RESTEasy JAX-RS

RESTFul Web Services for Java

1.0.0.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)

Installation/Configuration

RESTeasy is deployed as a WAR archive and thus depends on a Servlet container. When you download RESTeasy and unzip it you will see that it contains an exploded WAR. Make a deep copy of the WAR archive for your particular application. 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 configured by default to scan jars and classes within these directories for JAX-RS annotated classes and deploy and register them within the system.

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

```
<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <context-param>
    <param-name>resteasy.scan</param-name>
    <param-value>true</param-value>
  </context-param>

  <!-- set this if you map the Resteasy servlet to something other than /*
  <context-param>
    <param-name>resteasy.servlet.mapping.prefix</param-name>
    <param-value>/resteasy</param-value>
  </context-param>
  -->
     <!-- if you are using Spring, Seam or EJB as your component model, remove the
 ResourceMethodSecurityInterceptor -->
  <context-param>
    <param-name>resteasy.resource.method-interceptors</param-name>
    <param-value>
      org.jboss.resteasy.core.ResourceMethodSecurityInterceptor
    </param-value>
  </context-param>

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

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

```
  </servlet>

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

The ResteasyBootstrap listener is responsible for initializing some basic components of RESTeasy as well as scanning for annotation classes you have in your WAR file. It receives configuration options from <context-param> elements. Here's a list of what options are available

This config 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>
```

Table 2.1.

Option Name	Default Value	Description
resteasy.servlet.mapping.prefix	no default	If the url-pattern for the Resteasy servlet-mapping is not /*
resteasy.scan.providers	false	Scan for @Provider classes and register them
resteasy.scan.resources	false	

Option Name	Default Value	Description
		Scan for JAX-RS resource classes
resteasy.scan	false	Scan for both @Provider and JAX-RS resource classes (@Path, @GET, @POST etc..) and register them
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.core.Application	no default	Fully qualified name of Application class to bootstrap in a spec portable way

The ResteasyBootstrap listener 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.

2.1. javax.ws.rs.core.Application

javax.ws.rs.core.Application 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 context-param, javax.ws.rs.core.Application with a fully qualified class that implements Application. For example:

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

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

2.2. 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 2.2.

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(@PathParm("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.

3.1. @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

```
```

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

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(@PathParm("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(@PathParm("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) {...}
```

```
}
```

4.1. 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 4.1.

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

4.2. @PathParam and PathSegment

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

```
public interface PathSegment {

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

}
```

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

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

@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) {...}
```

@Cache, @NoCache, and CacheControl

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.

@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 @HttpContext 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, 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.

```
@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.

```
@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 @HttpContext. Read the javadoc for more info on these.

Content Marshalling/Providers

17.1. Default Providers and default JAX-RS Content Marshalling

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

Table 17.1.

Media Types	Java Type
application/*+xml, text/*+xml, application/*+json, application/*+fastinfoset, application/atom+*	JaxB annotated classes
/	java.lang.String
/	java.io.InputStream
text/plain	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

17.2. 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

17.3. MessageBodyWorkers

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

```
        /**
        * An injectable interface providing lookup of {@link MessageBodyReader} and
        * {@link MessageBodyWriter} instances.
        *
        * @see javax.ws.rs.core.Context
        * @see MessageBodyReader
        * @see MessageBodyWriter
        */
        public interface MessageBodyWorkers
        {

        /**
        * Get a message body reader that matches a set of criteria.
        *
        * @param mediaType the media type of the data that will be read, this will
        * be compared to the values of {@link javax.ws.rs.Consumes} for
        * each candidate reader and only matching readers will be queried.
        * @param type the class of object to be produced.
        * @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.
        */
          public abstract <T> MessageBodyReader<T> getMessageBodyReader(Class<T> type,
Type
        genericType, Annotation annotations[], MediaType mediaType);


        /**
        * Get a message body writer that matches a set of criteria.
        *
        * @param mediaType the media type of the data that will be written, this will
        * be compared to the values of {@link javax.ws.rs.Produces} for
        * each candidate writer and only matching writers will be queried.
```

```
 * @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.
 */
public abstract <T> MessageBodyWriter<T> getMessageBodyWriter(Class<T> type, Type
 genericType, Annotation annotations[], MediaType mediaType);
 }
```

MessageBodyWorkers are injectable into MessageBodyReader or Writers:

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

private @Context MessageBodyWorkers workers;

...

}
```

17.4. 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.

17.4.1. 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 ResteasyProviderFactory.getInstance().registerProvider(Class) or registerProviderInstance(Object)

17.4.2. JAXB + XML provider

17.4.3. 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/

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
```

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) {...}
```

17.4.4. 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.

17.4.5. Arrays and Collections of JAXB Objects

In combination with the @org.jboss.resteasy.annotations.providers.jaxb.Wrapped annotation on your methods and parameters, you can 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).

```
@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(@Wrapped Customer[] customers)
    {
      Assert.assertEquals("bill", customers[0].getName());
      Assert.assertEquals("monica", customers[1].getName());
    }

    @GET
    @Path("set")
    @Produces("application/xml")
    @Wrapped
    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(@Wrapped 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 in the http://jboss.org/resteasy namespace.

```
<resteasy:collection xmlns:resteasy="http://jboss.org/resteasy">
  <customer><name>bill</name></customer>
  <customer><name>monica</name></customer>
</resteasy:collection>
```

17.5. 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();
  }
  ...
}
```

17.6. 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

17.6.1. 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();
}
```

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.

17.6.2. 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.

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

17.6.3. 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

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

  <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.

17.6.4. 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)
   {
      ...
   }
}
```

17.6.5. 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;
    }
```

17.6.6. 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()
   {
      ...
   }
}
```

17.6.7. 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;
   }
```

17.6.8. 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()
   {
```

```
      ...
   }
}
```

17.6.9. @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 {
   @FormData("customer")
   @PartType("application/xml")
   private Customer customer;

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

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

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

17.7. 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.

17.7.1. 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.

```
import org.jboss.resteasy.plugins.providers.atom.Content;
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()
   {
      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");
      feed.getEntries().add(content);
      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")

17.7.2. 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);
  }
}
```

17.8. 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:

17.8.1. 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>
```

17.8.2. 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;
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);
   }
}
```

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 a proprietary @Provider interface that you can plug in:

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

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.

ExceptionMappers

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.

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

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.

```
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();
   }
}
```

22.1. 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>
```

22.2. Servlet 3.0 Support

As of October 20th, 2008, only Jetty 7.0.pre3 (mortbay.org) supports the current draft of the unfinished Servlet 3.0 specification so you will have to download and use Jetty to get this going.

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>
</servlet>
```

22.3. 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>
```

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:

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

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


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

The server can either host non-encrypted or SSL based resources, but not both. See the 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

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

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

    tjws.start();
  }
```

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.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 set up 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. 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>
  <display-name>Archetype Created Web Application</display-name>
  <context-param>
    <param-name>resteasy.scan</param-name>
    <param-value>true</param-value>
  </context-param>
```

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

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

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

  <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>
```

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

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

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

</web-app>
```

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 2.5. We are interested in other forms of Spring integration, so please help contribute.

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

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

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

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


</web-app>
```

The SpringContextLoaderListener must be declared after ResteasyBootstrap 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.resteasy.spi.ResteasyProviderFactory and org.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.

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

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


</web-app>
```

Then within your main Spring beans xml, 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"/>
....
```

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:

```
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.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);
```

28.1. 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.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.

28.2. 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>>());
```

28.3. 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
```

Maven and RESTEasy

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

Here's the pom.xml fragment to use. Please replace 1.0.0.GA with the current Resteasy version you want to use.

```
<repositories>
  <repository>
    <id>jboss</id>
    <url>http://repository.jboss.org/maven2</url>
  </repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jaxrs</artifactId>
    <version>1.0.0.GA</version>
  </dependency>
</dependencies>
```

Please download the RESTEasy JAX-RS source and view our testsuite. Everything is done in maven and we use our Embedded JAX-RS Container to do our testing.

Migration from older versions

30.1. Migrating to Resteasy Beta 6

- The org.resteasy package has been moved to org.jboss.resteasy. This means all your web.xml configurations will not work anymore
- @Path.limited doesn't exist in the spec anymore. Regular expressions are now support in @Path. See docs/spec for more details.
- Media type and language extension mappings are no longer in the specification. i.e. /foo.xml.en
- @ProduceMime/ConsumeMime annotations have been renamed @Produces/@Consumes respectively
- The default JAXB/JSON format is now Mapped rather than BadgerFish.
- javax.ws.rs.ApplicationConfig has been renamed to javax.ws.rs.Application and only has 2 methods. getClasses() and getSingletons()