RESTEasy JAX-RS

# RESTFul Web Services for Java

2.0.1.GA

## Preface

Commercial development support, production support and training for RESTEasy JAX-RS is available through JBoss, a division of Red Hat Inc. (see http://www.jboss.com/).

In some of the example listings, what is meant to be displayed on one line does not fit inside the available page width. These lines have been broken up. A '\' at the end of a line means that a break has been introduced to fit in the page, with the following lines indented. So:

```
Let's pretend to have an extremely \
long line that \
does not fit
This one is short
```

Is really:

```
Let's pretend to have an extremely long line that does not fit
This one is short
```

# Overview

JAX-RS, JSR-311, is a new JCP specification that provides a Java API for RESTful Web Services over the HTTP protocol. Resteasy is an portable implementation of this specification which can run in any Servlet container. Tighter integration with JBoss Application Server is also available to make the user experience nicer in that environment. While JAX-RS is only a server-side specification, Resteasy has innovated to bring JAX-RS to the client through the RESTEasy JAX-RS Client Framework. This client-side framework allows you to map outgoing HTTP requests to remote servers using JAX-RS annotations and interface proxies.

- JAX-RS implementation

- Portable to any app-server/Tomcat that runs on JDK 5 or higher

- Embeddedable server implementation for junit testing

- EJB and Spring integration

- Client framework to make writing HTTP clients easy (JAX-RS only define server bindings)

# License

RESTEasy is distributed under the ASL 2.0 license. It does not distribute any thirdparty libraries that are GPL. It does ship thirdparty libraries licensed under Apache ASL 2.0 and LGPL.

# Installation/Configuration

RESTEasy is installed and configured in different ways depending on which environment you are running in. If you are running in JBoss AS 6-M4 (milestone 4) or higher, resteasy is already bundled and integrated completely so there is very little you have to do. If you are running in a different distribution, there is some manual installation and configuration you will have to do.

## 3.1. Standalone Resteasy

If you are using resteasy outside of JBoss AS 6, you will need to do a few manual steps to install and configure resteasy. RESTeasy is deployed as a WAR archive and thus depends on a Servlet container. We strongly suggest that you use Maven to build your WAR files as RESTEasy is split into a bunch of different modules. You can see an example Maven project in one of the examples in the examples/ directory

Also, when you download RESTeasy and unzip it you will see a lib/ directory that contains the libraries needed by resteasy. Copy these into your /WEB-INF/lib directory. Place your JAX-RS annotated class resources and providers within one or more jars within /WEB-INF/lib or your raw class files within /WEB-INF/classes.

RESTeasy is implemented as a Servlet and deployed within a WAR file. If you open up the WEB-INF/web.xml in one of the example projects of your RESTeasy download you will see this:

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

   <servlet>
      <servlet-name>Resteasy</servlet-name>
      <servlet-class>
         org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
      </servlet-class>
      <init-param>
         <param-name>javax.ws.rs.Application</param-name>
         <param-value>com.restfully.shop.services.ShoppingApplication</param-value>
      </init-param>
   </servlet>

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

```
</web-app>
```

The Resteasy servlet is responsible for initializing some basic components of RESTeasy.

## 3.2. Configuration Switches

Resteasy receives configuration options from <context-param> elements.

**Table 3.1.**

| Option Name | Default Value | Description |
|---|---|---|
| resteasy.servlet.mapping.prefix | no default | If the url-pattern for the Resteasy servlet-mapping is not /* |
| resteasy.scan | false | Automatically scan WEB-INF/lib jars and WEB-INF/classes directory for both @Provider and JAX-RS resource classes (@Path, @GET, @POST etc..) and register them |
| resteasy.scan.providers | false | Scan for @Provider classes and register them |
| resteasy.scan.resources | false | Scan for JAX-RS resource classes |
| resteasy.providers | no default | A comma delimited list of fully qualified @Provider class names you want to register |
| resteasy.use.builtin.providers | true | Whether or not to register default, built-in @Provider classes. (Only available in 1.0-beta-5 and later) |
| resteasy.resources | no default | A comma delimited list of fully qualified JAX-RS resource class names you want to register |
| resteasy.jndi.resources | no default | A comma delimited list of JNDI names which reference objects you want to register as JAX-RS resources |
| javax.ws.rs.Application | no default | |

| Option Name | Default Value | Description |
|---|---|---|
| | | Fully qualified name of Application class to bootstrap in a spec portable way |
| resteasy.media.type.mappings | no default | Replaces the need for an Accept header by mapping file name extensions (like .xml or .txt) to a media type. Used when the client is unable to use a Accept header to choose a representation (i.e. a browser). See JAX-RS Content Negotiation chapter for more details. |
| resteasy.language.mappings | no default | Replaces the need for an Accept-Language header by mapping file name extensions (like .en or .fr) to a language. Used when the client is unable to use a Accept-Language header to choose a language (i.e. a browser). See JAX-RS Content Negotiation chapter for more details |

The resteasy.servlet.mapping.prefix <context param> variable must be set if your 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>
```

## 3.3. javax.ws.rs.core.Application

The javax.ws.rs.core.Application class is a standard JAX-RS class that you may implement to provide information on your deployment. It is simply a class the 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 a servlet init-param, javax.ws.rs.Application with a fully qualified class that implements Application. For example:

```
  <servlet>
      <servlet-name>Resteasy</servlet-name>
      <servlet-class>
          org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
      </servlet-class>
      <init-param>
          <param-name>javax.ws.rs.Application</param-name>
          <param-value>com.restfully.shop.services.ShoppingApplication</param-value>
      </init-param>
  </servlet>
```

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

## 3.4. RESTEasy as a ServletContextListener

The initialization of RESTEasy can be performed within a ServletContextListener instead of within the Servlet. You may need this if you are writing custom Listeners that need to interact with RESTEasy at boot time. An example of this is the RESTEasy Spring integration that requires a Spring ServletContextListener. The org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap class is a ServletContextListener that configures an instance of an ResteasyProviderFactory

and Registry. You can obtain instances of a ResteasyProviderFactory and Registry from the ServletContext attributes org.jboss.resteasy.spi.ResteasyProviderFactory and org.jboss.resteasy.spi.Registry. From these instances you can programmatically interact with RESTEasy registration interfaces.

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

  <!-- ** INSERT YOUR LISTENERS HERE!!!! -->

  <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>/resteasy/*</url-pattern>
  </servlet-mapping>

</web-app>
```

## 3.5. RESTEasy as a servlet Filter

The downside of running Resteasy as a Servlet is that you cannot have static resources like .html and .jpeg files in the same path as your JAX-RS services. Resteasy allows you to run as a Filter instead. If a JAX-RS resource is not found under the URL requested, Resteasy will delegate back to the base servlet container to resolve URLs.

```
<web-app>
```

```
<filter>
    <filter-name>Resteasy</filter-name>
    <filter-class>
        org.jboss.resteasy.plugins.server.servlet.FilterDispatcher
    </filter-class>
    <init-param>
        <param-name>javax.ws.rs.Application</param-name>
        <param-value>com.restfully.shop.services.ShoppingApplication</param-value>
    </init-param>
</filter>

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

</web-app>
```

## 3.6. Install/Config in JBoss 6-M4 and Higher

RESTEasy is preconfigured and completely integrated with JBoss 6-M4 and higher. You can use it with EJB and CDI and you can rely completely on JBoss for scanning for your JAX-RS services and deploying them. All you have to provide is your JAX-RS service classes packaged within a WAR either as POJOs, CDI beans, or EJBs and provide an empty web.xml file as follows:

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/
web-app_3_0.xsd">
</web-app>
```

## 3.7. RESTEasyLogging

RESTEasy logs various events using slf4j.

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

The logging categories are still a work in progress, but the initial set should make it easier to trouleshoot issues. Currently, the framework has defined the following log categories:

**Table 3.2.**

| 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 |
| 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're developing RESTEasy code, the LoggerCategories class provide easy access to category names and provides easy access to the various loggers.

# 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 {...}

}
```

Let's say 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 and/or a resource method. If it exists on both the class and method, the relative path to the resource method is a concatenation of the class and method.

In the @javax.ws.rs package there are annotations for each HTTP method. @GET, @POST, @PUT, @DELETE, and @HEAD. You place these on public methods that you want to map to that certain kind of HTTP method. As long as there is a @Path annotation on the class, you do not have to have a @Path annotation on the method you are mapping. You can have more than one HTTP method as long as they can be distinguished from other methods.

When you have a @Path annotation on a method without an HTTP method, these are called JAXRSResourceLocators.

## 4.1. @Path and regular expression mappings

The @Path annotation is not limited to simple path expressions. You also have the ability to insert regular expressions into @Path's value. 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 ] "}"
```

The regular-expression part is optional. When the expression is not provided, it defaults to a wildcard matching of one particular segment. In regular-expression terms, the expression defaults to

```
"([^/]*)"                                                                    15
```

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
```

# @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
  }
}
```

What this allows you to do is embed variable identification within the URIs of your resources. In the above example, an isbn URI parameter is used to pass information about the book we want to access. The parameter type you inject into can be any primitive type, a String, or any Java object that has a constructor that takes a String parameter, or a static valueOf method that takes a String as a parameter. For example, lets say we wanted isbn to be a real object. We could do:

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



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

Or instead of a public String constructors, have a valueOf method:

```
  public class ISBN {

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

```
    }
```

# 5.1. Advanced @PathParam and Regular Expressions

There are a few more complicated uses of @PathParams not discussed in the previous section.

You are allowed to specify one or more path params embedded in one URI segment. Here are some examples:

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

So, a URI of "/aaa111bbb" would match #1. "/bill-02115" would match #2. "foobill-02115bar" would match #3.

We discussed before how you can use regular expression patterns within @Path values.

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

For the following requests, lets see what the values of the "param" and "many" @PathParams would be:

**Table 5.1.**

| Request | param | many |
|---|---|---|
| GET /aaabb/some/stuff | bb | some |
| GET /aaab/a/lot/of/stuff | b | a/lot/of |

# 5.2. @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();

}
```

You can have Resteasy inject a PathSegment instead of a value with your @PathParam.

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

This is very useful if you have a bunch of @PathParams that use matrix parameters. The idea of matrix parameters is that they are an arbitrary set of name-value pairs embedded in a uri path segment. The PathSegment object gives you access to theese parameters. See also MatrixParam.

A matrix parameter example is:

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

The basic idea of matrix parameters is that it represents resources that are addressable by their attributes as well as their raw id.

# @**QueryParam**

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

GET /books?num=5

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

Currently since Resteasy is built on top of a Servlet, it does not distinguish between URI query strings or url form encoded paramters. Like PathParam, your parameter type can be an String, primitive, or class that has a String constructor or static valueOf() method.

# @HeaderParam

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

GET /books?num=5

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

Like PathParam, your parameter type can be an String, primitive, or class that has a String constructor or static valueOf() method. For example, MediaType has a valueOf() method and you could do:

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

# Linking resources

There are two mechanisms available in RESTEasy to link a resource to another, and to link resources to operations: the Link HTTP header, and Atom links inside the resource representations.

## 8.1. Link Headers

RESTEasy has both client and server side support for the *Link header specification* [http://tools.ietf.org/html/draft-nottingham-http-link-header-06]. See the javadocs for org.jboss.resteasy.spi.LinkHeader, org.jboss.resteasy.spi.Link, and org.jboss.resteasy.client.ClientResponse.

The main advantage of Link headers over Atom links in the resource is that those links are available without parsing the entity body.

## 8.2. Atom links in the resource representations

RESTEasy allows you to inject *Atom links* [http://tools.ietf.org/html/rfc4287#section-4.2.7] directly inside the entity objects you are sending to the client, via auto-discovery.

> **Warning**
>
> This is only available when using the Jettison or JAXB providers (for JSON and XML).

The main advantage over Link headers is that you can have any number of Atom links directly over the concerned resources, for any number of resources in the response. For example, you can have Atom links for the root response entity, and also for each of its children entities.

### 8.2.1. Configuration

There is no configuration required to be able to inject Atom links in your resource representation, you just have to have this maven artifact in your path:

**Table 8.1. Maven artifact for Atom link injection**

| Group | Artifact | Version |
|---|---|---|
| org.jboss.resteasy | resteasy-links | 2.0-beta-2 |

### 8.2.2. Your first links injected

You need three things in order to tell RESTEasy to inject Atom links in your entities:

- Annotate the JAX-RS method with `@AddLinks` to indicate that you want Atom links injected in your response entity.

- Add `RESTServiceDiscovery` fields to the resource classes where you want Atom links injected.

- Annotate the JAX-RS methods you want Atom links for with `@LinkResource`, so that RESTEasy knows which links to create for which resources.

The following example illustrates how you would declare everything in order to get the Atom links injected in your book store:

```
@Path("/")
@Consumes({"application/xml", "application/json"})
@Produces({"application/xml", "application/json"})
public interface BookStore {

 @AddLinks
 @LinkResource(value = Book.class)
 @GET
 @Path("books")
 public Collection<Book> getBooks();

 @LinkResource
 @POST
 @Path("books")
 public void addBook(Book book);

 @AddLinks
 @LinkResource
 @GET
 @Path("book/{id}")
 public Book getBook(@PathParam("id") String id);

 @LinkResource
 @PUT
 @Path("book/{id}")
 public void updateBook(@PathParam("id") String id, Book book);

 @LinkResource(value = Book.class)
 @DELETE
 @Path("book/{id}")
 public void deleteBook(@PathParam("id") String id);
}
```

And this is the definition of the Book resource:

```
@Mapped(namespaceMap  =  @XmlNsMap(jsonName  =  "atom",  namespace  =  "http://
www.w3.org/2005/Atom"))
@XmlRootElement
@XmlAccessorType(XmlAccessType.NONE)
public class Book {
 @XmlAttribute
 private String author;

 @XmlID
 @XmlAttribute
 private String title;

 @XmlElementRef
 private RESTServiceDiscovery rest;
}
```

If you do a GET /order/foo you will then get this XML representation:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<book xmlns:atom="http://www.w3.org/2005/Atom" title="foo" author="bar">
 <atom:link href="http://localhost:8081/books" rel="list"/>
 <atom:link href="http://localhost:8081/books" rel="add"/>
 <atom:link href="http://localhost:8081/book/foo" rel="self"/>
 <atom:link href="http://localhost:8081/book/foo" rel="update"/>
 <atom:link href="http://localhost:8081/book/foo" rel="remove"/>
</book>
```

And in JSON format:

```
{
 "book":
 {
  "@title":"foo",
  "@author":"bar",
  "atom.link":
   [
    {"@href":"http://localhost:8081/books","@rel":"list"},
    {"@href":"http://localhost:8081/books","@rel":"add"},
    {"@href":"http://localhost:8081/book/foo","@rel":"self"},
```

```
   {"@href":"http://localhost:8081/book/foo","@rel":"update"},
   {"@href":"http://localhost:8081/book/foo","@rel":"remove"}
  ]
 }
}
```

## 8.2.3. Customising how the Atom links are serialised

Because the `RESTServiceDiscovery` is in fact a JAXB type which inherits from `List` you are free to annotate it as you want to customise the JAXB serialisation, or just rely on the default with `@XmlElementRef`.

## 8.2.4. Specifying which JAX-RS methods are tied to which resources

This is all done by annotating the methods with the `@LinkResource` annotation. It supports the following optional parameters:

**Table 8.2.**

**`@LinkResource` parameters**

| Parameter | Type | Function | Default |
|---|---|---|---|
| value | Class | Declares an Atom link for the given type of resources. | Defaults to the entity body type (non-annotated parameter), or the method's return type. This default does not work with `Response` or `Collection` types, they need to be explicitly specified. |
| rel | String | The Atom link relation | list<br>For `GET` methods returning a `Collection`<br><br>self<br>For `GET` methods returning a non-`Collection` |

| Parameter | Type | Function | Default |
|---|---|---|---|
| | | | remove<br>    For `DELETE`<br>    methods |
| | | | update<br>    For `PUT` methods |
| | | | add<br>    For `POST` methods |

You can add several `@LinkResource` annotations on a single method by enclosing them in a `@LinkResources` annotation. This way you can add links to the same method on several resource types. For example the `/order/foo/comments` operation can belongs on the `Order` resource with the `comments` relation, and on the `Comment` resource with the `list` relation.

## 8.2.5. Specifying path parameter values for URI templates

When RESTEasy adds links to your resources it needs to insert the right values in the URI remplate. This is done either automatically by guessing the list of values from the entity, or by specifying the values in the `@LinkResource pathParameters` parameter.

### 8.2.5.1. Loading URI template values from the entity

URI template values are extracted from the entity from fields or Java Bean properties annotated with `@ResourceID`, JAXB's `@XmlID` or JPA's `@Id`. If there are more than one URI template value to find in a given entity, you can annotate your entity with `@ResourceIDs` to list the names of fields or properties that make up this entity's Id. If there are other URI template values required from a parent entity, we try to find that parent in a field or Java Bean property annotated with `@ParentResource`. The list of URI template values extracted up every `@ParentResource` is then reversed and used as the list of values for the URI template.

For example, let's consider the previous Book example, and a list of comments:

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.NONE)
public class Comment {
 @ParentResource
 private Book book;

 @XmlElement
 private String author;

 @XmlID
 @XmlAttribute
```

```
 private String id;

 @XmlElementRef
 private RESTServiceDiscovery rest;
}
```

Given the previous book store service augmented with comments:

```
@Path("/")
@Consumes({"application/xml", "application/json"})
@Produces({"application/xml", "application/json"})
public interface BookStore {

 @AddLinks
 @LinkResources({
  @LinkResource(value = Book.class, rel = "comments"),
  @LinkResource(value = Comment.class)
 })
 @GET
 @Path("book/{id}/comments")
 public Collection<Comment> getComments(@PathParam("id") String bookId);

 @AddLinks
 @LinkResource
 @GET
 @Path("book/{id}/comment/{cid}")
 public Comment getComment(@PathParam("id") String bookId, @PathParam("cid") String
commentId);

 @LinkResource
 @POST
 @Path("book/{id}/comments")
 public void addComment(@PathParam("id") String bookId, Comment comment);

 @LinkResource
 @PUT
 @Path("book/{id}/comment/{cid}")
 public void updateComment(@PathParam("id") String bookId, @PathParam("cid") String
commentId, Comment comment);

 @LinkResource(Comment.class)
 @DELETE
 @Path("book/{id}/comment/{cid}")
```

```
    public void deleteComment(@PathParam("id") String bookId, @PathParam("cid") String
commentId);


}
```

Whenever we need to make links for a `Book` entity, we look up the ID in the `Book`'s `@XmlID` property.
Whenever we make links for `Comment` entities, we have a list of values taken from the `Comment`'s
`@XmlID` and its `@ParentResource`: the `Book` and its `@XmlID`.

For a `Comment` with `id "1"` on a `Book` with `title "foo"` we will therefore get a list of URI template
values of `{"foo", "1"}`, to be replaced in the URI template, thus obtaining either `"/book/foo/
comments"` or `"/book/foo/comment/1"`.

### 8.2.5.2. Specifying path parameters manually

If you do not want to annotate your entities with resource ID annotations (`@ResourceID`,
`@ResourceIDs`, `@XmlID` or `@Id`) and `@ParentResource`, you can also specify the URI template
values inside the `@LinkResource` annotation, using Unified Expression Language expressions:

**Table 8.3.**

**`@LinkResource` URI template parameter**

| Parameter | Type | Function | Default |
|---|---|---|---|
| pathParameters | `String[]` | Declares a list of UEL expressions to obtain the URI template values. | Defaults to using `@ResourceID`, `@ResourceIDs`, `@XmlID` or `@Id` and `@ParentResource` annotations to extract the values from the model. |

The UEL expressions are evaluated in the context of the entity, which means that any unqualified
variable will be taken as a property for the entity itself, with the special variable `this` bound to
the entity we're generating links for.

The previous example of `Comment` service could be declared as such:

```
@Path("/")
@Consumes({"application/xml", "application/json"})
@Produces({"application/xml", "application/json"})
public interface BookStore {

 @AddLinks
```

```
@LinkResources({
 @LinkResource(value = Book.class, rel = "comments", pathParameters = "${title}"),
 @LinkResource(value = Comment.class, pathParameters = {"${book.title}", "${id}"})
})
@GET
@Path("book/{id}/comments")
public Collection<Comment> getComments(@PathParam("id") String bookId);


@AddLinks
@LinkResource(pathParameters = {"${book.title}", "${id}"})
@GET
@Path("book/{id}/comment/{cid}")
 public Comment getComment(@PathParam("id") String bookId, @PathParam("cid") String
commentId);


@LinkResource(pathParameters = {"${book.title}", "${id}"})
@POST
@Path("book/{id}/comments")
public void addComment(@PathParam("id") String bookId, Comment comment);


@LinkResource(pathParameters = {"${book.title}", "${id}"})
@PUT
@Path("book/{id}/comment/{cid}")
 public void updateComment(@PathParam("id") String bookId, @PathParam("cid") String
commentId, Comment comment);


@LinkResource(Comment.class, pathParameters = {"${book.title}", "${id}"})
@DELETE
@Path("book/{id}/comment/{cid}")
 public void deleteComment(@PathParam("id") String bookId, @PathParam("cid") String
commentId);

}
```

## 8.2.6. Securing entities

You can restrict which links are injected in the resource based on security restrictions for the client, so that if the current client doesn't have permission to delete a resource he will not be presented with the `"delete"` link relation.

Security restrictions can either be specified on the `@LinkResource` annotation, or using RESTEasy and EJB's security annotation `@RolesAllowed` on the JAX-RS method.

**Table 8.4.**

**`@LinkResource` security restrictions**

| Parameter | Type | Function | Default |
|-----------|------|----------|---------|
| constraint | String | A UEL expression which must evaluate to true to inject this method's link in the response entity. | Defaults to using `@RolesAllowed` from the JAX-RS method. |

## 8.2.7. Extending the UEL context

We've seen that both the URI template values and the security constraints of `@LinkResource` use UEL to evaluate expressions, and we provide a basic UEL context with access only to the entity we're injecting links in, and nothing more.

If you want to add more variables or functions in this context, you can by adding a `@LinkELProvider` annotation on the JAX-RS method, its class, or its package. This annotation's value should point to a class that implements the `ELProvider` interface, which wraps the default `ELContext` in order to add any missing functions.

For example, if you want to support the Seam annotation `s:hasPermission(target, permission)` in your security constraints, you can add a `package-info.java` file like this:

```
@LinkELProvider(SeamELProvider.class)
package org.jboss.resteasy.links.test;

import org.jboss.resteasy.links.*;
```

With the following provider implementation:

```
package org.jboss.resteasy.links.test;

import javax.el.ELContext;
import javax.el.ELResolver;
import javax.el.FunctionMapper;
import javax.el.VariableMapper;

import org.jboss.seam.el.SeamFunctionMapper;

import org.jboss.resteasy.links.ELProvider;

public class SeamELProvider implements ELProvider {

 public ELContext getContext(final ELContext ctx) {
```

```
  return new ELContext() {

  private SeamFunctionMapper functionMapper;

  @Override
  public ELResolver getELResolver() {
   return ctx.getELResolver();
  }

  @Override
  public FunctionMapper getFunctionMapper() {
   if (functionMapper == null)
    functionMapper = new SeamFunctionMapper(ctx
      .getFunctionMapper());
   return functionMapper;
  }

  @Override
  public VariableMapper getVariableMapper() {
   return ctx.getVariableMapper();
  }
 };
 }

}
```

And then use it as such:

```
@Path("/")
@Consumes({"application/xml", "application/json"})
@Produces({"application/xml", "application/json"})
public interface BookStore {

 @AddLinks
 @LinkResources({
  @LinkResource(value = Book.class, rel = "comments", constraint = "${s:hasPermission(this,
'add-comment')}"),
  @LinkResource(value = Comment.class, constraint = "${s:hasPermission(this, 'insert')}")
 })
 @GET
 @Path("book/{id}/comments")
 public Collection<Comment> getComments(@PathParam("id") String bookId);
```

```
@AddLinks
@LinkResource(constraint = "${s:hasPermission(this, 'read')}")
@GET
@Path("book/{id}/comment/{cid}")
 public Comment getComment(@PathParam("id") String bookId, @PathParam("cid") String
commentId);

@LinkResource(constraint = "${s:hasPermission(this, 'insert')}")
@POST
@Path("book/{id}/comments")
public void addComment(@PathParam("id") String bookId, Comment comment);

@LinkResource(constraint = "${s:hasPermission(this, 'update')}")
@PUT
@Path("book/{id}/comment/{cid}")
 public void updateComment(@PathParam("id") String bookId, @PathParam("cid") String
commentId, Comment comment);

@LinkResource(Comment.class, constraint = "${s:hasPermission(this, 'delete')}")
@DELETE
@Path("book/{id}/comment/{cid}")
 public void deleteComment(@PathParam("id") String bookId, @PathParam("cid") String
commentId);

}
```

## 8.2.8. Resource facades

Sometimes it is useful to add resources which are just containers or layers on other resources. For example if you want to represent a collection of `Comment` with a start index and a certain number of entries, in order to implement paging. Such a collection is not really an entity in your model, but it should obtain the `"add"` and `"list"` link relations for the `Comment` entity.

This is possible using resource facades. A resource facade is a resource which implements the `ResourceFacade<T>` interface for the type `T`, and as such, should receive all links for that type.

Since in most cases the instance of the `T` type is not directly available in the resource facade, we need another way to extract its URI template values, and this is done by calling the resource facade's `pathParameters()` method to obtain a map of URI template values by name. This map will be used to fill in the URI template values for any link generated for `T`, if there are enough values in the map.

Here is an example of such a resource facade for a collection of `Comment`s:

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.NONE)
public class ScrollableCollection implements ResourceFacade<Comment> {

 private String bookId;
 @XmlAttribute
 private int start;
 @XmlAttribute
 private int totalRecords;
 @XmlElement
 private List<Comment> comments = new ArrayList<Comment>();
 @XmlElementRef
 private RESTServiceDiscovery rest;

 public Class<Comment> facadeFor() {
  return Comment.class;
 }

 public Map<String, ? extends Object> pathParameters() {
  HashMap<String, String> map = new HashMap<String, String>();
  map.put("id", bookId);
  return map;
 }
}
```

This will produce such an XML collection:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<collection xmlns:atom="http://www.w3.org/2005/Atom" totalRecords="2" start="0">
 <atom.link href="http://localhost:8081/book/foo/comments" rel="add"/>
 <atom.link href="http://localhost:8081/book/foo/comments" rel="list"/>
 <comment xmlid="0">
  <text>great book</text>
  <atom.link href="http://localhost:8081/book/foo/comment/0" rel="self"/>
  <atom.link href="http://localhost:8081/book/foo/comment/0" rel="update"/>
  <atom.link href="http://localhost:8081/book/foo/comment/0" rel="remove"/>
  <atom.link href="http://localhost:8081/book/foo/comments" rel="add"/>
  <atom.link href="http://localhost:8081/book/foo/comments" rel="list"/>
 </comment>
 <comment xmlid="1">
  <text>terrible book</text>
```

```
    <atom.link href="http://localhost:8081/book/foo/comment/1" rel="self"/>
    <atom.link href="http://localhost:8081/book/foo/comment/1" rel="update"/>
    <atom.link href="http://localhost:8081/book/foo/comment/1" rel="remove"/>
    <atom.link href="http://localhost:8081/book/foo/comments" rel="add"/>
    <atom.link href="http://localhost:8081/book/foo/comments" rel="list"/>
  </comment>
</collection>
```

# @MatrixParam

The idea of matrix parameters is that they are an arbitrary set of name-value pairs embedded in a uri path segment. A matrix parameter example is:

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

The basic idea of matrix parameters is that it represents resources that are addressable by their attributes as well as their raw id. The @MatrixParam annotation allows you to inject URI matrix paramters into your method invocation

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

There is one big problem with @MatrixParam that the current version of the specification does not resolve. What if the same MatrixParam exists twice in different path segments? In this case, right now, its probably better to use PathParam combined with PathSegment.

# @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
publi cString getBooks(@CookieParam("sessionid") javax.ws.rs.core.Cookie id) {...}
```

Like PathParam, your parameter type can be an String, primitive, or class that has 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.

# @FormParam

When the input request body is of the type "application/x-www-form-urlencoded", a.k.a. 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 through that form, this is what the service might look like:

```
@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>. i.e. This is 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) {...}
```
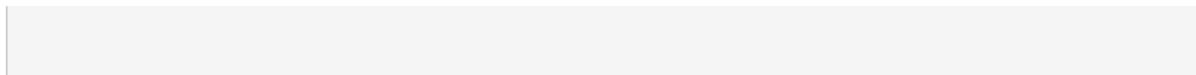
# @**Form**

This is a RESTEasy specific annotation that allows you to re-use any @*Param annotation within an injected class. RESTEasy will instantiate the class and inject values into any annotated @*Param or @Context property. This is useful if you have a lot of 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 somebody posts to /myservice, RESTEasy will instantiate an instance of MyForm and inject the form parameter "stuff" into the "stuff" field, the header "myheader" into the header field, and call the setFoo method with the path param variable of "foo".

# @DefaultValue

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

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

# @Encoded and encoding

JAX-RS allows you to get encoded or decoded @*Params 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 param. By default, inject @PathParam and @QueryParams are decoded. By additionally adding the @Encoded annotation, the value of these params will be provided in encoded form.

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

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

In the above example, the value of the @PathParam injected into the param of the get() method will be URL encoded. Adding the @Encoded annotation as a paramater annotation triggers this affect.

You may 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 the above example, the values of the "foo" query param and "param" path param 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. Controls whether the literal part of the supplied value (those characters that are not part of a template variable) are URL encoded. If true, any characters in the URI template that are not valid URI character will be automatically encoded. If false then all characters must be valid URI characters. By default this is set to true. If you want to encoded the characters yourself, you may.

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

Much like @Path.encode(), this controls whether the specified query param name should be encoded by the container before it tries to find the query param in the request.

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

# @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.

# JAX-RS Resource Locators and Sub Resources

Resource classes are able to partially process a request and provide another "sub" resource object that can 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 that have a @Path annotation, but no HTTP method are considered sub-resource locators. Their job is to provide an object that can process the request. In the above example ShoppingStore is a root resource because its class is annotated with @Path. The getCustomer() method is a sub-resource locator method.

If the client invoked:

```
GET /customer/123
```

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 to figure out how to dispatch the request. So, the ShoppingStore.getCustomer() method does not have to declare any specific type.

```java
@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 above example, getCustomer() returns a java.lang.Object. Per request, at runtime, the JAX-RS server will figure out how to dispatch the request based on the object returned by getCustomer(). What are the uses of this? Well, maybe you have a class hierarchy for your customers. Customer is the abstract base, CorporateCustomer and IndividualCustomer are subclasses. Your getCustomer() method might be doing a Hibernate polymorphic query and doesn't know, or care, what concrete class is it querying for, or what it returns.

```
@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() {...}

}
```

# JAX-RS Content Negotiation

The HTTP protocol has built in content negotiation headers that allow the client and server to specify what content they are transferring and what content they would prefer to get. 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 finds all methods that match the path, then, it sorts things based on the content-type header sent by the client. So, if a client sent:

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

thsi sis anice book
```

The stringBook() method would be invoked because it matches to the default "text/*" media type. Now, if the client instead sends XML:

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

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

The jaxbBook() method would be invoked.

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

```java
@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 list multiple media types that they support. The client's Accept header can also send multiple types it might like to receive. More specific media types are chosen first. The client Accept header or @Produces @Consumes can also specify weighted preferences that are used to match up requests with resource methods. This is best explained by RFC 2616 section 14.1 . Resteasy supports this complex way of doing content negotiation.

A variant in JAX-RS is a combination of media type, content-language, and content encoding as well as etags, last modified headers, and other preconditions. This is a more complex form of content negotiation that is done programmatically by the application developer using the javax.ws.rs.Variant, VarianListBuilder, and Request objects. Request is injected via @Context. Read the javadoc for more info on these.

# 17.1. URL-based negotiation

Some clients, like browsers, cannot use the Accept and Accept-Language headers to negotiation the representation's media type or language. RESTEasy allows you to map file name suffixes like (.xml, .txt, .en, .fr) to media types and languages. These file name suffixes take the place and override any Accept header sent by the client. You configure this using the resteasy.media.type.mappings and resteasy.language.mappings context-param variables within your web.xml

```
<web-app>
   <display-name>Archetype Created Web Application</display-name>
   <context-param>
      <param-name>resteasy.media.type.mappings</param-name>
      <param-value>html : text/html, json : application/json, xml : application/xml</param-value>
   </context-param>

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

   <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>
```

Mappings are a comma delimited list of suffix/mediatype or suffix/language mappings. Each mapping is delimited by a ':'. So, if you invoked GET /foo/bar.xml.en, this would be equivalent to invoking the following request:

```
GET /foo/bar
```

Accept: application/xml
Accept-Language: en-US

The mapped file suffixes are stripped from the target URL path before the request is dispatched to a corresponding JAX-RS resource.

# Content Marshalling/Providers

## 18.1. Default Providers and default JAX-RS Content Marshalling

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

**Table 18.1.**

| Media Types | Java Type |
|---|---|
| application/*+xml, text/*+xml, application/*+json, application/*+fastinfoset, application/atom+* | JaxB annotated classes |
| application/*+xml, text/*+xml | org.w3c.dom.Document |
| */* | java.lang.String |
| */* | java.io.InputStream |
| text/plain | primtives, 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 |

## 18.2. Content Marshalling with @Provider classes

The JAX-RS specification allows you to plug in your own request/response body reader and writers. To do this, you annotate a class with @Provider and specify the @Produces types for a writer and @Consumes types for a reader. You must also implement a MessageBodyReader/Writer interface respectively. Here is an example.

The Resteasy ServletContextLoader will automatically scan your WEB-INF/lib and classes directories for classes annotated with @Provider or you can manually configure them in web.xml. See Installation/Configuration

## 18.3. Providers Utility Class

javax.ws.rs.ext.Providers is a simple injectable interface that allows you to look up MessageBodyReaders, Writers, ContextResolvers, and ExceptionMappers. It is very useful, for

instance, for implementing multipart providers. Content types that embed other random 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&#47;* > *&#47;*)
 * 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&#47;* > *&#47;*)
 * 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);
}
```

A Providers instance is injectable into MessageBodyReader or Writers:

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

private @Context Providers providers;

...

}
```

# JAXB providers

As required by the specification, RESTEasy JAX-RS includes support for (un)marshalling JAXB annotated classes. RESTEasy provides multiple JAXB Providers to address some subtle differences between classes generated by XJC and classes which are simply annotated with @XmlRootElement, or working with JAXBElement classes directly.

For the most part, developers using the JAX-RS API, the selection of which provider is invoked will be completely transparent. For developers wishing to access the providers directly (which most folks won't need to do), this document describes which provider is best suited for different configurations.

A JAXB Provider is selected by RESTEasy when a parameter or return type is an object that is annotated with JAXB annotations (such as @XmlRootEntity or @XmlType) or if the type is a JAXBElement. Additionally, the resource class or resource method will be annotated with either a @Consumes or @Produces annotation and contain one or more of the following values:

- text/*+xml

- application/*+xml

- application/*+fastinfoset

- application/*+json

RESTEasy will select a different provider based on the return type or parameter type used in the resource. This section decribes how the selection process works.

@XmlRootEntity When a class is annotated with a @XmlRootElement annotation, RESTEasy will select the JAXBXmlRootElementProvider. This provider handles basic marhaling and and unmarshalling of custom JAXB entities.

@XmlType Classes which have been generated by XJC will most likely not contain an @XmlRootEntity annotation. In order for these classes to marshalled, they must be wrapped within a JAXBElement instance. This is typically accomplished by invoking a method on the class which serves as the XmlRegistry and is named ObjectFactory.

The JAXBXmlTypeProvider provider is selected when the class is annotated with an XmlType annotation and not an XmlRootElement annotation.

This provider simplifies this task by attempting to locate the XmlRegistry for the target class. By default, a JAXB implementation will create a class called ObjectFactory and is located in the same package as the target class. When this class is located, it will contain a "create" method that takes the object instance as a parameter. For example, of the target type is called "Contact", then the ObjectFactory class will have a method:

public JAXBElement createContact(Contact value) {..

JAXBElement<?> 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.

## 19.1. JAXB Decorators

Resteasy's JAXB providers have a pluggable way to decorate Marshaller and Unmarshaller instances. The way it works is that you can write an annotation that can trigger the decoration of a Marshaller or Unmarshaller. Your decorators can do things like set Marshaller or Unmarshaller properties, set up validation, stuff like that. Here's an example. Let's say we want to have an annotation that will trigger pretty-printing, nice formatting, of an XML document. If we were doing raw JAXB, we would set a property on the Marshaller of Marshaller.JAXB_FORMATTED_OUTPUT. Let's write a Marshaller decorator.

First we define a annotation:

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

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

To get this to work, we must annotate our @Pretty annotation with a meta-annotation called @Decorator. The target() attribute must be the JAXB Marshaller class. The processor() attribute is a class we will write next.

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

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

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

The processor implementation must implement the DecoratorProcessor interface and should also
be annotated with @DecorateTypes. This annotation specifies what media types the processor
can be used with. Now that we've defined our annotation and our Processor, we can use it on our
JAX-RS resource methods or JAXB types as follows:

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

If you are confused, check the Resteasy source code for the implementation of @XmlHeader

## 19.2. Pluggable JAXBContext's with ContextResolvers

You should not use this feature unless you know what you're doing.

Based on the class you are marshalling/unmarshalling, RESTEasy will, by default create
and cache JAXBContext instances per class type. If you do not want RESTEasy
to create JAXBContexts, 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 media type the context is meant for. You must also make sure to implement ContextResolver<JAXBContext>. This helps the runtime match to the correct context resolver. You must also annotate the ContextResolver class with @Provider.

There are multiple 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 Configuration Guide

4. Manually add it via R:esteasyProviderFactory.getInstance().registerProvider(Class) or registerProviderInstance(Object)

## 19.3. JAXB + XML provider

Resteasy is required to provide JAXB provider support for XML. It has a few extra annotations that can help code your app.

### 19.3.1. @XmlHeader and @Stylesheet

Sometimes when outputting XML documents you may want to set an XML header. Resteasy provides the @org.jboss.resteasy.annotations.providers.jaxb.XmlHeader annotation for this. 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("/header")
  @Produces("application/xml")
  @XmlHeader("<?xml-stylesheet type='text/xsl' href='${baseuri}foo.xsl' ?>")
  public Thing get()
  {
    Thing thing = new Thing();
    thing.setName("bill");
    return thing;
  }
}
```

The @XmlHeader here forces the XML output to have an xml-stylesheet header. This header could also have been put on the Thing class to get the same result. See the javadocs for more details on how you can use substitution values provided by resteasy.

Resteasy also has a convinience 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;
    }
}
```

## 19.4. JAXB + JSON provider

RESTEasy allows you to marshall JAXB annotated POJOs to and from JSON. This provider wraps the Jettison JSON library to accomplish this. You can obtain more information about Jettison and how it works from:

http://jettison.codehaus.org/

To use this integration with Jettision you need to import the resteasy-jettison-provider Maven module. Older versions of RESTEasy used to include this within the resteasy-jaxb-provider but we decided to modularize it more.

Jettison has two mapping formats. One is BadgerFish the other is a Jettison Mapped Convention format. The Mapped Convention is the default mapping.

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;
}
 }
```

This is how the JAXB Book class would be marshalled to JSON using the BadgerFish Convention

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

Notice that element values have a map associated with them and to get to the value of the element, you must access the "$" variable. Here's an example of accessing the book 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 on the JAXB class you are marshalling/unmarshalling, or, on the JAX-RS resource method or parameter:

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

If you are returning a book on the JAX-RS method and you don't want to (or can't) pollute your JAXB classes with RESTEasy annotations, add the annotation to the JAX-RS method:

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

If a Book is your input then you put it on the parameter:

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

The default Jettison Mapped Convention would return JSON that looked like this:

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

Notice that the @XmlAttribute "title" is prefixed with the '@' character. Unlike BadgerFish, the '$' does not represent the value of element text. This format is a bit simpler than the BadgerFish convention which is why it was chose as a default. Here's an example of accessing 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 allows you to fine tune the JAXB mapping using the @org.jboss.resteasy.annotations.providers.jaxb.json.Mapped annotation. You can provide an XML Namespace to JSON namespace mapping. For example, if you defined your JAXB namespace within your package-info.java class like this:

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

You would have to define a JSON to XML namespace mapping or you would receive an exception of something like this:

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

To fix this problem you need another annotation, @Mapped. You use the @Mapped annotation on your JAXB classes, on your JAX-RS resource method, or on the parameter you are unmarshalling

```
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() {...}
```

Besides mapping XML to JSON namespaces, you can also force @XmlAttribute's to be marshaled as XMLElements.

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

If you are returning a book on the JAX-RS method and you don't want to (or can't) pollute your JAXB classes with RESTEasy annotations, add the annotation to the JAX-RS method:

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

If a Book is your input then you put it on the parameter:

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

## 19.5. JAXB + FastinfoSet provider

RESTEasy supports the FastinfoSet mime type with JAXB annotated classes. Fast infoset documents are faster to serialize and parse, and smaller in size, than logically equivalent XML documents. Thus, fast infoset documents may be used whenever the size and processing time

of XML documents is an issue. It is configured the same way the XML JAXB provider is so really no other documentation is needed here.

To use this integration with Fastinfoset you need to import the resteasy-fastinfoset-provider Maven module. Older versions of RESTEasy used to include this within the resteasy-jaxb-provider but we decided to modularize it more.

## 19.6. Arrays and Collections of JAXB Objects

RESTEasy will automatically marshal arrays, java.util.Set's, and java.util.List's of JAXB objects to and from XML, JSON, Fastinfoset (or any other new JAXB mapper Restasy comes up with).

```java
@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 above resource can publish and receive JAXB objects. It is assumed that are wrapped in a collection element

```
 <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 this 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;
}
```

## 19.6.1. JSON and JAXB Collections/arrays

Resteasy supports using collections with JSON. It encloses lists, sets, or arrays of returned JAXB objects within 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;
  }
}
```

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

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

It also expects this format for input

## 19.7. Maps of JAXB Objects

RESTEasy will automatically marshal maps of JAXB objects to and from XML, JSON, Fastinfoset (or any other new JAXB mapper Restasy comes up with). Your parameter or method return type must be a generic with a String as the key and the JAXB object's type.

```java
@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;
```

```
  }
}
```

The above resource can publish and receive JAXB objects within a map. By default, they are wrapped in a "map" element in the default namespace. Also, 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, if we wanted to output this 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;
  }
```

## 19.7.1. JSON and JAXB maps

Resteasy supports using maps with JSON. It encloses maps returned JAXB objects within a simple JSON map. 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

## 19.7.2. 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 will screw up code that is dependent on the Jettison JAXB/JSon provider. If you had been using the Jettison JAXB/Json providers, you must either remove Jackson from your WEB-INF/lib or classpath, or use the @NoJackson annotation on your JAXB classes.

## 19.8. Interfaces, Abstract Classes, and JAXB

Some objects models use abstract classes and interfaces heavily. Unfortunately, JAXB doesn't work with interfaces that are root elements and RESTEasy can't unmarshal parameters that are interfaces or raw abstract classes because it doesn't have enough information 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, you would get an error from RESTEasy of something like "Cannot find a MessageBodyReader for...". This is because RESTEasy does not know that implementations of IFoo are JAXB classes and doesn't know how to create a JAXBContext for it. As a workaround, RESTEasy allows you to use the JAXB annotation @XmlSeeAlso on the interface to correct the problem. (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.

# Resteasy Atom Support

From W3.org (http://tools.ietf.org/html/rfc4287):

"Atom is an XML-based document format that describes lists of related information known as "feeds". Feeds are composed of a number of items, known as "entries", each with an extensible set of attached metadata. For example, each entry has a title. The primary use case that Atom addresses is the syndication of Web content such as weblogs and news headlines to Web sites as well as directly to user agents."

Atom is the next-gen RSS feed. Although it is used primarily for the syndication of blogs and news, many are starting to use this format as the envelope for Web Services, for example, distributed notifications, job queues, or simply a nice format for sending or receiving data in bulk from a service.

## 20.1. Resteasy Atom API and Provider

RESTEasy has defined a simple object model in Java to represent Atom and uses JAXB to marshal and unmarshal it. The main classes are in the org.jboss.resteasy.plugins.providers.atom package and are Feed, Entry, Content, and Link. If you look at the source, you'd see that these are annotated with JAXB annotations. The distribution contains the javadocs for this project and are a must to learn the model. Here is a simple example of sending an atom feed using the Resteasy API.

```java
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;
   }
}
```

Because Resteasy's atom provider is JAXB based, you are not limited to sending atom objects using XML. You can automatically re-use all the other JAXB providers that Resteasy has like JSON and fastinfoset. All you have to do is have "atom+" in front of the main subtype. i.e. @Produces("application/atom+json") or @Consumes("application/atom+fastinfoset")

## 20.2. Using JAXB with the Atom Provider

The org.jboss.resteasy.plugins.providers.atom.Content class allows you to unmarshal and marshal JAXB annotated objects that are the body of the content. Here's 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 is used to tell the content object you are sending back a Java JAXB object and want it marshalled appropriately. If you are using a different base format other than XML, i.e. "application/atom+json", this attached JAXB object will be marshalled into that same format.

If you have an atom document as your input, you can also extract JAXB objects from Content using the Content.getJAXBObject(Class clazz) method. Here is an example of an input atom document and 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);
  }
}
```

# Atom support through Apache Abdera

Resteasy provides support for Apache Abdera, an implementation of the Atom protocol and data format. http://incubator.apache.org/abdera/

Abdera is a full-fledged Atom server. Resteasy only supports integration with JAX-RS for Atom data format marshalling and unmarshalling to and from the Feed and Entry interface types in Abdera. Here's a simple example:

## 21.1. Abdera and Maven

The Abdera provider is not included with the Resteasy distribution. To include the Abdera provider in your WAR poms, include the following. Please change the version to be the version of resteasy you are working with. Also, Resteasy may be coded to pick up an older version of Abdera than what you want. You're on your own with fixing this one, sorry.

```
<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>
```

## 21.2. Using the Abdera Provider

```
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;
```

```java
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.createMarshaller().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.createUnmarshaller().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.createMarshaller().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.createUnmarshaller().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);
  }
}
```

# JSON Support via Jackson

Besides the Jettision JAXB adapter for JSON, Resteasy also support integration with the Jackson project. Many users find the output from Jackson much much nicer than the Badger format or Mapped format provided by Jettison. Jackson lives at http://jackson.codehaus.org. It allows you to easily marshal Java objects to and from JSON. It has a Java Bean based model as well as JAXB like APIs. Resteasy integrates with the JavaBean model as described at this url: http://jackson.codehaus.org/Tutorial.

While Jackson does come with its own JAX-RS integration. Resteasy expanded it a little. To include it within your project, just add this 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>2.0.1.GA</version>
</dependency>
```

The first extra piece that Resteasy added to the integration was to support "application/*+json". Jackson would only accept "application/json" and "text/json" as valid media types. This allows you to create json-based media types and still let Jackson marshal things for you. For example:

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

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

Another problem that occurs is when you are using the Resteasy JAXB providers alongside Jackson. You may want to use Jettision and JAXB to output your JSON instead of Jackson. In this case, you must either not install the Jackson provider, or use the annotation @org.jboss.resteasy.annotations.providers.NoJackson on your JAXB annotated classes. For example:

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

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

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

If you can't annotate the JAXB class with @NoJackson, then you can use the annotation on a method parameter. For example:

```
@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) {...}
    }
```

## 22.1. Possible Conflict With JAXB Provider

If your Jackson classes are annotated with JAXB annotations and you have the resteasy-jaxb-provider in your classpath, you may trigger the Jettision JAXB marshalling code. To turn off the JAXB json marshaller use the @org.jboss.resteasy.annotations.providers.jaxb.IgnoreMediaTypes("application/*+json") on your classes.

# Multipart Providers

Resteasy has rich support for the "multipart/*" and "multipart/form-data" mime types. The multipart mime format is used to pass lists of content bodies. Multiple content bodies are embedded in 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 is the same as 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 marshalling arbitrary List (for any multipart type) and Map (multipart/form-data only) objects

## 23.1. Input with multipart/mixed

When writing a JAX-RS service, RESTEasy provides an interface that allows you to read in 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();

   boolean isContentTypeFromMessage();
}
```

MultipartInput is a simple interface that allows you to get access to each part of the multipart message. Each part is represented by an InputPart interface. Each part has a set of headers associated with it You can unmarshall the part by calling one of the getBody() methods. The Type genericType parameter can be null, but the Class type parameter must be set. Resteasy will find a MessageBodyReader based on the media type of the part as well as the type information you pass in. The following piece of code is unmarshalling parts which are XML 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);
    }
  }
}
```

Sometimes you may want to unmarshall a body part that is sensitive to generic type metadata. In this case you can use the org.jboss.resteasy.util.GenericType class. Here's an example of unmarshalling a type that is sensitive to generic type metadata.

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

```
  }
```

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

## 23.2. java.util.List with multipart data

If your body parts are uniform, you do not have to manually unmarshall each and every part. You can just provide a java.util.List as your input parameter. It must have the type it is unmarshalling with the generic parameter of the List type declaration. Here's an example again of unmarshalling a list of customers.

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

## 23.3. Input with multipart/form-data

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

```java
public interface MultipartFormDataInput extends MultipartInput
{
  @Deprecated
  Map<String, InputPart> getFormData();

  Map<String, List<InputPart>> getFormDataMap();
```

```
  <T> T getFormDataPart(String key, Class<T> rawType, Type genericType) throws IOException;

  <T> T getFormDataPart(String key, GenericType<T> type) throws IOException;
}
```

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

## 23.4. java.util.Map with multipart/form-data

With form-data, if your body parts are uniform, you do not have to manually unmarshall each and every part. You can just provide a java.util.Map as your input parameter. It must have the type it is unmarshalling with the generic parameter of the List type declaration. Here's an example of of unmarshalling a Map of Customer objects which are JAXB annotated classes.

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

## 23.5. Input with multipart/related

When writing a JAX-RS service, RESTEasy provides an interface that allows you to read in multipart/related mime type. A multipart/related is used to indicate that message parts should not be considered individually but rather as parts of an aggregate whole. One example usage for multipart/related is to send a web page complete with images in a single message. Every multipart/related message has a root/start part that references the other parts of the message. The parts are identified by their "Content-ID" headers. multipart/related is defined by RFC 2387. The interface used 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 in much the same way as MultipartInput described earlier in this chapter.

## 23.6. 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()
}
```

When you want to output multipart data it is as simple as creating a MultipartOutput object and calling addPart() methods. Resteasy will automatically find a MessageBodyWriter to marshall your entity objects. Like MultipartInput, sometimes you may have marshalling which is sensitive to generic type metadata. In that case, use GenericType. Most of the time though passing in an Object and its MediaType is enough. In the example below, we are sending back a "multipart/mixed" format back to the calling client. The parts are Customer objects which are JAXB annotated and will be marshalling 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;
    }
```

## 23.7. Multipart Output with java.util.List

If your body parts are uniform, you do not have to manually marshall each and every part or even use a MultipartOutput object.. You can just provide a java.util.List. It must have the generic type it is marshalling with the generic parameter of the List type declaration. You must also annotate the method with the @PartType annotation to specify what media type each part is. Here's an example of sending back a list of customers back to a client. The customers are JAXB objects

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

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

## 23.8. 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()
}
```

When you want to output multipart/form-data it is as simple as creating a MultipartFormDataOutput object and calling addFormData() methods. Resteasy will automatically find a MessageBodyWriter to marshall your entity objects. Like MultipartInput, sometimes you may have marshalling which is sensitive to generic type metadata. In that case, use GenericType. Most of the time though passing in an Object and its MediaType is enough. In the example below, we are sending back a "multipart/form-data" format back to the calling client. The parts are Customer objects which are JAXB annotated and will be marshalling 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;
  }
```

## 23.9. Multipart FormData Output with java.util.Map

If your body parts are uniform, you do not have to manually marshall each and every part or even use a MultipartFormDataOutput object.. You can just provide a java.util.Map. It must have the generic type it is marshalling with the generic parameter of the Map type declaration. You must also annotate the method with the @PartType annotation to specify what media type each part is. Here's an example of sending back a list of customers back to a client. The customers are JAXB objects

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

## 23.10. 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)
}
```

When you want to output multipart/related it is as simple as creating a MultipartRelatedOutput object and calling addPart() methods. The first added part will be used as the root part of the multipart/related message. Resteasy will automatically find a MessageBodyWriter to marshall your entity objects. Like MultipartInput, sometimes you may have marshalling which is sensitive to generic type metadata. In that case, use GenericType. Most of the time though passing in an Object and its MediaType is enough. In the example below, we are sending back a "multipart/related" format back to the calling client. We are sending a html with 2 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;
  }
}
```

# 23.11. @MultipartForm and POJOs

If you have a exact knowledge of your multipart/form-data packets, you can map them to and from a POJO class to and from multipart/form-data using the @org.jboss.resteasy.annotations.providers.multipart.MultipartForm annotation and the JAX-RS @FormParam annotation. You simple define a POJO with at least a default constructor and annotate its fields and/or properties with @FormParams. These @FormParams must also be annotated with @org.jboss.resteasy.annotations.providers.multipart.PartType if you are doing output. For example:

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

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

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

After defining your POJO class you can then use it to represent multipart/form-data. Here's an example of sending 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);
}
```

You see that the @MultipartForm annotation was used to tell RESTEasy that the object has @FormParam and that it should be marshalled from that. You can also use the same object to receive multipart data. Here is an example of the server side counterpart of our customer portal.

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

  @Path("issues/{id})
  @Consumes("multipart/form-data")
  @PUT
  public void putIssue(@MultipartForm CustoemrProblemForm,
              @PathParam("id") int id) {
    ... write to database...
  }
}
```

## 23.12. XML-binary Optimized Packaging (Xop)

RESTEasy supports Xop messages packaged as multipart/related. What does this mean? If you have a JAXB annotated POJO that also holds some binary content you may choose to send it in such a way where the binary does not need to be encoded in any way (neither base64 neither

hex). This results in faster transport while still using the convenient POJO. More about Xop can be read here: *http://www.w3.org/TR/xop10/.* Now lets see an example:

First we have a JAXB annotated POJO to work with. @XmlMimeType tells JAXB the mime type of the binary content (its not required to do XOP packaging but it is recommended to be set if you know the exact type):

```
@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 ...
}
```

In the above POJO myBinary and myDataHandler will be processed as binary attachments while the whole Xop object will be sent as xml (in the places of the binaries only their references will be generated). javax.activation.DataHandler is the most general supported type so if you need an java.io.InputStream or a javax.activation.DataSource you need to go with the DataHandler. Some other special types are supported too: java.awt.Image and javax.xml.transform.Source. Let's assume that Customer is also JAXB friendly POJO in the above example (of course it can also have binary parts). Now lets see a an example Java client that sends this:

```
// 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);
}
```

We used @Consumes(MediaType.MULTIPART_RELATED) to tell RESTEasy that we want to
send multipart/related packages (thats the container format that will hold our Xop message). We
used @XopWithMultipartRelated to tell RESTEasy that we want to make Xop messages. So we
have a POJO and a client service that is willing to send it. All we need now a server that can read it:

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

We used @Consumes(MediaType.MULTIPART_RELATED) to tell RESTEasy that we want to
read multipart/related packages. We used @XopWithMultipartRelated to tell RESTEasy that we
want to read Xop messages. Of course we could also produce Xop return values but we would
than also need to annotate that and use a Produce annotation, too.

## 23.13. Note about multipart parsing and working with other frameworks

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

# 23.14. Overwriting the default fallback content type for multipart messages

By default if no Content-Type header is present in a part, "text/plain; charset=us-ascii" is used as fallback. This is the value defined by the MIME RFC. However for example some web clients (like most, if not all, web browsers) do not send Content-Type headers for all fields in a multipart/form-data request (only for the file parts). This can cause character encoding and unmarshalling errors on the server side. To correct this there is an option to define an other, non-rfc compliant fallback value. This can be done dynamicly per request with the PreProcessInterceptor infrastructure of RESTEasy. In the following example we will set "*/*; charset=UTF-8" as the new default fallback:

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

@Provider
@ServerInterceptor
public class ContentTypeSetterPreProcessorInterceptor implements
  PreProcessInterceptor {

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

}
```

# YAML Provider

Since Beta 6, resteasy comes with built in support for YAML using the Jyaml library. To enable YAML support, you need to drop in the jyaml-1.3.jar in RestEASY's classpath.

Jyaml jar file can either be downloaded from sourceforge: https://sourceforge.net/project/showfiles.php?group_id=153924

Or if you use maven, the jyaml jar is available through the main repositories and included using this dependency:

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

When starting resteasy look out in the logs for a line stating that the YamlProvider has been added - this indicates that resteasy has found the Jyaml jar:

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

The Yaml provider recognises three mime types:

- text/x-yaml

- text/yaml

- application/x-yaml

This is an example of how to use Yaml in a resource method.

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

# String marshalling for String based @*Param

@PathParam, @QueryParam, @MatrixParam, @FormParam, and @HeaderParam are represented as strings in a raw HTTP request. The specification says that these types of injected parameters can be converted to objects if these objects have a valueOf(String) static method or a constructor that takes one String parameter. What if you have a class where valueOf() or this string constructor doesn't exist or is inappropriate for an HTTP request? Resteasy has 2 proprietary @Provider interfaces that you can plug in:

## 25.1. StringConverter

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

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

You implement this interface to provide your own custom string marshalling. It is registered within your web.xml under the resteasy.providers context-param (See Installation and Configuration chapter). You can do it manually by calling the ResteasyProviderFactory.addStringConverter() method. Here's a simple example of using a StringConverter:

```
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
  {
```

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

## 25.2. StringParamUnmarshaller

org.jboss.resteasy.spi.StringParameterUnmarshaller is sensative to the annotations placed on the parameter or field you are injecting into. It is created per injector. The setAnnotations() method is called by resteasy to initialize the unmarshaller.

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

public interface StringParameterUnmarshaller<T>
{
   void setAnnotations(Annotation[] annotations);
   T fromString(String str);
}
```

You can add this by creating and registering a provider that implements this interface. You can also bind them using a meta-annotation called org.jboss.resteasy.annotationsStringParameterUnmarshallerBinder. Here's an example of formatting a java.util.Date based @PathParam

```java
public class StringParamUnmarshallerTest extends BaseResourceTest
{
   @Retention(RetentionPolicy.RUNTIME)
   @StringParameterUnmarshallerBinder(DateFormatter.class)
   public @interface DateFormat
   {
     String value();
   }

   public static class DateFormatter implements StringParameterUnmarshaller<Date>
   {
     private SimpleDateFormat formatter;

     public void setAnnotations(Annotation[] annotations)
     {
       DateFormat format = FindAnnotation.findAnnotation(annotations, DateFormat.class);
       formatter = new SimpleDateFormat(format.value());
     }
```

```java
    public Date fromString(String str)
    {
      try
      {
        return formatter.parse(str);
      }
      catch (ParseException e)
      {
        throw new RuntimeException(e);
      }
    }
  }

  @Path("/datetest")
  public static class Service
  {
    @GET
    @Produces("text/plain")
    @Path("/{date}")
    public String get(@PathParam("date") @DateFormat("MM-dd-yyyy") Date date)
    {
      System.out.println(date);
      Calendar c = Calendar.getInstance();
      c.setTime(date);
      Assert.assertEquals(3, c.get(Calendar.MONTH));
      Assert.assertEquals(23, c.get(Calendar.DAY_OF_MONTH));
      Assert.assertEquals(1977, c.get(Calendar.YEAR));
      return date.toString();
    }
  }

  @BeforeClass
  public static void setup() throws Exception
  {
    addPerRequestResource(Service.class);
  }

  @Test
  public void testMe() throws Exception
  {
    ClientRequest request = new ClientRequest(generateURL("/datetest/04-23-1977"));
    System.out.println(request.getTarget(String.class));
  }
}
```

In the example a new annotation is defined called @DateFormat. This annotation class is annotated with the meta-annotation StringParameterUnmarshallerBinder with a reference to the DateFormmater classes.

The Service.get() method has a @PathParam parameter that is also annotated with @DateFormat. The application of @DateFormat triggers the binding of the DateFormatter. The DateFormatter will now be run to unmarshal the path parameter into the date paramter of the get() method.

# Responses using javax.ws.rs.core.Response

You can build custom responses using the javax.ws.rs.core.Response and ResponseBuilder classes. If you want to do your own streaming, your entity response must be an implementation of javax.ws.rs.core.StreamingOutput. See the java doc for more information.

# Exception Handling

## 27.1. Exception Mappers

ExceptionMappers are custom, application provided, components that can catch thrown application exceptions and write specific HTTP responses. The are classes annotated with @Provider and that implement this 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 exception is thrown it will be caught by the JAX-RS runtime. JAX-RS will then scan registered ExceptionMappers to see which one support marshalling the exception type thrown. Here is an example of ExceptionMapper

```
@Provider
```

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

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


}
```

You register ExceptionMappers the same way you do MessageBodyReader/Writers. By scanning, through the resteasy provider context-param (if you're deploying via a WAR file), or programmatically through the ResteasyProviderFactory class.

## 27.2. Resteasy Built-in Internally-Thrown Exceptions

Resteasy has a set of built-in exceptions that are thrown by it when it encounters errors during dispatching or marshalling. They all revolve around specific HTTP error codes. You can find them in RESTEasy's javadoc under the package org.jboss.resteasy.spi. Here's a list of them:

**Table 27.1.**

| Exception | HTTP Code | Description |
| --- | --- | --- |
| BadRequestException | 400 | Bad Request. Request wasn't formatted correctly or problem processing request input. |
| UnauthorizedException | 401 | Unauthorized. Security exception thrown if you're using Resteasy's simple annotation-based role-based security |
| InternalServerErrorException | 500 | Internal Server Error. |
| MethodNotAllowedException | 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 | |

| Exception | HTTP Code | Description |
| --- | --- | --- |
| | | Not Found. There is no JAX-RS method that serves the request path/resource. |
| ReaderException | 400 | All exceptions thrown from MessageBodyReaders are wrapped within this exception. If there is no ExceptionMapper for the wrapped exception or if the excepiton isn't a WebApplicationException, then resteasy will return a 400 code by default. |
| WriterException | 500 | All exceptions thrown from MessageBodyWriters are wrapped within this exception. If there is no ExceptionMapper for the wrapped exception or if the exception isn't a WebApplicationException, then resteasy will return a 400 code by default. |
| o.j.r.plugins.providers.jaxb.JAXBUnmarshalException | 400 | The JAXB providers (XML and Jettison) throw this exception on reads. They may be wrapping JAXBExceptions. This class extends ReaderException |
| o.j.r.plugins.providers.jaxb.JAXBMarshalException | 500 | The JAXB providers (XML and Jettison) throw this exception on writes. They may be wrapping JAXBExceptions. This class extends WriterException |
| ApplicationException | N/A | This exception wraps all exceptions thrown from application code. It functions much in the same way as InvocationTargetException. If there is an ExceptionMapper for wrapped exception, then |

| Exception | HTTP Code | Description |
| --- | --- | --- |
| | | that is used to handle the request. |
| Failure | N/A | Internal Resteasy. Not logged |
| LoggableFailure | N/A | Internal Resteasy error. Logged |
| DefaultOptionsMethodException | N/A | If the user invokes HTTP OPTIONS and no JAX-RS method for it, Resteasy provides a default behavior by throwing this exception |

## 27.3. Overriding Resteasy Builtin Exceptions

You may override Resteasy built-in exceptions by writing an ExceptionMapper for the exception. For that matter, you can write an ExceptionMapper for any thrown exception including WebApplicationException

# Configuring Individual JAX-RS Resource Beans

If you are scanning your path for JAX-RS annotated resource beans, your beans will be registered in per-request mode. This means an instance will be created per HTTP request served. Generally, you will need information from your environment. If you are running within a servlet container using the WAR-file distribution, in Beta-2 and lower, you can only use the JNDI lookups to obtain references to Java EE resources and configuration information. In this case, define your EE configuration (i.e. ejb-ref, env-entry, persistence-context-ref, etc...) within web.xml of the resteasy WAR file. Then within your code do 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 manually configure and register your beans through the Registry. To do this in a WAR-based deployment, you need to write a specific ServletContextListener to do this. Within the listern, you can obtain a reference to the registry as follows:

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

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


  }
...
}
```

Please also take a look at our Spring Integration as well as the Embedded Container's Spring Integration

# GZIP Compression/Decompression

Resteasy has automatic GZIP decompression support. If the client framework or a JAX-RS service receives a messagy body with a Content-Encoding of "gzip", it will automatically decompress it. The client framework automatically sets the Accept-Encoding header to be "gzip, deflate". So you do not have to set this header yourself.

Resteasy also supports automatic compression. If the client framework is sending a request or the server is sending a response with the Content-Encoding header set to "gzip", Resteasy will do the compression. So that you do not have to set the Content-Encoding header directly, you can use the @org.jboss.resteasy.annotation.GZIP annotation.

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

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

In the above example, we tag the outgoing message body, order, to be gzip compressed. You can use the same annotation to tag server responses

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

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

# Resteasy Caching Features

Resteasy provides numerous annotations and facilities to support HTTP caching semantics. Annotations to make setting Cache-Control headers easier and both server-side and client-side in-memory caches are available.

## 30.1. @Cache and @NoCache Annotations

Resteasy provides an extension to JAX-RS that allows you to automatically set Cache-Control headers on a successful GET request. It can only be used on @GET annotated methods. A successful @GET request is any request that returns 200 OK response.

```
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 {};
}
```

While @Cache builds a complex Cache-Control header, @NoCache is a simplified notation to say that you don't want anything cached i.e. Cache-Control: nocache.

These annotations can be put on the resource class or interface and specifies a default cache value for each @GET resource method. Or they can be put individually on each @GET resource method.

## 30.2. Client "Browser" Cache

Resteasy has the ability to set up a client-side, browser-like, cache. You can use it with the Client Proxy Framework, or with raw ClientRequests. This cache looks for Cache-Control headers sent back with a server response. If the Cache-Control headers specify that the client is allowed to cache the response, Resteasy caches it within local memory. The cache obeys max-age requirements and will also automatically do HTTP 1.1 cache revalidation if either or both the Last-Modified and/or ETag headers are sent back with the original response. See the HTTP 1.1 specification for details on how Cache-Control or cache revalidation works.

It is very simple to enable caching. Here's an example of using the client cache with the Client Proxy Framework

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

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

To create a proxy for this interface and enable caching for that proxy requires only a few simple steps:

```
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 to make invocations rather than the proxy framework, it is just as easy

```
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);
}
```

The LightweightBrowserCache, by default, has a maximum 2 megabytes of caching space. You can change this programmatically by callings its setMaxBytes() method. If the cache gets full, the cache completely wipes itself of all cached data. This may seem a bit draconian, but the cache was written to avoid unnecessary synchronizations in a concurrent environment where the cache is shared between multiple threads. If you desire a more complex caching solution or if you want to plug in a thirdparty cache please contact our resteasy-developers list and discuss it with the community.

## 30.3. Local Server-Side Response Cache

Resteasy has a server-side, local, in-memory cache that can sit in front of your JAX-RS services. It automatically caches marshalled responses from HTTP GET JAX-RS invocations if, and only if your JAX-RS resource method sets a Cache-Control header. When a GET comes in, the Resteasy Server Cache checks to see if the URI is stored in the cache. If it does, it returns the already marshalled response without invoking your JAX-RS method. Each cache entry has a max age to whatever is specified in the Cache-Control header of the initial request. The cache also will automatically generate an ETag using an MD5 hash on the response body. This allows the client to do HTTP 1.1 cache revalidation with the IF-NONE-MATCH header. The cache is also smart enough to perform revalidation if there is no initial cache hit, but the jax-rs method still returns a body that has the same ETag.

To set up the server-side cache, there are a few simple steps you have to perform. If you are using Maven you must depend on the resteasy-cache-core artifact:

```
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-cache-core</artifactId>
  <version>1.2.1.GA</version>
</dependency>
```

The next thing you have to do is to add a ServletContextListener, org.jboss.resteasy.plugins.cache.server.ServletServerCache. This must be specified 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: http://jboss.org/ jbosscache. There are two context-param configuration variables that you can set. resteasy.server.cache.maxsize sets the number of elements that can be cached. The resteasy.server.cache.eviction.wakeup.interval sets the rate at which the background eviction thread runs to purge the cache of stale entries.

# Interceptors

Resteasy has the capability to intercept JAX-RS invocations and route them through listener-like objects called interceptors. There are 4 different interception points on the serverside: wrapping around MessageBodyWriter invocations, wrapping around MessageBodyReader invocations, pre-processors the intercept the incoming request before anything is unmarshalled, and post processors which are invoked right after the JAX-RS method is finished. On the client side you can also intercept MessageBodyReader and Writer as well as the remote invocation to the server.

## 31.1. MessageBodyReader/Writer Interceptors

MessageBodyReader and Writer interceptors work off of the same principles. They wrap around the invocation of MessageBodyReader.readFrom() or MessageBodyWriter.writeTo(). You can use them to wrap the Output or InputStream. For example, the Resteasy GZIP support has interceptors that create and override the default Output and InputStream with a GzipOutpuStream or GzipInputStream so that gzip encoding can work. You could use them to append headers to the response (or on the client side, the outgoing request).

To implement one you 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. The interceptors and the MessageBodyReader or Writer is invoked in one big Java call stack. You must call MessageBodyReaderContext.proceed() or MessageBodyWriterContext.proceed() to go to the next interceptor or, if there are no more interceptors to invoke, the readFrom() or writeTo()

method of the MessageBodyReader or MessageBodyWriter. This wrapping allows you to modify things before they get to the Reader or Writer then clean up after proceed() returns. The Context objects also have methods to modify the parameters going to the Reader or Writer.

```java
public interface MessageBodyReaderContext
{
   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, 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 used on the serverside or client side. They must be annotated with @org.jboss.resteasy.annotations.interception.ServerInterceptor or @org.jboss.resteasy.annotations.interception.ClientInterceptor so that resteasy knows whether or not to add them to the interceptor list. If you do not annotate your interceptor classes with one or both of these annotations, you will receive a deployment error. They also should be annotated with @Provider. Lets look at an example:

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

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

Here we have a server side interceptor that adds a header value to the response. You see that it is annotated with @Provider and @ServerInterceptor. It must modify the header before calling context.proceed() as the reseponse may be committed after the MessageBodyReader runs. Remember, you MUST call context.proceed(). If you don't, your invocation will not happen.

## 31.2. PreProcessInterceptor

The org.jboss.resteasy.spi.interception.PreProcessInterceptor runs after a JAX-RS resource method is found to invoke on, but before the actual invocation happens. They are only usable on the server, but still must be annotated with @ServerInterceptor. They can be used to implement security features or can preempt the Java request. The Resteasy security implementation uses this type of interceptor to abort requests before the actually happen if the user does not pass authorization. The Resteasy caching framework also uses this to return cached responses to avoid invoking methods again. Here's what the interceptor interface looks like:

```
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. Here's some pseudo code that illustrates how they work:

```
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 get invoked and the runtime will process the response and return to the client.

## 31.3. PostProcessInterceptors

The org.jboss.resteasy.spi.interception.PostProcessInterceptor runs after the JAX-RS method was invoked but before MessageBodyWriters are invoked. They can only be used on the server side. Use them if you need to set a response header when there might not be any

MessageBodyWriter invoked. They are there for symetry with PreProcessInterceptor. They do not wrap anything and are invoked in order like PreProcessInterceptors are.

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

## 31.4. ClientExecutionInterceptors

org.jboss.resteasy.spi.interception.ClientExecutionInterceptor classes only are usable on the client side. They run after the MessageBodyWriter and after the ClientRequest has been totally built on the client side. They wrap around the actually HTTP invocation that goes to the server. Resteasy GZIP support uses them to set the Accept header to contain "gzip, deflate" before the request goes out. The Resteasy client cache uses it to check to see if its cache contains the resource before going over the wire. These interceptors must be annotated with @ClientInterceptor and @Provider.

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

public interface ClientExecutionContext
{
    ClientRequest getRequest();

    ClientResponse proceed() throws Exception;
}
```

The work work in the same pattern as MessageBodyReader/WriterInterceptors in that you must call proceed() unless you want to abort the invocation.

## 31.5. Binding Interceptors

By default, any registered interceptor will be invoked for any request you do. By default, every request will use your interceptors. You can fine tune 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 will invoke the accept() method. If this method returns true, Resteasy will add that interceptor to the JAX-RS method's call chain. If it returns false then it won't 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, our accept() method checks to see if the @GET annotation is present on our JAX-RS method. If it is, then this interceptor will be applied to that method's call chain.

## 31.6. Registering Interceptors

Registering interceptors is easy. Since they are a @Provider, (you remebered to annotate it right?) 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 Appication.getSingletons() method.

## 31.7. Interceptor Ordering and Precedence

Some interceptors are very sensitive in which order they are invoked. For example, you always want your security interceptor invoked first. Other interceptor's behavior might be triggered by a different interceptor that adds a header. By default, you have no control over the order in which registered interceptors are invoked. There is a way to specify interceptor precedence though.

You do not specify interceptor precedence by listing interceptor classes. Instead, there are precedence families and a particular interceptor class is associated with a family via the @org.jboss.resteasy.annotations.interception.Precedence annotation. We did this because some of the built in interceptors included with Resteasy are very sensitive to ordering. By specifying precedence through a family structure, we can protect these built in interceptors. An advantage to this approach is that configuration is also a lot easier too for you.

These are the families and the order in which they are executed:

```
SECURITY
HEADER_DECORATOR
ENCODER
REDIRECT
DECODER
```

Any interceptor not associated with a precedence family will be invoked last. SECURITY usually involves PreProcessInterceptors. They should be invoked first because you want to do as little as possible before your invocation is authorized. HEADER_DECORATORs are interceptors that add headers to a response or an outgoing request. They need to come next because these added headers may effect 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 usually are used in PreProcessInterceptors as they may reroute the request and totally bypas the JAX-RS method. DECODER interceptors wrap the InputStream. For example, the GZIP interceptor decoder wraps the InputStream in a GzipInputStream instance.

To marry your custom interceptors to a particular family you annotate it with the @org.jboss.resteasy.annotations.interception.Precendence annotation.

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

For complete type safety, there are convenience annotations in the org.jbos.resteasy.annotations.interception package: @DecoredPrecedence, @EncoderPrecedence, @HeaderDecoratorPrecedence, @RedirectPrecedence, @SecurityPrecedence. Use these instead of the @Precedence annotation

## 31.7.1. Custom Precedence

You can define your own precedence families. Apply them using 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. Otherwise, Resteasy will give you an error at deployment time. You do this with the context params:

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

resteasy.append.interceptor.precedence simply appends the precedence family to the list. resteasy.interceptor.before.precedence allows you to specify a family your new precedence comes before. resteasy.interceptor.after.precedence allows you to specify a family your new precedence comes 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</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>/test/*</url-pattern>
   </servlet-mapping>

</web-app>
```

In this web.xml file, we've define 3 new precedence families: END, BEFORE_ENCODER, and
AFTER_ENCODER. Here's what the family order would look like with this configuration:

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

# Asynchronous HTTP Request Processing

Asynchronous HTTP Request Processing is a relatively new technique that allows you to process a single HTTP request using non-blocking I/O and, if desired in separate threads. Some refer to it as COMET capabilities. The primary usecase for Asynchronous HTTP is in the case where the client is polling the server for a delayed response. The usual example is an AJAX chat client where you want to push/pull from both the client and the server. These scenarios have the client blocking a long time on the server's socket waiting for a new message. What happens in synchronous HTTP where the server is blocking on incoming and outgoing I/O is that you end up having a thread consumed per client connection. This eats up memory and valuable thread resources. Not such a big deal in 90% of applications (in fact using asynchronous processing make actually hurt your performance in most common scenaiors), but when you start getting a lot of concurrent clients that are blocking like this, there's a lot of wasted resources and your server does not scale that well.

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

The Resteasy asychronous HTTP support is implemented via two classes. The @Suspend annotation and the AsynchronousResponse interface.

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

import javax.ws.rs.core.Response;

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

The @Suspend annotation tells Resteasy that the HTTP request/response should be detached from the currently executing thread and that the current thread should not try to automaticaly process the response. The argument to @Suspend is a timeout in milliseconds until the request will be cancelled.

The AsynchronousResponse is the callback object. It is injected into the method by Resteasy. Application code hands off the AsynchronousResponse to a different thread for processing. The act of calling setResponse() will cause a response to be sent back to the client and will also terminate the HTTP request. Here 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();
  }
}
```

## 32.1. 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 Restasy 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 this:

```
<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. web.xml

```
<servlet>
   <servlet-name>Resteasy</servlet-name>
      <servlet-class>org.jboss.resteasy.plugins.server.servlet.Tomcat6CometDispatcherServlet</
servlet-class>
</servlet>
```

## 32.2. Servlet 3.0 Support

Our Servlet 3.0 support has only been tested with JBoss AS 6 M4 (trunk SVN as of 7/12/2010).

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. web.xml:

```
  <servlet>
    <servlet-name>Resteasy</servlet-name>
```

```
    <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServlet30Dispatcher</servlet-
class>
    <async-supported>true</async-supported>
  </servlet>
```

There's also a Filter30Dispatcher class if you want to use Resteasy as a filter. If you are running within JBoss AS 6 M4 or higher, you do not have to add this config to your web.xml if you are relying on the app server to do automatic scanning and have web.xml empty.

## 32.3. JBossWeb, JBoss AS 5.0.x Support

The JBossWeb container shipped with JBoss AS 5.0.x and higher requires you to install the JBoss Native plugin to enable asynchronous HTTP processing. Please see the JBoss Web documentation on how to do this.

Your deployed Resteasy applications must also 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. web.xml:

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

# Asynchronous Job Service

The Resteasy Asynchronous Job Service is an implementation of the Asynchronous Job pattern defined in O'Reilly's "Restful Web Services" book. The idea of it is to bring asynchronicity to a synchronous protocol.

## 33.1. Using Async Jobs

While HTTP is a synchronous protocol it does have a faint idea of asynchronous invocations. The HTTP 1.1 response code 202, "Accepted" means that the server has received and accepted the response for processing, but the processing has not yet been completed. The Resteasy Asynchronous Job Service builds around this idea.

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

For example, if you make the above post with the asynch query parameter set to true, Resteasy will return a 202, "Accepted" response code and run the invocation in the background. It also sends back a Location header with a URL pointing to where the response of the background method is located.

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 the GET, POST, and DELETE operations on this job URL. GET returns whatever the JAX-RS resource method you invoked returned as a response if the job was completed. If the job has not completed, this GET will return a response code of 202, Accepted. Invoking GET does not remove the job, so you can call it multiple times. When Resteasy's job queue gets full, it will evict the least recently used job from memory. You can manually clean up after yourself by calling DELETE on the URI. POST does a read of the JOB response and will remove the JOB it has been completed.

Both GET and POST allow you to specify a maximum wait time in milliseconds, a "wait" query parameter. Here's an 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. While these invocations may not change the state of the resource if invoked more than once, they do change the state of the server as new Job entries with each invocation. If you want to be a purist, stick with only invoking POST methods asynchronously.

Security NOTE! Resteasy role-based security (annotations) does not work with the Asynchronoous Job Service. You must use XML declaritive security within your web.xml file. Why? It is impossible to implement role-based security portably. In the future, we may have specific JBoss integration, but will not support other environments.

## 33.2. Oneway: Fire and Forget

Resteasy also supports the notion of fire and forget. This will also return a 202, Accepted response, but no Job will be created. This is as simple as using the oneway query parameter instead of asynch. For example:

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

Security NOTE! Resteasy role-based security (annotations) does not work with the Asynchronoous Job Service. You must use XML declaritive security within your web.xml file. Why? It is impossible to implement role-based security portably. In the future, we may have specific JBoss integration, but will not support other environments.

## 33.3. Setup and Configuration

You must enable the Asychronous Job Service in your web.xml file as it is not turned on by default.

```
<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>
```

```xml
<!-- 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>
```

# Embedded Container

RESTeasy JAX-RS comes with an embeddable server that you can run within your classpath. It packages TJWS embeddable servlet container with JAX-RS.

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

```java
@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.getDeployment().getActualResourceClasses().add(MyResource.class);
    tjws.start();
  }
}
```

The server can either host non-encrypted or SSL based resources, but not both. See the Javadoc for TJWSEmbeddedJaxrsServer as well as its superclass TJWSServletServer. The TJWS website is also a good place for information.

If you want to use Spring, see the SpringBeanProcessor. Here's a pseudo-code example

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

    tjws.start();
```

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

# Server-side Mock Framework

Although RESTEasy has an Embeddable Container, you may not be comfortable with the idea of starting and stopping a web server within unit tests (in reality, the embedded container starts in milli seconds), or you might not like the idea of using Apache HTTP Client or java.net.URL to test your code. RESTEasy provides a mock framework so that you can invoke on your resource directly.

```
import org.jboss.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 Javadoc for all the ease-of-use methods associated with MockHttpRequest, and MockHttpResponse.

# 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 do not mesh very well with JAX-RS in some situations. The problem is that web.xml URL pattern matching is very very limited. URL patterns in web.xml only support simple wildcards, so JAX-RS resources like:

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

Cannot be mapped as a web.xml URL pattern like:

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

To get around this problem you will need to use the security annotations defined below on your JAX-RS methods. You will still need to set up some general security constraint elements in web.xml to turn on authentication.

Resteasy JAX-RS supports the @RolesAllowed, @PermitAll and @DenyAll annotations on JAX-RS methods. By default though, Resteasy does not recognize these annotations. You have to configure Resteasy to turn on role-based security by setting a context parameter. NOTE!!! Do not turn on this switch if you are using EJBs. The EJB container will provide this functionality instead of Resteasy.

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

There is a bit of quirkiness with this approach. You will have to declare all roles used within the Resteasy JAX-RS war file that you are using in your JAX-RS classes and set up a security constraint that permits all of these roles access to every URL handled by the JAX-RS runtime. You'll just have to trust that Resteasy JAX-RS authorizes properly.

How does Resteasy do authorization? Well, its really simple. It just sees if a method is annotated with @RolesAllowed and then just does HttpServletRequest.isUserInRole. If one of the the @RolesAllowed passes, then allow the request, otherwise, a response is sent back with a 401 (Unauthorized) response code.

So, here's an example of a modified RESTEasy WAR file. You'll notice that every role declared is allowed access to every URL controlled by the Resteasy servlet.

```
<web-app>

  <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>

  <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>
```

# Authentication

RESTEasy has preliminary support for OAuth.

## 37.1. OAuth core 1.0a

RESTEasy has preliminary support for *OAuth core 1.0a* [http://oauth.net/core/1.0a]. This includes support for authenticating with OAuth (as described by the *spec section 6* [http://oauth.net/core/1.0a#rfc.section.6]) and OAuth authentication for protected resources (as described by the *spec section 7* [http://oauth.net/core/1.0a#rfc.section.7]).

> **Important**
>
> This API should be considered experimental and not suitable for production yet, especially for tight security. It is not final yet and subject to change. If you have comments, bugs, feature requests or questions, contact us through the *RESTEasy mailing list* [https://lists.sourceforge.net/lists/listinfo/RESTEasy-developers].

### 37.1.1. Authenticating with OAuth

OAuth authentication is the process in which Users grant access to their Protected Resources without sharing their credentials with the Consumer.

OAuth Authentication is done in three steps:

1. The Consumer obtains an unauthorized Request Token. This part is handled by RESTEasy.

2. The User authorizes the Request Token. This part is *not handled by RESTEasy* because it requires a user interface where the User logs in and authorizes or denies the Request Token. This cannot be implemented automatically as it needs to be integrated with your User login process and user interface.

3. The Consumer exchanges the Request Token for an Access Token. This part is handled by RESTEasy.

In order for RESTEasy to provide the two URL endpoints where the Client will request unauthorized Request Tokens and exchange authorized Request Tokens for Access Tokens, you need to enable the OAuthServlet in your web.xml:

```
<!-- The OAuth Servlet handles token exchange -->
<servlet>
```

```
 <servlet-name>OAuth</servlet-name>
 <servlet-class>org.jboss.RESTEasy.auth.oauth.OAuthServlet</servlet-class>
</servlet>

<!-- This will be the base for the token exchange endpoint URL -->
<servlet-mapping>
 <servlet-name>OAuth</servlet-name>
 <url-pattern>/oauth/*</url-pattern>
</servlet-mapping>
```

The following configuration options are available using `<context-param> elements`:

**Table 37.1. OAuth Servlet options**

| Option Name | Default | Description |
| --- | --- | --- |
| oauth.provider.provider-class | *Required* | Defines the fully-qualified class name of your OAuthProvider implementation |
| oauth.provider.tokens.request | /requestToken | This defines the endpoint URL for requesting unauthorized Request Tokens |
| oauth.provider.tokens.access | /accessToken | This defines the endpoint URL for exchanging authorized Request Tokens for Access Tokens |

## 37.1.2. Accessing protected resources

After successfully receiving the Access Token and Token Secret, the Consumer is able to access the Protected Resources on behalf of the User.

RESTEasy supports OAuth authentication for protected resources using a servlet filter wich should be mapped in your web.xml for all protected resources:

```
<!-- The OAuth Filter handles authentication for protected resources -->
<filter>
 <filter-name>OAuth Filter</filter-name>
 <filter-class>org.jboss.RESTEasy.auth.oauth.OAuthFilter</filter-class>
</filter>
```

```
<!-- This defines the URLs which should require OAuth authentication for your protected resources
 -->
<filter-mapping>
 <filter-name>OAuth Filter</filter-name>
 <url-pattern>/rest/*</url-pattern>
</filter-mapping>
```

The following configuration options are available using `<context-param> elements`:

**Table 37.2. OAuth Filter options**

| Option Name | Default | Description |
| --- | --- | --- |
| oauth.provider.provider-class | *Required* | Defines the fully-qualified class name of your OAuthProvider implementation |

Once authenticated, the OAuth Servlet Filter will set your request's Principal and Roles, which can then be accessed using the JAX-RS SecurityContext. You can also protect your resources using Roles as described in the section "Securing JAX-RS and RESTeasy".

## 37.1.3. Implementing an OAuthProvider

In order for RESTEasy to implement OAuth it needs you to provide an instance of `OAuthProvider` which will provide access to the list of Consumer, Request and Access Tokens. Because one size doesn't fit all we cannot know if you wish to store your Tokens and Consumer credentials in a configuration file, in memory, or on persistent storage.

All you need to do is implement the `OAuthProvider` interface:

```
public interface OAuthProvider {
 String getRealm();

 OAuthConsumer getConsumer(String consumerKey)throws OAuthException;
    OAuthToken    getRequestToken(String    consumerKey,    String    requestToken)    throws
OAuthException;
    OAuthToken    getAccessToken(String    consumerKey,    String    accessToken)    throws
OAuthException;
```

```
OAuthToken makeRequestToken(String consumerKey, String callback) throws OAuthException;
 OAuthToken makeAccessToken(String consumerKey, String requestToken, String verifier)
throws OAuthException;

  String  authoriseRequestToken(String  consumerKey,  String  requestToken)  throws
OAuthException;

void checkTimestamp(OAuthToken token, long timestamp) throws OAuthException;
}
```

If a Consumer Key, or Token doesn't exist, or if the timestamp is not valid, simply throw an `OAuthException`.

The rest of the interfaces used in `OAuthProvider` are:

```
public interface OAuthConsumer {
 String getKey();
 String getSecret();
}

public interface OAuthToken {
 OAuthConsumer getConsumer();
 String getToken();
 String getSecret();
 Principal getPrincipal();
 Set<String> getRoles();
}
```

# EJB Integration

To integrate with EJB you must first modify your EJB's published interfaces. Resteasy currently only has simple portable integration with EJBs so you must also manually configure your Resteasy WAR.

Resteasy currently only has simple integration with EJBs. To make an EJB a JAX-RS resource, you must annotate an SLSB's @Remote or @Local interface with JAX-RS annotations:

```
@Local
@Path("/Library")
public interface Library {

   @GET
   @Path("/books/{isbn}")
   public String getBook(@PathParam("isbn") String isbn);
}

@Stateless
public class LibraryBean implements Library {

...

}
```

Next, in RESTeasy's web.xml file you must manually register the EJB with RESTeasy using the resteasy.jndi.resources <context-param>

```
<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <context-param>
    <param-name>resteasy.jndi.resources</param-name>
    <param-value>LibraryBean/local</param-value>
  </context-param>

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

This is the only portable way we can offer EJB integration. Future versions of RESTeasy will have tighter integration with JBoss AS so you do not have to do any manual registrations or modications to web.xml. For right now though, we're focusing on portability.

If you're using Resteasy with an EAR and EJB, a good structure to have is:

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

From the distribution, remove all libraries from WEB-INF/lib and place them in a common EAR lib. OR. Just place the Resteasy jar dependencies in your application server's system classpath. (i.e. In JBoss put them in server/default/lib)

An example EAR project is available from our testsuite here.

# Spring Integration

RESTEasy integrates with Spring 3.0.x. We are interested in other forms of Spring integration, so please help contribute.

## 39.1. 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 comes with its own Spring ContextLoaderListener that registers a RESTeasy specific BeanPostProcessor that processes JAX-RS annotations when a bean is created by a BeanFactory. What does this mean? RESTeasy will automatically scan for @Provider and JAX-RS resource annotations on your bean class and register them as JAX-RS resources.

Here is what you have to do with your web.xml file

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

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

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

  <servlet>
    <servlet-name>Resteasy</servlet-name>
```

```
        <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-
class>
  </servlet>

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


</web-app>
```

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

If you do not use a Spring ContextLoaderListener to create your bean factories, then 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 attributes org.jboss.resteasy.spi.ResteasyProviderFactory and org.jboss.resteasy.spi.Registry. (Really the string FQN of these classes). There is also a org.jboss.resteasy.plugins.spring.SpringBeanProcessorServletAware, that will automatically inject references to the Registry and ResteasyProviderFactory from the Servlet Context. (that is, if you have used RestasyBootstrap to bootstrap Resteasy).

Our Spring integration supports both singletons and the "prototype" scope. RESTEasy handles injecting @Context references. Constructor injection is not supported though. Also, with the "prototype" scope, RESTEasy will inject any @*Param annotated fields or setters before the request is dispatched.

NOTE: You can only use auto-proxied beans with our base Spring integration. You will have undesirable affects if you are doing handcoded proxying with Spring, i.e., with ProxyFactoryBean. If you are using auto-proxied beans, you will be ok.

## 39.2. Spring MVC Integration

RESTEasy can also integrate with the Spring DispatcherServlet. The advantages of using this are that you have a simpler web.xml file, you can dispatch to either Spring controllers or Resteasy from under the same base URL, and finally, the most important, you can use Spring ModelAndView objects as return arguments from @GET resource methods. Setup requires you using the Spring DispatcherServlet in your web.xml file, as well as importing the springmvc-resteasy.xml file into your base Spring beans xml file. Here's an example web.xml file:

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

  <servlet>
    <servlet-name>Spring</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, 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"/>
....
```

# CDI Integration

This module provides integration with JSR-299 (Contexts and Dependency Injection for the Java EE platform)

## 40.1. Using CDI beans as JAX-RS components

Both the JAX-RS and CDI specifications introduce their own component model. On the one hand, every class placed in a CDI archive that fulfills a set of basic constraints is implicitly a CDI bean. On the other hand, explicit decoration of your Java class with `@Path` or `@Provider` is required for it to become a JAX-RS component. Without the integration code, annotating a class suitable for being a CDI bean with JAX-RS annotations leads into a faulty result (JAX-RS component not managed by CDI) The resteasy-cdi module is a bridge that allows RESTEasy to work with class instances obtained from the CDI container.

During a web service invocation, resteasy-cdi asks the CDI container for the managed instance of a JAX-RS component. Then, this instance is passed to RESTEasy. If a managed instance is not available for some reason (the class is placed in a jar which is not a bean deployment archive), RESTEasy falls back to instantiating the class itself.

As a result, CDI services like injection, lifecycle management, events, decoration and interceptor bindings can be used in JAX-RS components.

## 40.2. Default scopes

A CDI bean that does not explicitly define a scope is `@Dependent` scoped by default. This pseudo scope means that the bean adapts to the lifecycle of the bean it is injected into. Normal scopes (request, session, application) are more suitable for JAX-RS components as they designate component's lifecycle boundaries explicitly. Therefore, the resteasy-cdi module alters the default scoping in the following way:

- If a JAX-RS root resource does not define a scope explicitly, it is bound to the Request scope.

- If a JAX-RS Provider or `javax.ws.rs.Application` subclass does not define a scope explicitly, it is bound to the Application scope.

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

## 40.3. Configuration within JBoss 6 M4 and Higher

CDI integration is provided with no additional configuration with JBoss AS 6-M4 and higher.

## 40.4. Configuration with different distributions

Provided you have an existing RESTEasy application, all that needs to be done is to add the resteasy-cdi jar into your project's `WEB-INF/lib` directory. When using maven, this can be achieve by defining the following dependency.

```
<dependency>
 <groupId>org.jboss.resteasy</groupId>
 <artifactId>resteasy-cdi</artifactId>
 <version>${project.version}</version>
</dependency>
```

Furthermore, when running a pre-Servlet 3 container, the following context parameter needs to be specified in web.xml. (This is done automatically via web-fragment in a Servlet 3 environment)

```
<context-param>
 <param-name>resteasy.injector.factory</param-name>
 <param-value>org.jboss.resteasy.cdi.CdiInjectorFactory</param-value>
</context-param>
```

When deploying an application to a Servlet container that does not support CDI out of the box (Tomcat, Jetty, Google App Engine), a CDI implementation needs to be added first. *Weld-servlet module* [http://docs.jboss.org/weld/reference/latest/en-US/html/environments.html] can be used for this purpose.

# Seam Integration

RESTEasy integrates quite nicely with the JBoss Seam framework. This integration is maintained by the Seam developers and documented there as well. Check out *Seam documentation* [http://docs.jboss.org/seam/latest/en-US/html/webservices.html#d0e22078].

# Guice 2.0 Integration

RESTEasy has some simple integration with Guice 2.0. RESTEasy will scan the binding types for a Guice Module for @Path and @Provider annotations. It will register these bindings with RESTEasy. 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;
  }
}
```

First you start off by specifying a JAX-RS resource class. The HelloResource is just that. Next you create a Guice Module class that defines all 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);
    }
}
```

You put all these classes somewhere within your WAR WEB-INF/classes or in a JAR within WEB-INF/lib. Then you need to create your web.xml file. You need to use the GuiceResteasyBootstrapServletContextListener as follows

```
<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.GuiceResteasyBootstrapServletContextListener
        </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 notice that there is a resteasy.guice.modules context-param. This can take a comma delimited list of class names that are Guice Modules.

## 42.1. Configuring Stage

You can configure the stage Guice uses to deploy your modules by specific a context param, resteasy.guice.stage. If this value is not specified, Resteasy uses whatever Guice's default is.

```
<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>

<context-param>
   <param-name>resteasy.guice.stage</param-name>
   <param-value>PRODUCTION</param-value>
</context-param>

<listener>
   <listener-class>
      org.jboss.resteasy.plugins.guice.GuiceResteasyBootstrapServletContextListener
   </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>
```

# Client Framework

The Resteasy Client Framework is the mirror opposite of 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 builds an HTTP request that it uses to invoke on a remote RESTful Web Service. This remote service does not have to be a JAX-RS service and can be any web resource that accepts HTTP requests.

Resteasy has a client proxy framework that allows you to use JAX-RS annotations to invoke on a remote HTTP resource. The way it works is that you 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);
}
```

Resteasy has a simple API based on Apache HttpClient. You generate a proxy then you can invoke methods on the proxy. The invoked method gets translated to an HTTP request based on how you annotated the method and posted to the server. Here's how you would set this up:

```
import org.jboss.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");
```

Please see the ProxyFactory javadoc for more options. For instance, you may want to fine tune the HttpClient configuration.

@CookieParam works the mirror opposite of its server-side counterpart and creates a cookie header to send to the server. You do not need to use @CookieParam if you allocate your own javax.ws.rs.core.Cookie object and pass it as a parameter to a client proxy method. The client framework understands that you are passing a cookie to the server so no extra metadata is needed.

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);
```

## 43.1. Abstract Responses

Sometimes you are interested not only in the response body of a client request, but also either the response code and/or response headers. The Client-Proxy framework has two ways to get at this information

You may return a javax.ws.rs.core.Response.Status enumeration from your method calls:

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

Interally, after invoking on the server, the client proxy internals will convert the HTTP response code into a Response.Status enum.

If you are interested in everything, you can get it with the org.jboss.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 the 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 empty paramed getEntity() method can only be used if you have templated the ClientResponse within your method declaration. Resteasy uses this generic type information to know what type to unmarshal the

raw OutputStream into. The other two getEntity() methods that take parameters, allow you to specify which Object types you want to marshal the response into. These methods allow you to dynamically extract whatever types you want at runtime. Here's an example:

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

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

We need to include the LibraryPojo in ClientResponse's generic declaration so that the client proxy framework knows how to unmarshal the HTTP response body.

## 43.2. Sharing an interface between client and server

It is generally possible to share an interface between the client and server. In this scenario, you just have your JAX-RS services implement an annotated interface and then reuse that same interface to create client proxies to invoke on on the client-side. One caveat to this is when your JAX-RS methods return a Response object. The problem on the client is that the client does not have any type information with a raw Response return type declaration. There are two ways of getting around this. 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 approach isn't always good enough. The problem is that some MessageBodyReaders and Writers 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>>());
```

## 43.3. Client Error Handling

If you are using the Client Framework and your proxy methods return something other than a ClientResponse, then the default client error handling comes into play. Any response code that is greater tha 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
```

In cases where Client Proxy methods do not return Response or ClientResponse, it may be not be desireable for the Client Proxy Framework to throw generic ClientResponseFailure exceptions. In these scenarios, where more fine-grained control of thrown Exceptions is required, the ClientErrorInterceptor API may be used.

```
public static T getClientService(final Class clazz, final String serverUri)
{
```

```
ResteasyProviderFactory pf = ResteasyProviderFactory.getInstance();
pf.addClientErrorInterceptor(new DataExceptionInterceptor());


System.out.println("Generating REST service for: " + clazz.getName());
return ProxyFactory.create(clazz, serverUri);
}
```

ClientErrorInterceptor provides a hook into the proxy ClientResponse request lifecycle. If a Client Proxy method is called, resulting in a client exception, and the proxy return type is not Response or ClientResponse, registered interceptors will be given a chance to process the response manually, or throw a new exception. If all interceptors successfully return, RestEasy will re-throw the original encountered exception. Note, however, that the response input stream may need to be reset before additional reads will succeed.

```
public class ExampleInterceptor implements ClientErrorInterceptor
{
 public void handle(ClientResponse response) throws RuntimeException
 {
  try
  {
   BaseClientResponse r = (BaseClientResponse) response;
   InputStream stream = r.getStreamFactory().getInputStream();
   stream.reset();

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


   if(FORBIDDEN.equals(response.getResponseStatus()))
   {
    throw new MyCustomException("This exception will be thrown "
     + "instead of the ClientResponseFailure");
   }

  }
  catch (IOException e)
  {
   //...
  }
  // If we got here, and this method returns successfully,
  // RESTEasy will throw the original ClientResponseFailure
 }
```

```
}
```

## 43.4. Manual ClientRequest API

Resteasy has a manual API for invoking requests: org.jboss.resteasy.client.ClientRequest See the Javadoc for the full capabilities of this class. Here is a simple example:

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

// We're posting XML and a JAXB object
request.body("application/xml", someJaxb);

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

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

## 43.5. Spring integration on client side

When using spring you can generate a REST client proxy from an interface with the help of org.jboss.resteasy.client.spring.RestClientProxyFactoryBean.

```
<bean id="echoClient" class="org.jboss.resteasy.client.spring.RestClientProxyFactoryBean"
   p:serviceInterface="a.b.c.Echo" p:baseUri="http://server.far.far.away:8080/echo" />
```

# AJAX Client

RESTEasy resources can be accessed in JavaScript using AJAX using a proxy API generated by RESTEasy.

## 44.1. Generated JavaScript API

RESTEasy can generate a JavaScript API that uses AJAX calls to invoke JAX-RS operations.

**Example 44.1. First JAX-RS JavaScript API example**

Let's take a simple JAX-RS API:

```
@Path("orders")
public interface Orders {
 @Path("{id}")
 @GET
 public String getOrder(@PathParam("id") String id){
  return "Hello "+id;
 }
}
```

The preceding API would be accessible using the following JavaScript code:

```
var order = Orders.getOrder({id: 23});
```

## 44.1.1. JavaScript API servlet

In order to enable the JavaScript API servlet you must configure it in your web.xml file as such:

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

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

## 44.1.2. JavaScript API usage

Each JAX-RS resource class will generate a JavaScript object of the same name as the declaring class (or interface), which will contain every JAX-RS method as properties.

### Example 44.2. Structure of JAX-RS generated JavaScript

For example, if the JAX-RS resource X defines methods Y and Z:

```
@Path("/")
public interface X{
 @GET
 public String Y();
 @PUT
 public void Z(String entity);
}
```

Then the JavaScript API will define the following functions:

```
var X = {
 Y : function(params){…},
 Z : function(params){…}
};
```

Each JavaScript API method takes an optional object as single parameter where each property is a cookie, header, path, query or form parameter as identified by their name, or the following special parameters:

> **Warning**
>
> The following special parameter names are subject to change.

### Table 44.1. API parameter properties

| Property name | Default | Description |
| --- | --- | --- |
| $entity | | The entity to send as a PUT, POST request. |
| $contentType | As determined by @Consumes. | The MIME type of the body entity sent as the Content-Type header. |
| $accepts | | |

| Property name | Default | Description |
| --- | --- | --- |
| | Determined by @Provides, defaults to */*. | The accepted MIME types sent as the Accept header. |
| $callback | | Set to a function(httpCode, xmlHttpRequest, value) for an asynchronous call. If not present, the call will be synchronous and return the value. |
| $apiURL | Determined by container | Set to the base URI of your JAX-RS endpoint, not including the last slash. |
| $username | | If username and password are set, they will be used for credentials for the request. |
| $password | | If username and password are set, they will be used for credentials for the request. |

## Example 44.3. Using the API

Here is an example of JAX-RS API:

```
@Path("foo")
public class Foo{
 @Path("{id}")
 @GET
 public String get(@QueryParam("order") String order, @HeaderParam("X-Foo") String header,
          @MatrixParam("colour") String colour, @CookieParam("Foo-Cookie") String cookie){
 …
 }
 @POST
 public void post(String text){
 }
}
```

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

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

```
Foo.put({$entity: text});
```

## 44.1.3. MIME types and unmarshalling.

The Accept header sent by any client JavaScript function is controlled by the $accepts parameter, which overrides the @Produces annotation on the JAX-RS endpoint. The returned value however is controlled by the Content-Type header sent in the response as follows:

**Table 44.2. Return values by MIME type**

| MIME | Description |
|------|-------------|
| text/xml,application/xml,application/*+xml | The response entity is parsed as XML before being returned. The return value is thus a DOM Document. |
| application/json | The response entity is parsed as JSON before being returned. The return value is thus a JavaScript Object. |
| Anything else | The response entity is returned raw. |

**Example 44.4. Unmarshalling example**

The RESTEasy JavaScript client API can automatically unmarshall JSON and XML:

```
@Path("orders")
public interface Orders {

 @XmlRootElement
 public static class Order {
  @XmlElement
  private String id;

  public Order(){}

  public Order(String id){
   this.id = id;
  }
 }

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

```
 }

 @Path("{id}/json")
 @GET
 @Produces("application/json")
public Order getOrderJSON(@PathParam("id") String id){
  return new Order(id);
 }
}
```

Let us look at what the preceding JAX-RS API would give us on the client side:

```
// this returns a JSON object
var orderJSON = Orders.getOrderJSON({id: "23"});
orderJSON.id == "23";


// this one returns a DOM Document whose root element is the order, with one child (id)
// whose child is the text node value
var orderXML = Orders.getOrderXML({id: "23"});
orderXML.documentElement.childNodes[0].childNodes[0].nodeValue == "23";
```

## 44.1.4. MIME types and marshalling.

The Content-Type header sent in the request is controlled by the $contentType parameter which overrides the @Consumes annotation on the JAX-RS endpoint. The value passed as entity body using the $entity parameter is marshalled according to both its type and content type:

### Table 44.3. Controlling sent entities

| Type | MIME | Description |
| --- | --- | --- |
| DOM Element | Empty or text/xml,application/xml,application/*+xml | The DOM Element is marshalled to XML before being sent. |
| JavaScript Object (JSON) | Empty or application/json | The JSON object is marshalled to a JSON string before being sent. |
| Anything else | Anything else | The entity is sent as is. |

### Example 44.5. Marshalling example

The RESTEasy JavaScript client API can automatically marshall JSON and XML:

```
@Path("orders")
public interface Orders {

 @XmlRootElement
 public static class Order {
  @XmlElement
  private String id;

  public Order(){}

  public Order(String id){
   this.id = id;
  }
 }

 @Path("{id}/xml")
 @PUT
 @Consumes("application/xml")
 public void putOrderXML(Order order){
  // store order
 }

 @Path("{id}/json")
 @PUT
 @Consumes("application/json")
 public void putOrderJSON(Order order){
  // store order
 }
}
```

Let us look at what the preceding JAX-RS API would give us on the client side:

```
// this saves a JSON object
Orders.putOrderJSON({$entity: {id: "23"}});

// It is a bit more work with XML
var order = document.createElement("order");
var id = document.createElement("id");
order.appendChild(id);
id.appendChild(document.createTextNode("23"));
Orders.putOrderXML({$entity: order});
```

## 44.2. Using the JavaScript API to build AJAX queries

The RESTEasy JavaScript API can also be used to manually construct your requests.

### 44.2.1. The REST object

The REST object contains the following read-write properties:

**Table 44.4. The REST object**

| Property | Description |
| --- | --- |
| apiURL | Set by default to the JAX-RS root URL, used by every JavaScript client API functions when constructing the requests. |
| log | Set to a function(string) in order to receive RESTEasy client API logs. This is useful if you want to debug your client API and place the logs where you can see them. |

**Example 44.6. Using the REST object**

The REST object can be used to override RESTEasy JavaScript API client behaviour:

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

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

### 44.2.2. The REST.Request class

The REST.Request class is used to build custom requests. It has the following members:

**Table 44.5. The REST.Request class**

| Member | Description |
| --- | --- |
| execute(callback) | Executes the request with all the information set in the current object. The value is never returned but passed to the optional argument callback. |
| setAccepts(acceptHeader) | Sets the Accept request header. Defaults to */*. |

| Member | Description |
| --- | --- |
| setCredentials(username, password) | Sets the request credentials. |
| setEntity(entity) | Sets the request entity. |
| setContentType(contentTypeHeader) | Sets the Content-Type request header. |
| setURI(uri) | Sets the request URI. This should be an absolute URI. |
| setMethod(method) | Sets the request method. Defaults to GET. |
| setAsync(async) | Controls whether the request should be asynchronous. Defaults to true. |
| addCookie(name, value) | Sets the given cookie in the current document when executing the request. Beware that this will be persistent in your browser. |
| addQueryParameter(name, value) | Adds a query parameter to the URI query part. |
| addMatrixParameter(name, value) | Adds a matrix parameter (path parameter) to the last path segment of the request URI. |
| addHeader(name, value) | Adds a request header. |

## Example 44.7. Using the REST.Request class

The REST.Request class can be used to build custom requests:

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

# Maven and RESTEasy

JBoss's Maven Repository is at: http://repository.jboss.org/maven2

Here's the pom.xml fragment to use. Resteasy is modularized into various components. Mix and max as you see fit. Please replace 1.2.1.GA with the current 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>1.2.1.GA</version>
  </dependency>

  <!-- optional modules -->

  <!-- JAXB support -->
  <dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jaxb-provider</artifactId>
    <version>1.2.1.GA</version>
  </dependency>
  <!-- multipart/form-data and multipart/mixed support -->
  <dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-multipart-provider</artifactId>
    <version>1.2.1.GA</version>
  </dependency>
  <!-- Resteasy Server Cache -->
  <dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-cache-core</artifactId>
    <version>1.2.1.GA</version>
  </dependency>
```

```xml
<!-- Ruby YAML support -->
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-yaml-provider</artifactId>
  <version>1.2.1.GA</version>
</dependency>
<!-- JAXB + Atom support -->
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-atom-provider</artifactId>
  <version>1.2.1.GA</version>
</dependency>
<!-- Apache Abdera Integration -->
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>abdera-atom-provider</artifactId>
  <version>1.2.1.GA</version>
</dependency>
<!-- Spring integration -->
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-spring</artifactId>
  <version>1.2.1.GA</version>
</dependency>
<!-- Guice integration -->
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-guice</artifactId>
  <version>1.2.1.GA</version>
</dependency>

<!-- Asynchronous HTTP support with JBossWeb -->
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>async-http-jbossweb</artifactId>
  <version>1.2.1.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>1.2.1.GA</version>
</dependency>
```

```
    <!-- Asynchronous HTTP support with Tomcat 6 -->
    <dependency>
       <groupId>org.jboss.resteasy</groupId>
       <artifactId>async-http-tomcat6</artifactId>
       <version>1.2.1.GA</version>
    </dependency>

</dependencies>
```

There is also a pom that can be imported so the versions of the individual modules do not have to be specified. Note that maven 2.0.9 is required for this.

```
  <dependencyManagement>
    <dependencies>
      <dependency>
         <groupId>org.jboss.resteasy</groupId>
         <artifactId>resteasy-bom</artifactId>
         <version>1.2.1.GA</version>
         <type>pom</type>
         <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
```

# JBoss AS 5.x Integration

Resteasy 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.

# JBoss AS 6 Integration

RESTEasy is preconfigured and completely integrated with JBoss 6-M4 and higher. You can use it with EJB and CDI and you can rely completely on JBoss for scanning for your JAX-RS services and deploying them. All you have to provide is your JAX-RS service classes packaged within a WAR either as POJOs, CDI beans, or EJBs and provide an empty web.xml file as follows:

```xml
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
</web-app>
```

# Migration from older versions

## 48.1. Migrating from 1.2.x to 2.0

- TJWS has been forked to fix some bugs. The new groupId is org.jboss.resteasy, the artifactId is tjws. It will match the resteasy distribution version

- Please check out the JBoss 6 integration. It makes things alot easier if you are deploying in that environment

- There is a new Filter implementation that is the preferred deployment mechanism. Servlet-based deployments are still supported, but it is suggested you use to using a FilterDispatcher. See documentation for more details.

- As per required by the spec List or array injection of empty values will return an empty collection or array, not null. I.e. (@QueryParam("name") List<String> param) param will be an empty List. Resteasy 1.2.x and earlier would return null.

- We have forked TJWS, the servlet container used for embedded testing into the group org.jboss.resteasy, with the artifact id of tjws. You will need to remove these dependencies from your maven builds if you are using any part of the resteasy embebbable server. TJWS has a number of startup/shutdown race conditions we had to fix in order to make unit testing viable.

- Spring integration compiled against Spring 3.0.3. It may or may not still work with 2.5.6 and lower

## 48.2. Migrating from 1.2.GA to 1.2.1.GA

Methods @Deprecated within 1.2.GA have been removed. This is in the Client Framework and has to do with all references to Apache HTTP Client. You must now create an ClientExecutor if you want to manage your Apache HTTP Client sessions.

## 48.3. Migrating from 1.1 to 1.2

- The resteasy-maven-import artifact has been renamed to resteasy-bom

- Jettison and Fastinfoset have been broken out of the resteasy-jaxb-provider maven module. You will now need to include resteasy-jettison-provider or resteasy-fastinfoset-provider if you use either of these libraries.

- The constructors for ClientRequest that have a HttpClient parameter (Apache Http Client 3.1 API) are now deprecated. They will be removed in the final release of 1.2. You must create a Apache hTTP Client Executor and pass it in as a parameter if you want to re-use existing Apache HttpClient sessions or do any special configuration. The same is true for the ProxyFactoyr methods.

- Apache HttpClient 4.0 support is available if you want to use it. I've had some trouble with it so it is not the default implementation yet for the client framework.

- It is no longer required to call RegisterBuiltin.register() to initialize the set of providers. Too many users forgot to do this (include myself!). You can turn this off by calling the static method ResteasyProviderFactory.setRegisterBuiltinByDefault(false)

- The Embedded Container's API has changed to use org.jboss.resteasy.spi.ResteasyDeployment. Please see embedded documentation for more details.

# Books You Can Read

There are a number of great books that you can learn REST and JAX-RS from

- *RESTful Web Services* [http://oreilly.com/catalog/9780596529260/] by Leonard Richardson and Sam Ruby. A great introduction to REST.

- *RESTful Java with JAX-RS* [http://oreilly.com/catalog/9780596158040/] by Bill Burke. Overview of REST and detailed explanation of JAX-RS. Book examples are distributed with RESTEasy.

- *RESTful Web Services Cookbook* [http://oreilly.com/catalog/9780596808679/] by Subbu Allamaraju and Mike Amundsen. Detailed cookbook on how to design RESTful services.