANT**
>
> RESTEasy role-based security (annotations) does not work with the Asynchronous Job Service. *You must use XML declarative security within your* `web.xml` *file.* It is currently impossible to implement role-based security portably. In the future, we may have specific JBoss integration, but will not support other environments.

30.2. ONEWAY: FIRE AND FORGET

RESTEasy also supports the notion of *fire and forget*. This returns a **202** (Accepted) response, but no Job is created. Use the **oneway** query parameter instead of **asynch**, like so:

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

> **IMPORTANT**
>
> RESTEasy role-based security (annotations) does not work with the Asynchronous Job Service. *You must use XML declarative security within your* `web.xml` *file.* It is currently impossible to implement role-based security portably. In the future, we may have specific JBoss integration, but will not support other environments.

30.3. SETUP AND CONFIGURATION

The Asynchronous Job Service is not enabled by default, so you will need to enable it in your `web.xml`:

```xml
<web-app>
    <!-- enable the Asynchronous Job Service -->
    <context-param>
        <param-name>resteasy.async.job.service.enabled</param-name>
        <param-value>true</param-value>
    </context-param>

    <!-- The next context parameters are all optional.
         Their default values are shown as example param-values -->

    <!-- How many jobs results can be held in memory at once? -->
    <context-param>
        <param-name>resteasy.async.job.service.max.job.results</param-name>
        <param-value>100</param-value>
    </context-param>

    <!-- Maximum wait time on a job when a client is querying for it -->
    <context-param>
        <param-name>resteasy.async.job.service.max.wait</param-name>
        <param-value>300000</param-value>
    </context-param>

    <!-- Thread pool size of background threads that run the job -->
    <context-param>
        <param-name>resteasy.async.job.service.thread.pool.size</param-name>
        <param-value>100</param-value>
```

```
        </context-param>

        <!-- Set the base path for the Job uris -->
        <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>
```

CHAPTER 31. EMBEDDED CONTAINER

RESTEasy JAX-RS comes with an embeddable server that can be run from your classpath. It packages the TJWS (Tiny Java Web Server) embeddable Servlet container with JAX-RS.

From the distribution, move the **JARs** in `resteasy-jaxrs.war/WEB-INF/lib` into your classpath. You must register your JAX-RS beans programmatically using the embedded server's **Registry**. Here's an example:

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

    @GET
    public String get() { return "hello world"; }


    public static void main(String[] args) throws Exception
    {
        TJWSEmbeddedJaxrsServer tjws = new TJWSEmbeddedJaxrsServer();
        tjws.setPort(8081);
        tjws.getRegistry().addPerRequestResource(MyResource.class);
        tjws.start();
    }
}
```

The server can either host non-encrypted or SSL-based resources, but not both. See the Java Documentation for **TJWSEmbeddedJaxrsServer** and its superclass **TJWSServletServer** for further information. The TJWS website is also very informative.

To use Spring, you will need the **SpringBeanProcessor**. Here is a pseudo-code example:

```
    public static void main(String[] args) throws Exception
    {
        final TJWSEmbeddedJaxrsServer tjws = new TJWSEmbeddedJaxrsServer();
        tjws.setPort(8081);

        org.resteasy.plugins.server.servlet.SpringBeanProcessor processor =
new SpringBeanProcessor(tjws.getRegistry(), tjws.getFactory());
        ConfigurableBeanFactory factory = new XmlBeanFactory(...);
        factory.addBeanPostProcessor(processor);

        tjws.start();
    }
```

CHAPTER 32. SERVER-SIDE MOCK FRAMEWORK

RESTEasy provides a mock framework so that you can invoke directly on your resource, without using the embeddable container.

```
import org.resteasy.mock.*;
...

    Dispatcher dispatcher = MockDispatcherFactory.createDispatcher();

    POJOResourceFactory noDefaults = new
POJOResourceFactory(LocatingResource.class);
    dispatcher.getRegistry().addResourceFactory(noDefaults);

    {
        MockHttpRequest request = MockHttpRequest.get("/locating/basic");
        MockHttpResponse response = new MockHttpResponse();

        dispatcher.invoke(request, response);


        Assert.assertEquals(HttpServletResponse.SC_OK,
response.getStatus());
        Assert.assertEquals("basic", response.getContentAsString());
    }
```

See the RESTEasy Java Documentation for a complete list of the methods associated with **MockHttpRequest** and **MockHttpResponse**.

CHAPTER 33. SECURING JAX-RS AND RESTEASY

Because RESTEasy is deployed as a Servlet, you must use standard `web.xml` constraints to enable authentication and authorization.

Unfortunately, `web.xml` constraints have limited compatibility with JAX-RS because of the limited URL pattern matching available in `web.xml`. URL patterns in `web.xml` support only simple wildcards, so JAX-RS resources like the following:

```
/{pathparam1}/foo/bar/{pathparam2}
```

Cannot be mapped as a `web.xml` URL pattern such as:

```
/*/foo/bar/*
```

To work around this problem, use the following security annotations on your JAX-RS methods. You must also set up some general security constraint elements in `web.xml` to enable authentication.

RESTEasy JAX-RS supports the @**RolesAllowed**, @**PermitAll** and @**DenyAll** annotations on JAX-RS methods. By default, RESTEasy does not recognize these annotations. You must configure RESTEasy to enable role-based security by setting a context parameter, like so:
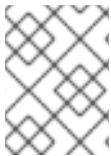
> **NOTE**
>
> Do not enable this if you are using EJBs. The EJB container will provide this function instead of RESTEasy.

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

With this approach, you must declare all roles used within both the RESTEasy JAX-RS **WAR** file, and in your JAX-RS classes, and establish a security constraint that lets these roles access every URL handled by the JAX-RS runtime, assuming that RESTEasy authorizes correctly.

RESTEasy authorization checks if a method is annotated with @**RolesAllowed** and then performs `HttpServletRequest.isUserInRole`. If one of the the @**RolesAllowed** passes, the request is allowed. If not, a response is returned with a **401** (Unauthorized) response code.

The following is an example of a modified RESTEasy WAR file. Note that every role declared is allowed access to every URL controlled by the RESTEasy Servlet.

```
<web-app>

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

```
   <listener>
      <listener-
class>org.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-
class>
   </listener>

   <servlet>
      <servlet-name>Resteasy</servlet-name>
      <servlet-
class>org.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-
class>
   </servlet>

   <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>

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

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


</web-app>
```

CHAPTER 34. EJB INTEGRATION

To integrate with Enterprise JavaBeans (EJB), you must first modify your EJB's published interfaces. Currently, RESTEasy only has simple portable integration with EJBs, so you must manually configure your RESTEasy **WAR**.

To make an EJB a JAX-RS resource, annotate a stateless session bean's **@Remote** or **@Local** interface with JAX-RS annotations, as follows:

```
@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 RESTEasy's **web.xml**, manually register the EJB with RESTEasy by 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.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-
class>
    </listener>

    <servlet>
        <servlet-name>Resteasy</servlet-name>
        <servlet-
class>org.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-
class>
    </servlet>

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

</web-app>
```

At present, this is the only portable integration method for EJBs. Future versions of RESTEasy will be more tightly integrated with EAP, so manual registrations and modifications to `web.xml` will be unnecessary.

If you are using RESTEasy with an **EAR** and EJBs, the following structure is helpful:

```
my-ear.ear
|------myejb.jar
|------resteasy-jaxrs.war
        |
       ----WEB-INF/web.xml
       ----WEB-INF/lib (nothing)
|------lib/
        |
       ----All Resteasy jar files
```

Remove all libraries from **WEB-INF/lib** and place them in a common **EAR** library, or place the RESTEasy **JAR** dependencies in your application server's system classpath (that is, in EAP, place them in **server/default/lib**).

CHAPTER 35. SPRING INTEGRATION

RESTEasy integrates with Spring 2.5. (We are interested in other forms of Spring integration, and encourage you to contribute.)

35.1. BASIC INTEGRATION

For Maven users, you must use the resteasy-spring artifact. Otherwise, the jar is available in the downloaded distribution.

```
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-spring</artifactId>
    <version>whatever version you are using</version>
</dependency>
```

RESTEasy includes its own Spring **ContextLoaderListener**. This 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.

You will need to alter your `web.xml` file:

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

    <listener>
        <listener-
class>org.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-
class>
    </listener>

    <listener>
        <listener-
class>org.resteasy.plugins.spring.SpringContextLoaderListener</listener-
class>
    </listener>

    <servlet>
        <servlet-name>Resteasy</servlet-name>
        <servlet-
class>org.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-
class>
    </servlet>

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


</web-app>
```

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

If you do not use a Spring **ContextLoaderListener** to create your bean factories, you can manually register the RESTEasy **BeanFactoryPostProcessor** by allocating an instance of **org.jboss.resteasy.plugins.spring.SpringBeanProcessor**. You can obtain instances of a **ResteasyProviderFactory** and **Registry** from the **ServletContext** attribute, **org.resteasy.spi.ResteasyProviderFactory and org.resteasy.spi.Registry** (the fully-qualified name String of these classes). There is also **org.jboss.resteasy.plugins.spring.SpringBeanProcessorServletAware**, which automatically injects references to the **Registry** and **ResteasyProviderFactory** from the Servlet Context, assuming that you have used **RestasyBootstrap** to bootstrap RESTEasy.

RESTEasy Spring integration supports both singletons and the *prototype* scope. It handles injecting @**Context** references. However, constructor injection is not supported, and when the *prototype* scope is used, RESTEasy injects any @***Param**-annotated fields and setters before the request is dispatched.
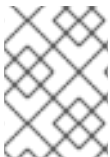
> **NOTE**
>
> You can only use automatically proxied beans with RESTEasy Spring integration. Hand-coded proxying (that is, with **ProxyFactoryBean**) will have undesirable effects.

35.2. SPRING MVC INTEGRATION

RESTEasy can also integrate with the Spring **DispatcherServlet**. This is advantageous because the **web.xml** is simpler, you can dispatch to either Spring controllers or RESTEasy from under the same base URL, and you can use Spring **ModelAndView** objects as return arguments from @**GET** resource methods. To set this up, you must use the Spring **DispatcherServlet** in your **web.xml** file, and import the **springmvc-resteasy.xml** file into your base Spring beans xml file. Here is an example **web.xml** file:

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

    <servlet>
        <servlet-name>Resteasy</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet;
</servlet-class>
    </servlet>

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


</web-app>
```

Then, within your main Spring beans xml file, import the **springmvc-resteasy.xml** file.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
```

```
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd
        http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.5.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
">

    <!-- Import basic SpringMVC Resteasy integration -->
    <import resource="classpath:springmvc-resteasy.xml"/>
....
```

CHAPTER 36. GUICE 1.0 INTEGRATION

RESTEasy does have some basic integration with Guice 1.0. RESTEasy scans a Guice Module's binding types for @**Path** and @**Provider** annotations, and registers these bindings. The **guice-hello** project that comes in the RESTEasy **examples/** directory gives a nice example of this.

```
@Path("hello")
public class HelloResource
{
    @GET
    @Path("{name}")
    public String hello(@PathParam("name") final String name) {
        return "Hello " + name;
    }
}
```

Start by specifying a JAX-RS resource class — in this case, the **HelloResource**. Next, create a Guice Module class that defines your bindings:

```
import com.google.inject.Module;
import com.google.inject.Binder;

public class HelloModule implements Module
{
    public void configure(final Binder binder)
    {
        binder.bind(HelloResource.class);
    }
}
```

Place these classes within your **WAR WEB-INF/classes** or in a **JAR** within **WEB-INF/lib**. Then, create your **web.xml** file. You will need to use the **GuiceResteasyBootstrapServletContextListener** like so:

```
<web-app>
    <display-name>Guice Hello</display-name>

    <context-param>
      <param-name>
        resteasy.guice.modules
      </param-name>
      <param-value>
        org.jboss.resteasy.examples.guice.hello.HelloModule
      </param-value>
    </context-param>

    <listener>
      <listener-class>

org.jboss.resteasy.plugins.guice.GuiceResteasyBootstrapServletContextListe
ner
      </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>
```

GuiceResteasyBootstrapServletContextListener is a subclass of **ResteasyBootstrap**, so you can use any other RESTEasy configuration option within your **web.xml** file. Also note the **resteasy.guice.modules** context parameter. This can take a comma delimited list of class names that are Guice Modules.

CHAPTER 37. CLIENT FRAMEWORK

The RESTEasy Client Framework is the alternative to the JAX-RS server-side specification. Instead of using JAX-RS annotations to map an incoming request to your RESTful Web Service method, the client framework creates a HTTP request to invoke on a remote RESTful Web Service, which can be any web resource that accepts HTTP requests.

RESTEasy has a client proxy framework that lets you invoke upon a remote HTTP resource by using JAX-RS annotations. You can write a Java interface and use JAX-RS annotations on methods and the interface. For example:

```java
public interface SimpleClient
{
    @GET
    @Path("basic")
    @Produces("text/plain")
    String getBasic();

    @PUT
    @Path("basic")
    @Consumes("text/plain")
    void putBasic(String body);

    @GET
    @Path("queryParam")
    @Produces("text/plain")
    String getQueryParam(@QueryParam("param")String param);

    @GET
    @Path("matrixParam")
    @Produces("text/plain")
    String getMatrixParam(@MatrixParam("param")String param);

    @GET
    @Path("uriParam/{param}")
    @Produces("text/plain")
    int getUriParam(@PathParam("param")int param);
}
```

The RESTEasy API is simple, and based on Apache HttpClient. You generate a proxy, and invoke methods on the proxy. The invoked method is then translated to a HTTP request (based on the method's annotations) and posted to the server. To set it up:

```java
        import org.resteasy.plugins.client.httpclient.ProxyFactory;
        ...
        // this initialization only needs to be done once per VM

RegisterBuiltin.register(ResteasyProviderFactory.getInstance());


        SimpleClient client = ProxyFactory.create(SimpleClient.class,
"http://localhost:8081");
        client.putBasic("hello world");
```

See the **ProxyFactory** Java Documentation for more options. For instance, you may want to fine tune the **HttpClient** configuration.

@CookieParam creates a cookie header to send to the server. If you allocate your own **javax.ws.rs.core.Cookie** object and pass it as a parameter to a client proxy method, you do not require @**CookieParam** — the client framework understands that you are passing a cookie to the server, so no extra metadata is required.

The client framework can use the same providers available on the server. You must manually register them through the **ResteasyProviderFactory** singleton using the **addMessageBodyReader()** and **addMessageBodyWriter()** methods.

```
ResteasyProviderFactory.getInstance().addMessageBodyReader(MyReader.class)
;
```

37.1. ABSTRACT RESPONSES

When you need to access the response code or the response headers of a client request, the **Client-Proxy** framework provides two options:

You can return a **javax.ws.rs.core.Response.Status** enumeration from your method calls, like so:

```
@Path("/")
public interface MyProxy {
    @POST
    Response.Status updateSite(MyPojo pojo);
}
```

After invoking on the server, the client proxy internally converts the HTTP response code into a **Response.Status** enumeration.

You can retrieve all data associated with a request with the **org.resteasy.spi.ClientResponse** interface:

```
/**
 * Response extension for the RESTEasy client framework. Use this, or
Response
 * in your client proxy interface method return type declarations if you
want
 * access to the response entity as well as status and header information.
 *
 * @author <a href="mailto:bill@burkecentral.com">Bill Burke</a>
 * @version $Revision: 1 $
 */
public abstract class ClientResponse<T> extends Response
{
    /**
     * This method returns the same exact map as Response.getMetadata()
except as a map of strings
     * rather than objects.
     *
     * @return
```

```
     */
    public abstract MultivaluedMap<String, String> getHeaders();

    public abstract Response.Status getResponseStatus();

    /**
     * Unmarshal the target entity from the response OutputStream.  You
must have type information
     * set via <T> otherwise, this will not work.
     * <p/>
     * This method actually does the reading on the OutputStream.  It will
only do the read once.
     * Afterwards, it will cache the result and return the cached result.
     *
     * @return
     */
    public abstract T getEntity();

    /**
     * Extract the response body with the provided type information
     * <p/>
     * This method actually does the reading on the OutputStream.  It will
only do the read once.
     * Afterwards, it will cache the result and return the cached result.
     *
     * @param type
     * @param genericType
     * @param <T2>
     * @return
     */
    public abstract <T2> T2 getEntity(Class<T2> type, Type genericType);

    /**
     * Extract the response body with the provided type information.
GenericType is a trick used to
     * pass in generic type information to the resteasy runtime.
     * <p/>
     * For example:
     * <pre>
     * List<String> list = response.getEntity(new GenericType<List<String>
() {});
     * <p/>
     * <p/>
     * This method actually does the reading on the OutputStream.  It will
only do the read once.  Afterwards, it will
     * cache the result and return the cached result.
     *
     * @param type
     * @param <T2>
     * @return
     */
    public abstract <T2> T2 getEntity(GenericType<T2> type);
}
```

All **getEntity()** methods are deferred until you invoke them. In other words, the response
OutputStream is not read until you call one of these methods. The **getEntity()** method with no

parameters can only be used if you have templated the **ClientResponse** within your method declaration. RESTEasy uses this generic type information to determine which media type the **OutputStream** is unmarshaled into. The **getEntity()** methods that take parameters let you specify the Object type the response should be marshaled into. This lets you dynamically extract the desired types at runtime. For example:

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

    @GET
    @Produces("application/xml")
    ClientResponse<LibraryPojo> getAllBooks();
}
```

Include the **LibraryPojo** in **ClientResponse**'s generic declaration so that the client proxy framework can unmarshal the HTTP response body.

37.2. SHARING AN INTERFACE BETWEEN CLIENT AND SERVER

It is usually possible to share an interface between the client and server. In the previous scenario, your JAX-RS services must implement an annotated interface, then reuse that same interface to create client proxies to invoke on the client side. However, this is limited when your JAX-RS methods return a Response object. This is problematic because, in a raw Response return type declaration, the client has no type information. There are two ways to work around this problem. The first is to use the @**ClientResponseType** annotation.

```
import org.jboss.resteasy.annotations.ClientResponseType;
import javax.ws.rs.core.Response;

@Path("/")
public interface MyInterface {

    @GET
    @ClientResponseType(String.class)
    @Produces("text/plain")
    public Response get();
}
```

This will not always work, because some **MessageBodyReader**s and **MessageBodyWriter**s need generic type information in order to match and service a request.

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

    @GET
    @Produces("application/xml")
    public Response getMyListOfJAXBObjects();
}
```

In this case, your client code can cast the returned Response object to a **ClientResponse** and use one of the typed **getEntity()** methods.

```
MyInterface proxy = ProxyFactory.create(MyInterface.class,
"http://localhost:8081");
```

```
ClientResponse response = (ClientResponse)proxy.getMyListOfJAXBObjects();
List<MyJaxbClass> list = response.getEntity(new
GenericType<List<MyJaxbClass>>());
```

37.3. CLIENT ERROR HANDLING

If you are using the Client Framework and your proxy methods return something other than a **ClientResponse**, the default client error handling comes into play. Any response code that is greater than **399** will automatically cause a **org.jboss.resteasy.client.ClientResponseFailure** exception:

```
    @GET
    ClientResponse<String> get() // will throw an exception if you call
getEntity()

    @GET
    MyObject get(); // will throw a ClientResponseFailure on response code
> 399
```

37.4. MANUAL CLIENTREQUEST API

RESTEasy has a manual API for invoking requests: **org.jboss.resteasy.client.ClientRequest** See the Java Documentation for a complete description of this class. A simple example is the following:

```
    ClientRequest request = new
ClientRequest("http://localhost:8080/some/path");
    request.header("custom-header", "value");

    // we are posting XML and a JAXB object
    request.body("application/xml", someJaxb);

    // we are expecting a String back
    ClientResponse<String> response = request.post(String.class);

    if (response.getStatus() == 200) // OK!
    {
        String str = response.getEntity();
    }
```

CHAPTER 38. MAVEN AND RESTEASY

The JBoss Maven repository is located at http://repository.jboss.org/maven2.

You can combine them with the following **pom.xml** fragment. RESTEasy is divided into various components. You can mix and match components as required. Remember to replace *1.1.GA* with the RESTEasy version you want to use.

```xml
<repositories>
   <repository>
      <id>jboss</id>
      <url>http://repository.jboss.org/maven2</url>
   </repository>
</repositories>
<dependencies>
   <!-- core library -->
   <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-jaxrs</artifactId>
      <version>1.1.GA</version>
   </dependency>


   <!-- optional modules -->

   <!-- JAXB support -->
   <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-jaxb-provider</artifactId>
      <version>1.1.GA</version>
   </dependency>
   <!-- multipart/form-data and multipart/mixed support -->
   <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-multipart-provider</artifactId>
      <version>1.1.GA</version>
   </dependency>
   <!-- Resteasy Server Cache -->
   <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-cache-core</artifactId>
      <version>1.1.GA</version>
   </dependency>
   <!-- Ruby YAML support -->
   <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-yaml-provider</artifactId>
      <version>1.1.GA</version>
   </dependency>
   <!-- JAXB + Atom support -->
   <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-atom-provider</artifactId>
      <version>1.1.GA</version>
   </dependency>
   <!-- JAXB + Atom support -->
   <dependency>
```

```
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>resteasy-atom-provider</artifactId>
        <version>1.1.GA</version>
    </dependency>
    <!-- Apache Abdera Integration -->
    <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>abdera-atom-provider</artifactId>
        <version>1.1.GA</version>
    </dependency>
    <!-- Spring integration -->
    <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>resteasy-spring</artifactId>
        <version>1.1.GA</version>
    </dependency>
    <!-- Guice integration -->
    <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>resteasy-guice</artifactId>
        <version>1.1.GA</version>
    </dependency>

    <!-- Asynchronous HTTP support with JBossWeb -->
    <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>async-http-jbossweb</artifactId>
        <version>1.1.GA</version>
    </dependency>

    <!-- Asynchronous HTTP support with Servlet 3.0 (Jetty 7 pre5) -->
    <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>async-http-servlet-3.0</artifactId>
        <version>1.1.GA</version>
    </dependency>

    <!-- Asynchronous HTTP support with Tomcat 6 -->
    <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>async-http-tomcat6</artifactId>
        <version>1.1.GA</version>
    </dependency>

</dependencies>
```

You can also import a **POM** to ensure that individual module versions need not me specified.

NOTE

This requires Maven 2.0.9.

```
  <dependencyManagement>
     <dependencies>
         <dependency>
```

```
            <groupId>org.jboss.resteasy</groupId>
            <artifactId>resteasy-maven-import</artifactId>
            <version<1.1.GA</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

CHAPTER 39. JBOSS 5.X INTEGRATION

Resteasy 1.1.GA has no special integration with JBoss Application Server so it must be configured and installed like any other container. There are some issues though. You must make sure that there is not a copy of servlet-api-xxx.jar in your WEB-INF/lib directory as this may cause problems. Also, if you are running with JDK 6, make sure to filter out the JAXB jars as they come with JDK 6.

CHAPTER 40. MIGRATION FROM OLDER VERSIONS

40.1. MIGRATING FROM 1.0.X AND 1.1-RC1

You can expect the following changes when you migrate to the latest version of RESTEasy:

- You can now turn on RESTEasy's role-based security (**@RolesAllowed**) by using the new **resteasy.role.based.security context-param**.

- **@Wrapped** is now enabled by default for Lists, Arrays, and Sets of JAXB objects. You can also change the namespace and element names with this annotation.

- **@Wrapped** is not enclosed in a RESTEasy namespace prefix, and now uses the default namespace instead of the **http://jboss.org/resteasy** namespace.

- **@Wrapped JSON** is now enclosed in a simple JSON Array.

- Placing the **resteasy-jackson-provider-xxx.jar** in your classpath triggers the Jackson JSON provider. This can cause code errors if you had previously been using the Jettison JAXB/JSON providers. To fix this, you must either remove Jackson from your **WEB-INF/lib** or the classpath, or use the **@NoJackson** annotation on your JAXB classes.

- The **tjws** and **servlet-api** artifacts are now scoped as *provided* in the **resteasy-jar** dependencies. If you have trouble with *Class not found* errors, you may need to scope them as **provided** or **test** in your **pom** files.

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>tjws</groupId>
    <artifactId>webserver</artifactId>
    <scope>provided</scope>
</dependency>
```

APPENDIX A. REVISION HISTORY

Revision 5.2.0-100.400   2013-10-31   **Rüdiger Landmann**
   Rebuild with publican 4.0.0

Revision 5.2.0-100   **Wed 23 Jan 2013**   **Russell Dickenson**
   Incorporated changes for JBoss Enterprise Application Platform 5.2.0 GA. For information about documentation changes to
   this guide, refer to *Release Notes 5.2.0*.

Revision 5.1.2-100   **Thu 8 December 2011**   **Russell Dickenson**
   Incorporated changes for JBoss Enterprise Application Platform 5.1.2 GA. For information about documentation changes to
   this guide, refer to *Release Notes 5.1.2*