Building RESTful Web Services with JAX-RS

This chapter describes the REST architecture, RESTful web services, and the Java API for RESTful Web Services (JAX-RS, defined in JSR 311).

Jersey, the reference implementation of JAX-RS, implements support for the annotations defined in JSR 311, making it easy for developers to build RESTful web services by using the Java programming language.

If you are developing with GlassFish Server, you can install the Jersey samples and documentation by using the Update Tool.

Instructions for using the Update Tool can be found in Java EE 6 Tutorial Component.

The Jersey samples and documentation are provided in the Available Add-ons area of the Update Tool.

The following topics are addressed here:

- What Are RESTful Web Services?
- Creating a RESTful Root Resource Class
- Example Applications for JAX-RS
- Further Information about JAX-RS

What Are RESTful Web Services?

RESTful web services are built to work best on the Web.

Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability,

that enable services to work best on the Web.

In the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs), typically links on the Web.

The resources are acted upon by using a set of simple, well-defined operations.

The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP.

In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

The following principles encourage RESTful applications to be simple, lightweight, and fast:

- **Resource identification through URI:** A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients.

Resources are identified by URIs, which provide a global addressing space for resource and service discovery.

See The `@Path` Annotation and URI Path Templates for more information.

- **Uniform interface: Resources are manipulated using a fixed set of four create, read, update, delete operations: `PUT`, `GET`, `POST`, and `DELETE`.**

`PUT` creates a new resource, which can be then deleted by using `DELETE`.

`GET` retrieves the current state of a resource in some representation.

`POST` transfers a new state onto a resource.

See Responding to HTTP Resources for more information.

- **Self-descriptive messages: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others.**

Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control.

See Responding to HTTP Resources and Using Entity Providers to Map HTTP Response and Request Entity Bodies for more information.

- **Stateful interactions through hyperlinks: Every interaction with a resource is stateless; that is, request messages are self-contained.**

Stateful interactions are based on the concept of explicit state transfer.

Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields.

State can be embedded in response messages to point to valid future states of the interaction.

See Using Entity Providers to Map HTTP Response and Request Entity Bodies and "Building URIs" in the JAX-RS Overview document for more information.

Creating a RESTful Root Resource Class

Root resource classes are POJOs that are either annotated with `@Path` or have at least one method annotated with `@Path` or a request method designator, such as `@GET`, `@PUT`, `@POST`, or `@DELETE`.

Resource methods are methods of a resource class annotated with a request method designator.

This section explains how to use JAX-RS to annotate Java classes to create RESTful web services.

Developing RESTful Web Services with JAX-RS

JAX-RS is a Java programming language API designed to make it easy to develop applications that use the REST architecture.

The JAX-RS API uses Java programming language annotations to simplify the development of RESTful web services.

Developers decorate Java programming language class files with JAX-RS annotations to define resources and the actions that can be performed on those resources.

JAX-RS annotations are runtime annotations; therefore, runtime reflection will generate the helper classes and artifacts for the resource.

A Java EE application archive containing JAX-RS resource classes will have the resources configured, the helper classes and artifacts generated, and the resource exposed to clients by deploying the archive to a Java EE server.

Table 19-1 lists some of the Java programming annotations that are defined by JAX-RS, with a brief description of how each is used.

Further information on the JAX-RS APIs can be viewed at http://download.oracle.com/javaee/6/api/.

Table 19-1 Summary of JAX-RS Annotations

Annotation	Description
`@Path`	The `@Path` annotation's value is a relative URI path indicating where the Java class will be hosted: for example, `/helloworld`. You can also embed variables in the URIs to make a URI path template. For example, you could ask for the name of a user and pass it to the application as a variable in the URI: `/helloworld/{username}`.

`@GET`	The `@GET` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@POST`	The `@POST` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP POST requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.

`@PUT`	The `@PUT` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP PUT requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@DELETE`	The `@DELETE` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP DELETE requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.

`@HEAD`	The `@HEAD` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP HEAD requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@PathParam`	The `@PathParam` annotation is a type of parameter that you can extract for use in your resource class. URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the `@Path` class-level annotation.

`@QueryParam`	The `@QueryParam` annotation is a type of parameter that you can extract for use in your resource class. Query parameters are extracted from the request URI query parameters.
`@Consumes`	The `@Consumes` annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client.
`@Produces`	The `@Produces` annotation is used to specify the MIME media types of representations a resource can produce and send back to the client: for example, `"text/plain"`.
`@Provider`	The `@Provider` annotation is used for anything that is of interest to the JAX-RS runtime, such as `MessageBodyReader` and `MessageBodyWriter`.

For HTTP requests, the `MessageBodyReader` is used to map an HTTP request entity body to method parameters.

On the response side, a return value is mapped to an HTTP response entity body by using a `MessageBodyWriter`.

If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a `Response` that wraps the entity and that can be built using `Response.ResponseBuilder`.

Overview of a JAX-RS Application

The following code sample is a very simple example of a root resource class that uses JAX-RS annotations:

```
package
com.sun.jersey.samples.helloworld.
resources;
import  javax.ws.rs.GET;
```

```
import javax.ws.rs.Produces;
import javax.ws.rs.Path;
// The Java class will be hosted
// at the URI path "/helloworld"
@Path("/helloworld")
public class HelloWorldResource{
// The Java method will
// process HTTP GET requests
@GET
```

```
// The Java method will produce
// content identified by
// the MIME Media
// type "text/plain"
@Produces("text/plain")
public String getClichedMessage(){
// Return some cliched
// textual content
return "Hello World";
}
}
```

The following sections describe the annotations used in this example.

- The `@Path` annotation's value is a relative URI path.

In the preceding example, the Java class will be hosted at the URI path `/helloworld`.

This is an extremely simple use of the `@Path` annotation, with a static URI path.

Variables can be embedded in the URIs.

URI path templates are URIs with variables embedded within the URI syntax.

- The `@GET` annotation is a request method designator, along with `@POST`, `@PUT`, `@DELETE`, and `@HEAD`, defined by JAX-RS and corresponding to the similarly named HTTP methods.

In the example, the annotated Java method will process HTTP `GET` requests.

The behavior of a resource is determined by the HTTP method to which the resource is responding.

- The `@Produces` annotation is used to specify the MIME media types a resource can produce and send back to the client.

In this example, the Java method will produce representations identified by the MIME media type `"text/plain"`.

- The `@Consumes` annotation is used to specify the MIME media types a resource can consume that were sent by the client.

The example could be modified to set the message returned by the `getClichedMessage` method, as shown in this code example:

```
@POST
@Consumes("text/plain")
public void
postClichedMessage(String message)
{ // Store the message }
```

The `@Path` Annotation and URI Path Templates

The `@Path` annotation identifies the URI path template to which the resource responds and is specified at the class or method level of a resource.

The `@Path` annotation's value is a partial URI path template relative to the base URI of the server on which the resource is deployed, the context root of the application, and the URL pattern to which the JAX-RS runtime responds.

URI path templates are URIs with variables embedded within the URI syntax.

These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI.

Variables are denoted by braces ({ and }).

For example, look at the following @Path annotation:

```
@Path("/users/{username}")
```

In this kind of example, a user is prompted to type his or her name, and then a JAX-RS web service configured to respond to requests to this URI path template responds.

For example, if the user types the user name "Galileo," the web service responds to the following URL:

```
http://example.com/users/Galileo
```

To obtain the value of the user name, the @PathParam annotation may be used on the method parameter of a request method, as shown in the following code example:

```
@Path("/users/{username}")
public class UserResource {
@GET
@Produces("text/xml")

public String getUser
```

```
(@PathParam("username")
String userName){...}
}
```

By default, the URI variable must match the regular expression `"[^/]+?"`.

This variable may be customized by specifying a different regular expression after the variable

name.

For example, if a user name must consist only of lowercase and uppercase alphanumeric characters, override the default regular expression in the variable definition:

```
@Path("users/{username:
[a-zA-Z][a-zA-Z_0-9]}")
```

In this example the `username` variable will match only user names that begin with one uppercase or lowercase letter and zero or more alphanumeric characters and the underscore character.

If a user name does not match that template, a 404 (Not Found) response will be sent to the client.

A `@Path` value isn't required to have leading or trailing slashes (/).

The JAX-RS runtime parses URI path templates the same whether or not they have leading or trailing spaces.

A URI path template has one or more variables, with each variable name surrounded by braces: { to begin the variable name and } to end it.

In the preceding example, `username` is the variable name.

At runtime, a resource configured to respond to the preceding URI path template will attempt to process the URI data that corresponds to the location of `{username}` in the URI as the variable data for `username`.

For example, if you want to deploy a resource that responds to the URI path template `http://example.com/myContextRoot/resources/{name1}/{name2}/`, you must deploy the application to a Java EE server that responds to requests to the `http://example.com/myContextRoot` URI and then decorate your resource with the following `@Path` annotation:

```
@Path("/{name1}/{name2}/")
public class SomeResource {...}
```

In this example, the URL pattern for the JAX-RS helper servlet, specified in `web.xml`, is the default:

```
<servlet-mapping>
<servlet-name>
My JAX-RS Resource
```

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

A variable name can be used more than once in the URI path template.

If a character in the value of a variable would conflict with the reserved characters of a URI, the conflicting character should be substituted with percent encoding.

For example, spaces in the value of a variable should be substituted with %20.

When defining URI path templates, be careful that the resulting URI after substitution is valid.

Table 19-2 lists some examples of URI path template variables and how the URIs are resolved after substitution.

The following variable names and values are used in the examples:

- `name1: james`
- `name2: gatz`
- `name3:`
- `location: Main%20Street`
- `question: why`

Note - The value of the `name3` variable is an empty string.

Table 19-2 Examples of URI Path Templates

URI Path Template	URI After Substitution
`http://example.com/{name1}/{name2}/`	`http://example.com/james/gatz/`
`http://example.com/{question}/{question}/{question}/`	`http://example.com/why/why/why/`
`http://example.com/maps/{location}`	`http://example.com/maps/Main%20Street`
`http://example.com/{name3}/home/`	`http://example.com//home/`

Responding to HTTP Resources

The behavior of a resource is determined by the HTTP methods (typically, `GET`, `POST`, `PUT`, `DELETE`) to which the resource is responding.

The Request Method Designator Annotations

Request method designator annotations are runtime annotations, defined by JAX-RS, that correspond to the similarly named HTTP methods.

Within a resource class file, HTTP methods are mapped to Java programming language methods by using the request method designator annotations.

The behavior of a resource is determined by which HTTP method the resource is responding to.

JAX-RS defines a set of request method designators for the common HTTP methods `@GET`, `@POST`, `@PUT`, `@DELETE`, and `@HEAD`; you can also create your own custom request method designators.

Creating custom request method designators is outside the scope of this document.

The following example, an extract from the storage service sample, shows the use of the PUT method to create or update a storage container:

```
@PUT
public Response putContainer(){
System.out.println
("PUT CONTAINER " + container);
URI uri =
uriInfo.getAbsolutePath();
```

```
Container c = new Container
(container, uri.toString());
Response r;
if
(!MemoryStore.MS.hasContainer(c)){
r = Response.created(uri).build();
}else{
r = Response.noContent().build();
}
MemoryStore.MS.createContainer(c);
return r; }
```

By default, the JAX-RS runtime will automatically support the methods `HEAD` and `OPTIONS` if not explicitly implemented.

For `HEAD`, the runtime will invoke the implemented `GET` method, if present, and ignore the response entity, if set.

For `OPTIONS`, the `Allow` response header will be set to the set of HTTP methods supported by the resource.

In addition, the JAX-RS runtime will return a Web Application Definition Language (WADL) document describing the resource; see http://wadl.java.net/ for more information.

Methods decorated with request method designators must return `void`, a Java programming language type, or a `javax.ws.rs.core.Response` object.

Multiple parameters may be extracted from the URI by using the `PathParam` or `QueryParam` annotations as described in Extracting Request Parameters.

Conversion between Java types and an entity body is the responsibility of an entity provider, such as `MessageBodyReader` or `MessageBodyWriter`.

Methods that need to provide additional metadata with a response should return an instance of the `Response` class.

The `ResponseBuilder` class provides a convenient way to create a `Response` instance using a builder pattern.

The HTTP `PUT` and `POST` methods expect an HTTP request body, so you should use a `MessageBodyReader` for methods that respond to `PUT` and `POST` requests.

Both `@PUT` and `@POST` can be used to create or update a resource.

`POST` can mean anything, so when using `POST`, it is up to the application to define the semantics.

`PUT` has well-defined semantics.

When using `PUT` for creation, the client declares the URI for the newly created resource.

`PUT` has very clear semantics for creating and updating a resource.

The representation the client sends must be the same representation that is received using a `GET`, given the same media type.

`PUT` does not allow a resource to be partially updated, a common mistake when attempting to use the `PUT` method.

A common application pattern is to use `POST` to create a resource and return a `201` response with a location header whose value is the URI to the newly created resource.

In this pattern, the web service declares the URI for the newly created resource.

Using Entity Providers to Map HTTP Response and Request Entity Bodies

Entity providers supply mapping services between representations and their associated Java types.

The two types of entity providers are `MessageBodyReader` and `MessageBodyWriter`.

For HTTP requests, the `MessageBodyReader` is used to map an HTTP request entity body to method parameters.

On the response side, a return value is mapped to an HTTP response entity body by using a `MessageBodyWriter`.

If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a `Response` that wraps the entity and that can be built by using `Response.ResponseBuilder`.

Table 19-3 shows the standard types that are supported automatically for entities.

You need to write an entity provider only if you are not choosing one of these standard types.

Table 19-3 Types Supported for Entities

Java Type	Supported Media Types
`byte[]`	All media types (`*/*`)
`java.lang.String`	All text media types (`text/*`)
`java.io.InputStream`	All media types (`*/*`)
`java.io.Reader`	All media types (`*/*`)
`java.io.File`	All media types (`*/*`)
`javax.activation.DataSource`	All media types (`*/*`)

`javax.xml.transform.Source`	XML media types (`text/xml`, `application/xml`, and `application/*+xml`)
`javax.xml.bind.JAXBElement` and application-supplied JAXB classes	XML media types (`text/xml`, `application/xml`, and `application/*+xml`)
`MultivaluedMap<String, String>`	Form content (`application/x-www-form-urlencoded`)
`StreamingOutput`	All media types (`*/*`), `MessageBodyWriter` only

The following example shows how to use MessageBodyReader with the @Consumes and @Provider annotations:

```
@Consumes
("application/x-www-form-urlencoded")
@Provider
public class FormReader implements
MessageBodyReader<NameValuePair>{
```

The following example shows how to use MessageBodyWriter with the @Produces and @Provider annotations:

```
@Produces("text/html")
@Provider
public class FormWriter implements
MessageBodyWriter
<Hashtable<String, String>> {
```

The following example shows how to use ResponseBuilder:

```
@GET
public Response getItem(){
  System.out.println
("GET ITEM "+container+ " "+ item);
Item i = MemoryStore.MS.getItem
(container, item);
```

```
if (i == null)
throw new NotFoundException
("Item not found");
Date lastModified =
i.getLastModified().getTime();
EntityTag et =
new EntityTag(i.getDigest());
ResponseBuilder rb =
request.evaluatePreconditions
(lastModified, et);
```

```
if (rb != null)
return rb.build();
byte[] b =
MemoryStore.MS.getItemData
(container, item);
return
Response.ok(b, i.getMimeType()).
lastModified(lastModified).
tag(et).build();
}
```

Using `@Consumes` and `@Produces` to Customize Requests and Responses

The information sent to a resource and then passed back to the client is specified as a MIME media type in the headers of an HTTP request or response.

You can specify which MIME media types of representations a resource can respond to or produce by using the following annotations:

- `javax.ws.rs.Consumes`
- `javax.ws.rs.Produces`

By default, a resource class can respond to and produce all MIME media types of representations specified in the HTTP request and response headers.

The `@Produces` Annotation

The `@Produces` annotation is used to specify the MIME media types or representations a resource can produce and send back to the client.

If `@Produces` is applied at the class level, all the methods in a resource can produce the specified MIME types by default.

If applied at the method level, the annotation overrides any `@Produces` annotations applied at the class level.

If no methods in a resource are able to produce the MIME type in a client request, the JAX-RS runtime sends back an HTTP "406 Not Acceptable" error.

The value of `@Produces` is an array of `String` of MIME types.

For example:

```
@Produces({"image/jpeg,image/png"})
```

The following example shows how to apply `@Produces` at both the class and method levels:

```
@Path("/myResource")
@Produces("text/plain")
```

```
public class SomeResource {
@GET
public String doGetAsPlainText()
{... }
@GET
@Produces("text/html")
public String doGetAsHtml(){...}
}
```

The `doGetAsPlainText` method defaults to the MIME media type of the `@Produces` annotation at the class level.

The `doGetAsHtml` method's `@Produces` annotation overrides the class-level `@Produces` setting and specifies that the method can produce HTML rather than plain text.

If a resource class is capable of producing more than one MIME media type, the resource method chosen will correspond to the most acceptable media type as declared by the client.

More specifically, the `Accept` header of the HTTP request declares what is most acceptable.

For example, if the `Accept` header is `Accept: text/plain`, the `doGetAsPlainText` method will be invoked.

Alternatively, if the `Accept` header is `Accept: text/plain;q=0.9, text/html`, which declares that the client can accept media types of `text/plain` and `text/html` but prefers the latter, the `doGetAsHtml` method will be invoked.

More than one media type may be declared in the same `@Produces` declaration.

The following code example shows how this is done:

```
@Produces({"application/xml",
"application/json"})
public String doGetAsXmlOrJson()
{...}
```

The `doGetAsXmlOrJson` method will get invoked if either of the media types `application/xml` and `application/json` is acceptable.

If both are equally acceptable, the former will be chosen because it occurs first.

The preceding examples refer explicitly to MIME media types for clarity.

It is possible to refer to constant values, which may reduce typographical errors.

For more information, see the constant field values of `MediaType` at http://jsr311.java.net/nonav/releases/1.0/javax/ws/rs/core/MediaType.html.

The `@Consumes` Annotation

The `@Consumes` annotation is used to specify which MIME media types of representations a resource can accept, or consume, from the client.

If `@Consumes` is applied at the class level, all the response methods accept the specified MIME types by default.

If applied at the method level, `@Consumes` overrides any `@Consumes` annotations applied at the class level.

If a resource is unable to consume the MIME type of a client request, the JAX-RS runtime sends back an HTTP 415 ("Unsupported Media Type") error.

The value of `@Consumes` is an array of `String` of acceptable MIME types.

For example:

```
@Consumes({"text/plain,text/html"})
```

The following example shows how to apply `@Consumes` at both the class and method levels:

```
@Path("/myResource")
@Consumes("multipart/related")
public class SomeResource {
```

```
@POST
public String doPost
 (MimeMultipart mimeMultipartData)
 {...}
@POST
@Consumes
 ("application/x-www-form-urlencoded")
public String doPost2
 (FormURLEncodedProperties formData)
 {...}
 }
```

The `doPost` method defaults to the MIME media type of the `@Consumes` annotation at the class level.

The `doPost2` method overrides the class level `@Consumes` annotation to specify that it can accept URL-encoded form data.

If no resource methods can respond to the requested MIME type, an HTTP 415 ("Unsupported Media Type") error is returned to the client.

The `HelloWorld` example discussed previously in this section can be modified to set the message by using `@Consumes`, as shown in the following code example:

```
@POST
@Consumes("text/plain")
public void
postClichedMessage(String message){
// Store the message
}
```

In this example, the Java method will consume representations identified by the MIME media type `text/plain`.

Note that the resource method returns `void`.

This means that no representation is returned and that a response with a status code of HTTP 204 (“No Content”) will be returned.

Extracting Request Parameters

Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request.

A previous example presented the use of the `@PathParam` parameter to extract a path parameter from the path component of the request URL that matched the path declared in `@Path`.

You can extract the following types of parameters for use in your resource class:

- Query
- URI path
- Form
- Cookie
- Header
- Matrix

Query parameters are extracted from the request URI query parameters and are specified by using the `javax.ws.rs.QueryParam` annotation in the method parameter arguments.

The following example, from the `sparklines` sample application, demonstrates using `@QueryParam` to extract query parameters from the `Query` component of the request URL:

```
@Path("smooth")
@GET
public Response smooth(
@DefaultValue("2")
@QueryParam("step")
int step,
@DefaultValue("true")
@QueryParam("min-m")
boolean hasMin,
```

```
@DefaultValue("true")
@QueryParam("max-m")
boolean hasMax,
@DefaultValue("true")
@QueryParam("last-m")
boolean hasLast,
@DefaultValue("blue")
@QueryParam("min-color")
ColorParam minColor,
```

```
@DefaultValue("green")
@QueryParam("max-color")
ColorParam maxColor,
@DefaultValue("red")
@QueryParam("last-color")
ColorParam lastColor
) { ... }
```

If the query parameter `step` exists in the query component of the request URI, the value of `step` will be extracted and parsed as a 32-bit signed integer and assigned to the `step` method parameter.

If `step` does not exist, a default value of 2, as declared in the `@DefaultValue` annotation, will be assigned to the `step` method parameter.

If the `step` value cannot be parsed as a 32-bit signed integer, an HTTP 400 ("Client Error") response is returned.

User-defined Java programming language types may be used as query parameters.

The following code example shows the `ColorParam` class used in the preceding query parameter example:

```
public class ColorParam extends
Color {
public ColorParam(String s)
{super(getRGB(s));}
private static int getRGB(String s){
if (s.charAt(0) == '#') {
try {
Color c =
Color.decode("0x" +s.substring(1));
return c.getRGB();
} catch (NumberFormatException e) {
```

```
throw
new WebApplicationException(400);
}
} else {
try {
Field f = Color.class.getField(s);
return
((Color)f.get(null)).getRGB();
} catch (Exception e) {
throw
new WebApplicationException(400);
```

```
}}}}
```

The constructor for `ColorParam` takes a single `String` parameter.

Both `@QueryParam` and `@PathParam` can be used only on the following Java types:

- All primitive types except `char`

- All wrapper classes of primitive types except `Character`
- Any class with a constructor that accepts a single `String` argument
- Any class with the static method named `valueOf(String)` that accepts a single `String` argument
- `List<T>`, `Set<T>`, or `SortedSet<T>`, where *T* matches the already listed criteria.

Sometimes, parameters may contain more than one value for the same name.

If this is the case, these types may be used to obtain all values

If `@DefaultValue` is not used in conjunction with `@QueryParam`, and the query parameter is not present in the request, the value will be an empty collection for `List`, `Set`, or `SortedSet`;

null for other object types; and the default for primitive types.

URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the `@Path` class-level annotation.

URI parameters are specified using the `javax.ws.rs.PathParam` annotation in the method parameter arguments.

The following example shows how to use `@Path` variables and the `@PathParam` annotation in a method:

```
@Path("/{username}")
public class MyResourceBean { ...
```

```
@GET
public String printUsername
(@PathParam("username")
String userId)
{...} }
```

In the preceding snippet, the URI path template variable name `username` is specified as a parameter to the `printUsername` method.

The `@PathParam` annotation is set to the variable name `username`.

At runtime, before `printUsername` is called, the value of `username` is extracted from the URI and cast to a `String`.

The resulting `String` is then available to the method as the `userId` variable.

If the URI path template variable cannot be cast to the specified type, the JAX-RS runtime returns an HTTP 400 (“Bad Request”) error to the client.

If the `@PathParam` annotation cannot be cast to the specified type, the JAX-RS runtime returns an HTTP 404 (“Not Found”) error to the client.

The `@PathParam` parameter and the other parameter-based annotations (`@MatrixParam`, `@HeaderParam`, `@CookieParam`, and `@FormParam`) obey the same rules as `@QueryParam`.

Cookie parameters, indicated by decorating the parameter with `javax.ws.rs.CookieParam`, extract information from the cookies declared in cookie-related HTTP headers.

Header parameters, indicated by decorating the parameter with `javax.ws.rs.HeaderParam`, extract information from the HTTP headers.

Matrix parameters, indicated by decorating the parameter with `javax.ws.rs.MatrixParam`, extract information from URL path segments.

Form parameters, indicated by decorating the parameter with `javax.ws.rs.FormParam`, extract information from a request representation that is of the MIME media type `application/x-www-form-urlencoded` and conforms to the encoding specified by HTML forms, as described in http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.1.

This parameter is very useful for extracting information sent by `POST` in HTML forms.

The following example extracts the `name` form parameter from the `POST` form data:

```
@POST
@Consumes
("application/x-www-form-urlencoded")
public void post
```

```
(@FormParam("name") String name) {
// Store the message
}
```

To obtain a general map of parameter names and values for query and path parameters, use the following code:

```
@GET
public String get
(@Context UriInfo ui){
```

```
MultivaluedMap<String, String>
queryParams =
ui.getQueryParameters();
MultivaluedMap<String, String>
pathParams =
ui.getPathParameters();
}
```

The following method extracts header and cookie parameter names and values into a map:

```
@GET
public String get
(@Context HttpHeaders hh){
 MultivaluedMap<String, String>
headerParams =
ui.getRequestHeaders();
Map<String, Cookie> pathParams =
ui.getCookies();
}
```

In general, `@Context` can be used to obtain contextual Java types related to the request or response.

For form parameters, it is possible to do the following:

```
@POST
@Consumes
("application/x-www-form-urlencoded")
```

```
public void post
(MultivaluedMap<String, String>
formParams){
// Store the message
}
```

Example Applications for JAX-RS

This section provides an introduction to creating, deploying, and running your own JAX-RS applications.

This section demonstrates the steps that are needed to create, build, deploy, and test a very simple web application that uses JAX-RS annotations.

A RESTful Web Service

This section explains how to use NetBeans IDE to create a RESTful web service.

NetBeans IDE generates a skeleton for the application, and you simply need to implement the appropriate methods.

If you do not use an IDE, try using one of the example applications that ship with Jersey as a template to modify.

You can find a version of this application at *tut-install*`/examples/jaxrs/HelloWorldApplication`.

To Create a RESTful Web Service Using NetBeans IDE

1. In NetBeans IDE, create a simple web application.

   This example creates a very simple "Hello, World" web application.

   a. From the File menu, choose New Project.

b. From Categories, select Java Web.

From Projects, select Web Application.

Click Next.

Note - For this step, you could also create a RESTful web service in a Maven web project by selecting Maven as the category and Maven Web Project as the project.

The remaining steps would be the same.

c. Type a project name, `HelloWorldApplication`, and click Next.

d. Make sure that the Server is GlassFish Server (or similar wording.)

e. Click Finish.

The project is created.

The file `index.jsp` appears in the Source pane.

2. Right-click the project and select New; then select RESTful Web Services from Patterns.

a. Select Simple Root Resource and click Next.

b. Type a Resource Package name, such as `helloWorld`.

c. Type `helloworld` in the Path field.

Type `HelloWorld` in the Class Name field.

For MIME Type, select `text/html`.

d. Click Finish.

The REST Resources Configuration page appears.

e. Click OK.

A new resource, `HelloWorld.java`, is added to the project and appears in the Source pane.

This file provides a template for creating a RESTful web service.

3. In `HelloWorld.java`, find the `getHtml()` method.

Replace the //TODO comment and the exception with the following text, so that the finished product resembles the following method.

Note - Because the MIME type produced is HTML, you can use HTML tags in your return statement.

```
/**
 * Retrieves representation of
 * an instance of
 * helloWorld.HelloWorld
 * @return an instance
 * of java.lang.String
 */
@GET
@Produces("text/html")
public String getHtml(){
```

```
return "<html><body><h1>Hello,
World!!</body></h1></html>";
}
```

4. Test the web service.

To do this, right-click the project node and click Test RESTful Web Services.

This step deploys the application and brings up a test client in the browser.

5. When the test client appears, select the `helloworld` resource in the left pane, and click the Test button in the right pane.

The words `Hello, World!!` appear in the Response window below.

6. Set the Run Properties:

a. Right-click the project node and select Properties.

b. In the dialog, select the Run category.

c. Set the Relative URL to the location of the RESTful web service relative to the Context Path, which for this example is `resources/helloworld`.

Tip - You can find the value for the Relative URL in the Test RESTful Web Services browser window.

In the top of the right pane, after Resource, is the URL for the RESTful web service being tested.

The part following the Context Path (`http://localhost:8080/HelloWorldApp`) is the Relative URL that needs to be entered here.

If you don't set this property, the file `index.jsp` will appear by default when the application is run.

As this file also contains `Hello World` as its default value, you might not notice that your RESTful web service isn't running, so just be aware of this default and the need to set this property, or update `index.jsp` to provide a link to the RESTful web service.

7. Right-click the project and select Deploy.

8. Right-click the project and select Run.

A browser window opens and displays the return value of `Hello, World!!`

See Also

For other sample applications that demonstrate deploying and running JAX-RS applications using NetBeans IDE, see The `rsvp` Example Application and *Your First Cup: An Introduction to the Java EE Platform* at http://download.oracle.com/javaee/6/firstcup/doc/.

You may also look at the tutorials on the NetBeans IDE tutorial site, such as the one titled "Getting Started with RESTful Web Services" at http://www.netbeans.org/kb/docs/websvc/rest.html.

This tutorial includes a section on creating a CRUD application from a database.

Create, read, update, and delete (CRUD) are the four basic functions of persistent storage and relational databases.

The `rsvp` Example Application

The `rsvp` example application, located in *tut-install*`/examples/jaxrs/rsvp`, allows invitees to an event to indicate whether they will attend.

The events, people invited to the event, and the responses to the invite are stored in a Java DB database using the Java Persistence API.

The JAX-RS resources in `rsvp` are exposed in a stateless session enterprise bean.

Components of the `rsvp` Example Application

The three enterprise beans in the `rsvp` example application are `rsvp.ejb.ConfigBean`, `rsvp.ejb.StatusBean`, and `rsvp.ejb.ResponseBean`.

`ConfigBean` is a singleton session bean that initializes the data in the database.

`StatusBean` exposes a JAX-RS resource for displaying the current status of all invitees to an event.

The URI path template is declared as follows:

```
@Path("/status/{eventId}/")
```

The URI path variable `eventId` is a `@PathParam` variable in the `getResponse` method, which responds to HTTP `GET` requests and has been annotated with `@GET`.

The `eventId` variable is used to look up all the current responses in the database for that particular event.

`ResponseBean` exposes a JAX-RS resource for setting an invitee's response to a particular event.

The URI path template for `ResponseBean` is declared as follows:

```
@Path("/{eventId}/{inviteId}")
```

Two URI path variables are declared in the path template: `eventId` and `inviteId`.

As in `StatusBean`, `eventId` is the unique ID for a particular event.

Each invitee to that event has a unique ID for the invitation, and that is the `inviteId`.

Both of these path variables are used in two JAX-RS methods in `ResponseBean`:

`getResponse` and `putResponse`.

The `getResponse` method responds to HTTP `GET` requests and displays the invitee's current response and a form to change the response.

An invitee who wants to change his or her response selects the new response and submits the form data, which is processed as an HTTP `PUT` request by the `putResponse` method.

One of the parameters to the `putResponse` method, the `userResponse` string, is annotated with `@FormParam("attendeeResponse")`.

The HTML form created by `getResponse` stores the changed response in the `select` list with an ID of `attendeeResponse`.

The annotation `@FormParam("attendeeResponse")` indicates that the value of the select response is extracted from the HTTP `PUT` request and stored as the `userResponse` string.

The `putResponse` method uses `userResponse`, `eventId`, and `inviteId` to update the invitee's response in the database.

The events, people, and responses in `rsvp` are encapsulated in Java Persistence API entities.

The `rsvp.entity.Event`, `rsvp.entity.Person`, and `rsvp.entity.Response` entities respectively represent events, invitees, and responses to an event.

The `rsvp.util.ResponseEnum` class declares an enumerated type that represents all the possible response statuses an invitee may have.

Running the `rsvp` Example Application

Both NetBeans IDE and Ant can be used to deploy and run the `rsvp` example application.

To Run the `rsvp` Example Application in NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   `tut-install/examples/jaxrs/`

3. Select the `rsvp` folder.

4. Select the Open as Main Project check box.

5. Click Open Project.

6. Right-click the `rsvp` project in the left pane and select Run.

The project will be compiled, assembled, and deployed to GlassFish Server.

A web browser window will open to `http://localhost:8080/rsvp`.

7. In the web browser window, click the Event Status link for the Duke's Birthday event.

You'll see the current invitees and their responses.

8. Click on the name of one of the invitees, select a response, and click Submit response; then click Back to event page.

The invitee's new status should now be displayed in the table of invitees and their response statuses.

To Run the `rsvp` Example Application Using Ant

Before You Begin

You must have started the Java DB database before running `rsvp`.

1. In a terminal window, go to:

   `tut-install/examples/jaxrs/rsvp`

2. Type the following command:

   `ant all`

This command builds, assembles, and deploys `rsvp` to GlassFish Server.

3. Open a web browser window to `http://localhost:8080/rsvp`.

4. In the web browser window, click the Event Status link for the Duke's Birthday event.

You'll see the current invitees and their responses.

5. Click on the name of one of the invitees, select a response, and click Submit response, then click Back to event page.

The invitee's new status should now be displayed in the table of invitees and their response statuses.

Real-World Examples

Most blog sites use RESTful web services.

These sites involve downloading XML files, in RSS or Atom format, that contain lists of links to other resources.

Other web sites and web applications that use REST-like developer interfaces to data include Twitter and Amazon S3 (Simple Storage Service).

With Amazon S3, buckets and objects can be created, listed, and retrieved using either a REST-style HTTP interface or a SOAP interface.

The examples that ship with Jersey include a storage service example with a RESTful interface.

The tutorial at http://netbeans.org/kb/docs/websvc/twitter-swing.html uses NetBeans IDE to create a simple, graphical, REST-based client that displays Twitter public timeline messages and lets you view and update your Twitter status.

Further Information about JAX-RS

For more information about RESTful web services and JAX-RS, see

- "RESTful Web Services vs. 'Big' Web Services: Making the Right Architectural Decision":

  http://www2008.org/papers/pdf/p805-pautassoA.pdf

- The Community Wiki for Project Jersey, the JAX-RS reference implementation:

  http://wikis.sun.com/display/Jersey/Main

- "Fielding Dissertation: Chapter 5: Representational State Transfer (REST)":

  http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

- *RESTful Web Services*, by Leonard Richardson and Sam Ruby, available from O'Reilly Media at http://oreilly.com/catalog/9780596529260/

- JSR 311: JAX-RS: The Java API for RESTful Web Services:

  http://jcp.org/en/jsr/detail?id=311

- JAX-RS project:

  http://jsr311.java.net/

- Jersey project:

  http://jersey.java.net/

- JAX-RS Overview document:

  http://wikis.sun.com/display/Jersey/Overview+of+JAX-RS+1.0+Features