◆ ◆ ◆ CHAPTER 20

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" on page 70. 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?

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" on page 385 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 Methods and Requests" on page 387 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 Methods and Requests" on page 387 and "Using Entity Providers to Map HTTP Response and Request Entity Bodies" on page 389 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" on page 389 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 20–1 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://docs.oracle.com/javaee/6/api/`.

TABLE 20–1 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.

TABLE 20–1 Summary of JAX-RS Annotations *(Continued)*

Annotation	Description
`@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 20–2 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 20–2 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 Methods and Requests

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://www.w3.org/Submission/wadl/ 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" on page 392. 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 20–3 shows the standard types that are supported automatically for HTTP request and response entity bodies. You need to write an entity provider only if you are not choosing one of these standard types.

TABLE 20–3 Types Supported for HTTP Request and Response Entity Bodies

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 = hh.getRequestHeaders();
    Map<String, Cookie> pathParams = hh.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 lang=\"en\"><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" on page 398 and *Your First Cup: An Introduction to the Java EE Platform* at `http://docs.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 the *tut-install*`/examples/jaxrs/rsvp/` directory, 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
- "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/

◆ ◆ ◆ CHAPTER 21

JAX-RS: Advanced Topics and Example

The Java API for RESTful Web Services (JAX-RS, defined in JSR 311) is designed to make it easy to develop applications that use the REST architecture. This chapter describes advanced features of JAX-RS. If you are new to JAX-RS, see Chapter 20, "Building RESTful Web Services with JAX-RS," before you proceed with this chapter.

JAX-RS is part of the Java EE 6 full profile. JAX-RS is integrated with Contexts and Dependency Injection for the Java EE Platform (CDI), Enterprise JavaBeans (EJB) technology, and Java Servlet technology.

The following topics are addressed here:

- "Annotations for Field and Bean Properties of Resource Classes" on page 403
- "Subresources and Runtime Resource Resolution" on page 407
- "Integrating JAX-RS with EJB Technology and CDI" on page 408
- "Conditional HTTP Requests" on page 409
- "Runtime Content Negotiation" on page 410
- "Using JAX-RS With JAXB" on page 412
- "The `customer` Example Application" on page 418

Annotations for Field and Bean Properties of Resource Classes

JAX-RS annotations for resource classes let you extract specific parts or values from a Uniform Resource Identifier (URI) or request header.

JAX-RS provides the annotations listed in Table 21–1.

TABLE 21–1 Advanced JAX-RS Annotations

Annotation	Description
`@Context`	Injects information into a class field, bean property, or method parameter

TABLE 21–1 Advanced JAX-RS Annotations *(Continued)*

Annotation	Description
`@CookieParam`	Extracts information from cookies declared in the cookie request header
`@FormParam`	Extracts information from a request representation whose content type is `application/x-www-form-urlencoded`
`@HeaderParam`	Extracts the value of a header
`@MatrixParam`	Extracts the value of a URI matrix parameter
`@PathParam`	Extracts the value of a URI template parameter
`@QueryParam`	Extracts the value of a URI query parameter

Extracting Path Parameters

URI path templates are URIs with variables embedded within the URI syntax. The `@PathParam` annotation lets you use variable URI path fragments when you call a method.

The following code snippet shows how to extract the last name of an employee when the employee's email address is provided:

```
@Path(/employees/"{firstname}.{lastname}@{domain}.com")
public class EmpResource {

    @GET
    @Produces("text/xml")
    public String getEmployeelastname(@PathParam("lastname") String lastName) {
     ...
    }
}
```

In this example, the `@Path` annotation defines the URI variables (or path parameters) `{firstname}` , `{lastname}`, and `{domain}`. The `@PathParam` in the method parameter of the request method extracts the last name from the email address.

If your HTTP request is GET `/employees/john.doe@example.com`, the value "`doe`" is injected into `{lastname}`.

You can specify several path parameters in one URI.

You can declare a regular expression with a URI variable. For example, if it is required that the last name must consist only of lower and upper case characters, you can declare the following regular expression:

```
@Path(/employees/{"firstname}.{lastname[a-zA-Z]*}@{domain}.com")
```

If the last name does not match the regular expression, a 404 response is returned.

Extracting Query Parameters

Use the `@QueryParam` annotation to extract query parameters from the query component of the request URI.

For instance, to query all employees who have joined within a specific range of years, use a method signature like the following:

```
@Path(/employees/")
@GET
public Response getEmployees(
        @DefaultValue("2002") @QueryParam("minyear") int minyear,
        @DefaultValue("2010") @QueryParam("maxyear") int maxyear)
    {...}
```

This code snippet defines two query parameters, `minyear` and `maxyear`. The following HTTP request would query for all employees who have joined between 1999 and 2009:

```
GET /employees?maxyear=2009&minyear=1999
```

The `@DefaultValue` annotation defines a default value, which is to be used if no values are provided for the query parameters. By default, JAX-RS assigns a null value for `Object` values and zero for primitive data types. You can use the `@DefaultValue` annotation to eliminate null or zero values and define your own default values for a parameter.

Extracting Form Data

Use the `@FormParam` annotation to extract form parameters from HTML forms. For example, the following form accepts the name, address, and manager's name of an employee:

```
<FORM action="http://example.com/employees/" method="post">
<p>
<fieldset>
Employee name: <INPUT type="text" name="empname" tabindex="1">
Employee address: <INPUT type="text" name="empaddress" tabindex="2">
Manager name: <INPUT type="text" name="managername" tabindex="3">
</fieldset>
</p>
</FORM>
```

Use the following code snippet to extract the manager name from this HTML form:

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(@FormParam("managername") String managername) {
    // Store the value
    ...
}
```

To obtain a map of form parameter names to values, use a code snippet like the following:

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(MultivaluedMap<String. String> formParams) {
    // Store the message
}
```

Extracting the Java Type of a Request or Response

The `javax.ws.rs.core.Context` annotation retrieves the Java types related to a request or response.

The `javax.ws.rs.core.UriInfo` interface provides information about the components of a request URI. The following code snippet shows how to obtain a map of query and path parameter names to values:

```
@GET
public String getParams(@Context UriInfo ui) {
    MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
    MultivaluedMap<String, String> pathParams = ui.getPathParameters();
}
```

The `javax.ws.rs.core.HttpHeaders` interface provides information about request headers and cookies. The following code snippet shows how to obtain a map of header and cookie parameter names to values:

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

Subresources and Runtime Resource Resolution

You can use a resource class to process only a part of the URI request. A root resource can then implement subresources that can process the remainder of the URI path.

A resource class method that is annotated with `@Path` is either a subresource method or a subresource locator:

- A subresource method is used to handle requests on a subresource of the corresponding resource.
- A subresource locator is used to locate subresources of the corresponding resource.

Subresource Methods

A *subresource method* handles an HTTP request directly. The method must be annotated with a request method designator such as `@GET` or `@POST`, in addition to `@Path`. The method is invoked for request URIs that match a URI template created by concatenating the URI template of the resource class with the URI template of the method.

The following code snippet shows how a subresource method can be used to extract the last name of an employee when the employee's email address is provided:

```
@Path("/employeeinfo")
Public class EmployeeInfo {

    public employeeinfo() {}

    @GET
    @Path("/employees/{firstname}.{lastname}@{domain}.com")
    @Produces("text/xml")
    public String getEmployeeLastName(@PathParam("lastname") String lastName) {
       ...
    }

}
```

The `getEmployeeLastName` method returns `doe` for the following GET request:

```
GET /employeeinfo/employees/john.doe@example.com
```

Subresource Locators

A *subresource locator* returns an object that will handle an HTTP request. The method must not be annotated with a request method designator. You must declare a subresource locator within a subresource class, and only subresource locators are used for runtime resource resolution.

The following code snippet shows a subresource locator:

```
// Root resource class
@Path("/employeeinfo")
public class EmployeeInfo {

    // Subresource locator: obtains the subresource Employee
    // from the path /employeeinfo/employees/{empid}
    @Path("/employees/{empid}")
    public Employee getEmployee(@PathParam("empid") String id) {
        // Find the Employee based on the id path parameter
        Employee emp = ...;
        ...
        return emp;
    }
}

// Subresource class
public class Employee {

    // Subresource method: returns the employee’s last name
    @GET
    @Path("/lastname")
    public String getEmployeeLastName() {
        ...
        return lastName
    }
}
```

In this code snippet, the `getEmployee` method is the subresource locator that provides the `Employee` object, which services requests for `lastname`.

If your HTTP request is `GET /employeeinfo/employees/as209/`, the `getEmployee` method returns an `Employee` object whose id is `as209`. At runtime, JAX-RS sends a `GET /employeeinfo/employees/as209/lastname` request to the `getEmployeeLastName` method . The `getEmployeeLastName` method retrieves and returns the last name of the employee whose id is `as209`.

Integrating JAX-RS with EJB Technology and CDI

JAX-RS works with Enterprise JavaBeans technology (enterprise beans) and Contexts and Dependency Injection for the Java EE Platform (CDI).

In general, for JAX-RS to work with enterprise beans, you need to annotate the class of a bean with `@Path` to convert it to a root resource class. You can use the `@Path` annotation with stateless session beans and singleton POJO beans.

The following code snippet shows a stateless session bean and a singleton bean that have been converted to JAX-RS root resource classes.

```
@Stateless
@Path("stateless-bean")
public class StatelessResource {...}
```

```
@Singleton
@Path("singleton-bean")
public class SingletonResource {...}
```

Session beans can also be used for subresources.

JAX-RS and CDI have slightly different component models. By default, JAX-RS root resource classes are managed in the request scope, and no annotations are required for specifying the scope. CDI managed beans annotated with `@RequestScoped` or `@ApplicationScoped` can be converted to JAX-RS resource classes.

The following code snippet shows a JAX-RS resource class.

```
@Path("/employee/{id}")
public class Employee {
    public Employee(@PathParam("id") String id) {...}
}

@Path("{lastname}")
public final class EmpDetails {...}
```

The following code snippet shows this JAX-RS resource class converted to a CDI bean. The beans must be proxyable, so the `Employee` class requires a non-private constructor with no parameters, and the `EmpDetails` class must not be `final`.

```
@Path("/employee/{id}")
@RequestScoped
public class Employee {
    public Employee() {...}

    @Inject
    public Employee(@PathParam("id") String id) {...}
}

@Path("{lastname}")
@RequestScoped
public class EmpDetails {...}
```

Conditional HTTP Requests

JAX-RS provides support for conditional GET and PUT HTTP requests. Conditional GET requests help save bandwidth by improving the efficiency of client processing.

A GET request can return a Not Modified (304) response if the representation has not changed since the previous request. For example, a web site can return 304 responses for all its static images that have not changed since the previous request.

A PUT request can return a Precondition Failed (412) response if the representation has been modified since the last request. The conditional PUT can help avoid the lost update problem.

Conditional HTTP requests can be used with the `Last-Modified` and `ETag` headers. The `Last-Modified` header can represent dates with granularity of one second.

```
@Path("/employee/{joiningdate}")
public class Employee {

    Date joiningdate;

    @GET
    @Produces("application/xml")
    public Employee(@PathParam("joiningdate") Date joiningdate,
            @Context Request req,
            @Context UriInfo ui) {

        this.joiningdate = joiningdate;
        ...
        this.tag = computeEntityTag(ui.getRequestUri());
        if (req.getMethod().equals("GET")) {
            Response.ResponseBuilder rb = req.evaluatePreconditions(tag);
            if (rb != null) {
                throw new WebApplicationException(rb.build());
            }
        }
    }
}
```

In this code snippet, the constructor of the `Employee` class computes the entity tag from the request URI and calls the `request.evaluatePreconditions` method with that tag. If a client request returns an `If-none-match` header with a value that has the same entity tag that was computed, `evaluate.Preconditions` returns a pre-filled-out response with a 304 status code and an entity tag set that may be built and returned.

Runtime Content Negotiation

The `@Produces` and `@Consumes` annotations handle static content negotiation in JAX-RS. These annotations specify the content preferences of the server. HTTP headers such as `Accept`, `Content-Type`, and `Accept-Language` define the content negotiation preferences of the client.

For more details on the HTTP headers for content negotiation, see HTTP /1.1 - Content Negotiation (`http://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html`).

The following code snippet shows the server content preferences:

```
@Produces("text/plain")
@Path("/employee")
public class Employee {

    @GET
    public String getEmployeeAddressText(String address) { ... }

    @Produces("text/xml")
    @GET
    public String getEmployeeAddressXml(Address address) { ... }
}
```

The `getEmployeeAddressText` method is called for an HTTP request that looks as follows:

```
GET /employee
Accept: text/plain
```

This will produce the following response:

```
500 Oracle Parkway, Redwood Shores, CA
```

The `getEmployeeAddressXml` method is called for an HTTP request that looks as follows:

```
GET /employee
Accept: text/xml
```

This will produce the following response:

```
<address street="500 Oracle Parkway, Redwood Shores, CA" country="USA"/>
```

With static content negotiation, you can also define multiple content and media types for the client and server.

```
@Produces("text/plain", "text/xml")
```

In addition to supporting static content negotiation, JAX-RS also supports runtime content negotiation using the `javax.ws.rs.core.Variant` class and `Request` objects. The `Variant` class specifies the resource representation of content negotiation. Each instance of the `Variant` class may contain a media type, a language, and an encoding. The `Variant` object defines the resource representation that is supported by the server. The `Variant.VariantListBuilder` class is used to build a list of representation variants.

The following code snippet shows how to create a list of resource representation variants:

```
List<Variant> vs =
    Variant.mediatypes("application/xml", "application/json")
           .languages("en", "fr").build();
```

This code snippet calls the `build` method of the `VariantListBuilder` class. The `VariantListBuilder` class is invoked when you call the `mediatypes`, `languages`, or `encodings` methods. The `build` method builds a series of resource representations. The `Variant` list created by the `build` method has all possible combinations of items specified in the `mediatypes`, `languages`, and `encodings` methods.

In this example, the size of the vs object as defined in this code snippet is 4, and the contents are as follows:

```
[["application/xml","en"], ["application/json","en"],
    ["application/xml","fr"],["application/json","fr"]]
```

The `javax.ws.rs.core.Request.selectVariant` method accepts a list of `Variant` objects and chooses the `Variant` object that matches the HTTP request. This method compares its list of `Variant` objects with the `Accept`, `Accept-Encoding`, `Accept-Language`, and `Accept-Charset` headers of the HTTP request.

The following code snippet shows how to use the `selectVariant` method to select the most acceptable `Variant` from the values in the client request.

```
@GET
public Response get(@Context Request r) {
    List<Variant> vs = ...;
    Variant v = r.selectVariant(vs);
    if (v == null) {
        return Response.notAcceptable(vs).build();
    } else {
        Object rep = selectRepresentation(v);
        return Response.ok(rep, v);
    }
}
```

The `selectVariant` method returns the `Variant` object that matches the request, or null if no matches are found. In this code snippet, if the method returns null, a `Response` object for a non-acceptable response is built. Otherwise, a `Response` object with an OK status and containing a representation in the form of an `Object` entity and a `Variant` is returned.

Using JAX-RS With JAXB

Java Architecture for XML Binding (JAXB) is an XML-to-Java binding technology that simplifies the development of web services by enabling transformations between schema and Java objects and between XML instance documents and Java object instances. An XML schema defines the data elements and structure of an XML document. You can use JAXB APIs and tools to establish mappings between Java classes and XML schema. JAXB technology provides the tools that enable you to convert your XML documents to and from Java objects.

By using JAXB, you can manipulate data objects in the following ways:

- You can start with an XML schema definition (XSD) and use `xjc`, the JAXB schema compiler tool, to create a set of JAXB-annotated Java classes that map to the elements and types defined in the XSD schema.
- You can start with a set of Java classes and use `schemagen`, the JAXB schema generator tool, to generate an XML schema.

- Once a mapping between the XML schema and the Java classes exists, you can use the JAXB binding runtime to marshal and unmarshal your XML documents to and from Java objects and use the resulting Java classes to assemble a web services application.

XML is a common media format that RESTful services consume and produce. To deserialize and serialize XML, you can represent requests and responses by JAXB annotated objects. Your JAX-RS application can use the JAXB objects to manipulate XML data. JAXB objects can be used as request entity parameters and response entities. The JAX-RS runtime environment includes standard `MessageBodyReader` and `MessageBodyWriter` provider interfaces for reading and writing JAXB objects as entities.

With JAX-RS, you enable access to your services by publishing resources. Resources are just simple Java classes with some additional JAX-RS annotations. These annotations express the following:

- The path of the resource (the URL you use to access it)
- The HTTP method you use to call a certain method (for example, the GET or POST method)
- The MIME type with which a method accepts or responds

As you define the resources for your application, consider the type of data you want to expose. You may already have a relational database that contains information you want to expose to users, or you may have static content that does not reside in a database but does need to be distributed as resources. Using JAX-RS, you can distribute content from multiple sources. RESTful web services can use various types of input/output formats for request and response. The `customer` example, described in "The `customer` Example Application" on page 418, uses XML.

Resources have representations. A resource representation is the content in the HTTP message that is sent to, or returned from, the resource using the URI. Each representation a resource supports has a corresponding media type. For example, if a resource is going to return content formatted as XML, you can use `application/xml` as the associated media type in the HTTP message. Depending on the requirements of your application, resources can return representations in a preferred single format or in multiple formats. JAX-RS provides `@Consumes` and `@Produces` annotations to declare the media types that are acceptable for a resource method to read and write.

JAX-RS also maps Java types to and from resource representations using entity providers. A `MessageBodyReader` entity provider reads a request entity and deserializes the request entity into a Java type. A `MessageBodyWriter` entity provider serializes from a Java type into a response entity. For example, if a `String` value is used as the request entity parameter, the `MessageBodyReader` entity provider deserializes the request body into a new `String`. If a JAXB type is used as the return type on a resource method, the `MessageBodyWriter` serializes the JAXB object into a response body.

By default, the JAX-RS runtime environment attempts to create and use a default `JAXBContext` class for JAXB classes. However, if the default `JAXBContext` class is not suitable, then you can supply a `JAXBContext` class for the application using a JAX-RS `ContextResolver` provider interface.

The following sections explain how to use JAXB with JAX-RS resource methods.

Using Java Objects to Model Your Data

If you do not have an XML schema definition for the data you want to expose, you can model your data as Java classes, add JAXB annotations to these classes, and use JAXB to generate an XML schema for your data. For example, if the data you want to expose is a collection of products and each product has an ID, a name, a description, and a price, you can model it as a Java class as follows:

```
@XmlRootElement(name="product")
@XmlAccessorType(XmlAccessType.FIELD)
public class Product {

    @XmlElement(required=true)
    protected int id;
    @XmlElement(required=true)
    protected String name;
    @XmlElement(required=true)
    protected String description;
    @XmlElement(required=true)
    protected int price;

    public Product() {}

    // Getter and setter methods
    // ...
}
```

Run the JAXB schema generator on the command line to generate the corresponding XML schema definition:

```
schemagen Product.java
```

This command produces the XML schema as an `.xsd` file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="product" type="product"/>

  <xs:complexType name="product">
    <xs:sequence>
      <xs:element name="id" type="xs:int"/>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="description" type="xs:string"/>
      <xs:element name="price" type="xs:int"/>
    </xs:sequence>
  <xs:complexType>
</xs:schema>
```

Once you have this mapping, you can create `Product` objects in your application, return them, and use them as parameters in JAX-RS resource methods. The JAX-RS runtime uses JAXB to convert the XML data from the request into a `Product` object and to convert a `Product` object into XML data for the response. The following resource class provides a simple example:

```
@Path("/product")
public class ProductService {
    @GET
    @Path("/get")
    @Produces("application/xml")
    public Product getProduct() {
        Product prod = new Product();
        prod.setId(1);
        prod.setName("Mattress");
        prod.setDescription("Queen size mattress");
        prod.setPrice(500);
        return prod;
    }

    @POST
    @Path("/create")
    @Consumes("application/xml")
    public Response createProduct(Product prod) {
        // Process or store the product and return a response
        // ...
    }
}
```

Some IDEs, such as NetBeans IDE, will run the schema generator tool automatically during the build process if you add Java classes that have JAXB annotations to your project. For a detailed example, see "The `customer` Example Application" on page 418. The `customer` example contains a more complex relationship between the Java classes that model the data, which results in a more hierarchical XML representation.

Starting from an Existing XML Schema Definition

If you already have an XML schema definition in an `.xsd` file for the data you want to expose, use the JAXB schema compiler tool. Consider this simple example of an `.xsd` file:

```
<?xml version="1.0"?>
<xs:schema targetNamespace="http://xml.product"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           xmlns:myco="http://xml.product">

  <xs:element name="product" type="myco:Product"/>
  <xs:complexType name="Product">
    <xs:sequence>
      <xs:element name="id" type="xs:int"/>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="description" type="xs:string"/>
      <xs:element name="price" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Run the schema compiler tool on the command line as follows:

```
xjc Product.xsd
```

This command generates the source code for Java classes that correspond to the types defined in the `.xsd` file. The schema compiler tool generates a Java class for each `complexType` defined in the `.xsd` file. The fields of each generated Java class are the same as the elements inside the corresponding `complexType`, and the class contains getter and setter methods for these fields.

In this case the schema compiler tool generates the classes `product.xml.Product` and `product.xml.ObjectFactory`. The `Product` class contains JAXB annotations, and its fields correspond to those in the `.xsd` definition:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Product", propOrder = {
    "id",
    "name",
    "description",
    "price"
})
public class Product {
    protected int id;
    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true)
    protected String description;
    protected int price;

    // Setter and getter methods
    // ...
}
```

You can create instances of the `Product` class from your application (for example, from a database). The generated class `product.xml.ObjectFactory` contains a method that allows you to convert these objects to JAXB elements that can be returned as XML inside JAX-RS resource methods:

```
@XmlElementDecl(namespace = "http://xml.product", name = "product")
public JAXBElement<Product> createProduct(Product value) {
  return new JAXBElement<Product>(_Product_QNAME, Product.class, null, value);
}
```

The following code shows how to use the generated classes to return a JAXB element as XML in a JAX-RS resource method:

```
@Path("/product")
public class ProductService {
    @GET
    @Path("/get")
    @Produces("application/xml")
    public JAXBElement<Product> getProduct() {
        Product prod = new Product();
        prod.setId(1);
        prod.setName("Mattress");
        prod.setDescription("Queen size mattress");
        prod.setPrice(500);
        return new ObjectFactory().createProduct(prod);
    }
}
```

For `@POST` and `@PUT` resource methods, you can use a `Product` object directly as a parameter. JAX-RS maps the XML data from the request into a `Product` object.

```
@Path("/product")
public class ProductService {
    @GET
    // ...

    @POST
    @Path("/create")
    @Consumes("application/xml")
    public Response createProduct(Product prod) {
        // Process or store the product and return a response
        // ...
    }
}
```

Some IDEs, such as NetBeans IDE, will run the schema compiler tool automatically during the build process if you add an `.xsd` file to your project sources. For a detailed example, see "Modifying the Example to Generate Entity Classes from an Existing Schema" on page 426. The modified `customer` example contains a more hierarchical XML schema definition, which results in a more complex relationship between the Java classes that model the data.

Using JSON with JAX-RS and JAXB

JAX-RS can automatically read and write XML using JAXB, but it can also work with JSON data. JSON is a simple text-based format for data exchange derived from JavaScript. For the examples above, the XML representation of a product is:

```
<?xml version="1.0" encoding="UTF-8"?>
<product>
  <id>1</id>
  <name>Mattress</name>
  <description>Queen size mattress</description>
  <price>500</price>
</product>
```

The equivalent JSON representation is:

```
{
    "id":"1",
    "name":"Mattress",
    "description":"Queen size mattress",
    "price":500
}
```

You can add the format `application/json` to the `@Produces` annotation in resource methods to produce responses with JSON data:

```
@GET
@Path("/get")
@Produces({"application/xml","application/json"})
public Product getProduct() { ... }
```

In this example the default response is XML, but the response is a JSON object if the client makes a `GET` request that includes this header:

```
Accept: application/json
```

The resource methods can also accept JSON data for JAXB annotated classes:

```
@POST
@Path("/create")
@Consumes({"application/xml","application/json"})
public Response createProduct(Product prod) { ... }
```

The client should include the following header when submitting JSON data with a `POST` request:

```
Content-Type: application/json
```

The `customer` Example Application

This section describes how to build and run the `customer` sample application. This example application is a RESTful web service that uses JAXB to perform the Create, Read, Update, Delete (CRUD) operations for a specific entity.

The `customer` sample application is in the *tut-install*`/examples/jaxrs/customer/` directory. See Chapter 2, "Using the Tutorial Examples," for basic information on building and running sample applications.

Overview of the `customer` Example Application

The source files of this application are at *tut-install*`/examples/jaxrs/customer/src/java/`. The application has three parts:

- The `Customer` and `Address` entity classes. These classes model the data of the application and contain JAXB annotations. See "The `Customer` and `Address` Entity Classes" on page 419 for details.
- The `CustomerService` resource class. This class contains JAX-RS resource methods that perform operations on `Customer` instances represented as XML or JSON data using JAXB. See "The `CustomerService` Class" on page 422 for details.
- The `CustomerClientXML` and `CustomerClientJSON` client classes. These classes test the resource methods of the web service using XML and JSON representations of `Customer` instances. See "The `CustomerClientXML` and `CustomerClientJSON` Classes" on page 424 for details.

The `customer` sample application shows you how to model your data entities as Java classes with JAXB annotations. The JAXB schema generator produces an equivalent XML schema definition file (`.xsd`) for your entity classes. The resulting schema is used to automatically marshal and unmarshal entity instances to and from XML or JSON in the JAX-RS resource methods.

In some cases you may already have an XML schema definition for your entities. See "Modifying the Example to Generate Entity Classes from an Existing Schema" on page 426 for instructions on how to modify the `customer` example to model your data starting from an `.xsd` file and using JAXB to generate the equivalent Java classes.

The `Customer` and `Address` Entity Classes

The following class represents a customer's address:

```
@XmlRootElement(name="address")
@XmlAccessorType(XmlAccessType.FIELD)
public class Address {

    @XmlElement(required=true)
    protected int number;

    @XmlElement(required=true)
    protected String street;

    @XmlElement(required=true)
    protected String city;

    @XmlElement(required=true)
    protected String state;
    @XmlElement(required=true)
    protected String zip;

    @XmlElement(required=true)
    protected String country;

    public Address() { }

    // Getter and setter methods
    // ...
}
```

The `@XmlRootElement(name="address")` annotation maps this class to the `address` XML element. The `@XmlAccessorType(XmlAccessType.FIELD)` annotation specifies that all the fields of this class are bound to XML by default. The `@XmlElement(required=true)` annotation specifies that an element must be present in the XML representation.

The following class represents a customer:

```
@XmlRootElement(name="customer")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {

    @XmlAttribute(required=true)
    protected int id;

    @XmlElement(required=true)
    protected String firstname;

    @XmlElement(required=true)
    protected String lastname;

    @XmlElement(required=true)
    protected Address address;

    @XmlElement(required=true)
    protected String email;

    @XmlElement (required=true)
    protected String phone;


    public Customer() { }

    // Getter and setter methods
    // ...
}
```

The `Customer` class contains the same JAXB annotations as the previous class, except for the `@XmlAttribute(required=true)` annotation, which maps a property to an attribute of the XML element representing the class.

The `Customer` class contains a property whose type is another entity, the `Address` class. This mechanism allows you to define in Java code the hierarchical relationships between entities without having to write an `.xsd` file yourself.

JAXB generates the following XML schema definition for the two classes above:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="address" type="address"/>
  <xs:element name="customer" type="customer"/>

  <xs:complexType name="address">
    <xs:sequence>
      <xs:element name="number" type="xs:int"/>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="state" type="xs:string"/>
      <xs:element name="zip" type="xs:string"/>
      <xs:element name="country" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="customer">
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
      <xs:element ref="address"/>
      <xs:element name="email" type="xs:string"/>
      <xs:element name="phone" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:int" use="required"/>
  </xs:complexType>
</xs:schema>
```

The file sample-input.xml in the top-level directory of the project contains an example of an XML representation of a customer:

```
<?xml version="1.0" encoding="UTF-8"?>
<customer id="1">
  <firstname>Duke</firstname>
  <lastname>OfJava</lastname>
  <address>
    <number>1</number>
    <street>Duke’s Way</street>
    <city>JavaTown</city>
    <state>JA</state>
    <zip>12345</zip>
    <country>USA</country>
  </address>
  <email>duke@example.com</email>
  <phone>123-456-7890</phone>
</customer>
```

The file sample-input.json contains an example of a JSON representation of a customer:

```
{
    "@id": "1",
    "firstname": "Duke",
    "lastname": "OfJava",
    "address": {
        "number": 1,
        "street": "Duke’s Way",
        "city": "JavaTown",
        "state": "JA",
        "zip": "12345",
        "country": "USA"
    },
    "email": "duke@example.com",
    "phone": "123-456-7890"
}
```

The CustomerService Class

The `CustomerService` class has a `createCustomer` method that creates a customer resource based on the `Customer` class and returns a URI for the new resource. The `persist` method emulates the behavior of the JPA entity manager. This example uses a `java.util.Properties` file to store data. If you are using the default configuration of GlassFish Server, the properties file is at *domain-dir*/`CustomerDATA.txt`.

```
@Path("/Customer")
public class CustomerService {
    public static final String DATA_STORE = "CustomerDATA.txt";
    public static final Logger logger =
            Logger.getLogger(CustomerService.class.getCanonicalName());
    ...

    @POST
    @Consumes({"application/xml", "application/json"})
    public Response createCustomer(Customer customer) {
        try {
            long customerId = persist(customer);
            return Response.created(URI.create("/" + customerId)).build();
        } catch (Exception e) {
          throw new WebApplicationException(e,
                  Response.Status.INTERNAL_SERVER_ERROR);
        }
    }
    ...

    private long persist(Customer customer) throws IOException {

        File dataFile = new File(DATA_STORE);

        if (!dataFile.exists()) {
            dataFile.createNewFile();
        }

        long customerId = customer.getId();
        Address address = customer.getAddress();

        Properties properties = new Properties();
        properties.load(new FileInputStream(dataFile));
        properties.setProperty(String.valueOf(customerId),
                customer.getFirstname() + ","
                + customer.getLastname() + ","
                + address.getNumber() + ","
                + address.getStreet() + ","
                + address.getCity() + ","
                + address.getState() + ","
                + address.getZip() + ","
                + address.getCountry() + ","
                + customer.getEmail() + ","
                + customer.getPhone());

        properties.store(new FileOutputStream(DATA_STORE),null);

        return customerId;
    }
    ...
}
```

The response returned to the client has a URI to the newly created resource. The return type is an entity body mapped from the property of the response with the status code specified by the status property of the response. The `WebApplicationException` is a `RuntimeException` that is used to wrap the appropriate HTTP error status code, such as 404, 406, 415, or 500.

The `@Consumes({"application/xml","application/json"})` and `@Produces({"application/xml","application/json"})` annotations set the request and response media types to use the appropriate MIME client. These annotations can be applied to a resource method, a resource class, or even an entity provider. If you do not use these annotations, JAX-RS allows the use of any media type (`"*/*"`).

The following code snippet shows the implementation of the `getCustomer` and `findbyId` methods. The `getCustomer` method uses the `@Produces` annotation and returns a `Customer` object, which is converted to an XML or JSON representation depending on the `Accept:` header specified by the client.

```
@GET
@Path("{id}")
@Produces({"application/xml", "application/json"})
public Customer getCustomer(@PathParam("id") String customerId) {
    Customer customer = null;

    try {
        customer = findById(customerId);
    } catch (Exception ex) {
        logger.log(Level.SEVERE,
                "Error calling searchCustomer() for customerId {0}. {1}",
                new Object[]{customerId, ex.getMessage()});
    }
    return customer;
}

private Customer findById(String customerId) throws IOException {
    properties properties = new Properties();
    properties.load(new FileInputStream(DATA_STORE));
    String rawData = properties.getProperty(customerId);

    if (rawData != null) {
        final String[] field = rawData.split(",");

        Address address = new Address();
        Customer customer = new Customer();
        customer.setId(Integer.parseInt(customerId));
        customer.setAddress(address);

        customer.setFirstname(field[0]);
        customer.setLastname(field[1]);
        address.setNumber(Integer.parseInt(field[2]));
        address.setStreet(field[3]);
        address.setCity(field[4]);
        address.setState(field[5]);
        address.setZip(field[6]);
        address.setCountry(field[7]);
        customer.setEmail(field[8]);
        customer.setPhone(field[9]);

        return customer;
    }
    return null;
}
```

The `CustomerClientXML` and `CustomerClientJSON` Classes

Jersey is the reference implementation of JAX-RS (JSR 311). You can use the Jersey client API to write a test client for the `customer` example application. You can find the Jersey APIs at `http://jersey.java.net/nonav/apidocs/latest/jersey/`.

The `CustomerClientXML` class calls Jersey APIs to test the `CustomerService` web service:

```
package customer.rest.client;

import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;
import customer.data.Address;
import customer.data.Customer;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ws.rs.core.MediaType;

public class CustomerClientXML {
    public static final Logger logger =
            Logger.getLogger(CustomerClientXML.class.getCanonicalName());

    public static void main(String[] args) {

        Client client = Client.create();
        // Define the URL for testing the example application
```

```
WebResource webResource =
       client.resource("http://localhost:8080/customer/rest/Customer");

// Test the POST method
Customer customer = new Customer();
Address address = new Address();
customer.setAddress(address);

customer.setId(1);
customer.setFirstname("Duke");
customer.setLastname("OfJava");
address.setNumber(1);
address.setStreet("Duke’s Drive");
address.setCity("JavaTown");
address.setZip("1234");
address.setState("JA");
address.setCountry("USA");
customer.setEmail("duke@java.net");
customer.setPhone("12341234");

ClientResponse response =
        webResource.type("application/xml").post(ClientResponse.class,
        customer);

logger.info("POST status: {0}" + response.getStatus());
if (response.getStatus() == 201) {
    logger.info("POST succeeded");
} else {
    logger.info("POST failed");
}

// Test the GET method using content negotiation
response = webResource.path("1").accept(MediaType.APPLICATION_XML)
        .get(ClientResponse.class);
Customer entity = response.getEntity(Customer.class);

logger.log(Level.INFO, "GET status: {0}", response.getStatus());
if (response.getStatus() == 200) {
    logger.log(Level.INFO, "GET succeeded, city is {0}",
            entity.getAddress().getCity());
} else {
    logger.info("GET failed");
}

// Test the DELETE method
response = webResource.path("1").delete(ClientResponse.class);

logger.log(Level.INFO, "DELETE status: {0}", response.getStatus());
if (response.getStatus() == 204) {
    logger.info("DELETE succeeded (no content)");
} else {
    logger.info("DELETE failed");
}

response = webResource.path("1").accept(MediaType.APPLICATION_XML)
        .get(ClientResponse.class);
logger.log(Level.INFO, "GET status: {0}", response.getStatus());
if (response.getStatus() == 204) {
    logger.info("After DELETE, the GET request returned no content.");
```

```
        } else {
            logger.info("Failed, after DELETE, GET returned a response.");
        }
    }
}
```

This Jersey client tests the POST, GET, and DELETE methods using XML representations.

All of these HTTP status codes indicate success: 201 for POST, 200 for GET, and 204 for DELETE. For details about the meanings of HTTP status codes, see `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`.

The `CustomerClientJSON` class is similar to `CustomerClientXML` but it uses JSON representations to test the web service. In the `CustomerClientJSON` class `"application/xml"` is replaced by `"application/json"`, and `MediaType.APPLICATION_XML` is replaced by `MediaType.APPLICATION_JSON`.

Modifying the Example to Generate Entity Classes from an Existing Schema

This section describes how you can modify the `customer` example if you provide an XML schema definition file for your entities instead of providing Java classes. In this case JAXB generates the equivalent Java entity classes from the schema definition.

For the `customer` example you provide the following `.xsd` file:

```
<?xml version="1.0"?>
<xs:schema targetNamespace="http://xml.customer"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
    xmlns:ora="http://xml.customer">

  <xs:element name="customer" type="ora:Customer"/>

  <xs:complexType name="Address">
    <xs:sequence>
      <xs:element name="number" type="xs:int"/>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="state" type="xs:string"/>
      <xs:element name="zip" type="xs:string"/>
      <xs:element name="country" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Customer">
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
      <xs:element name="address" type="ora:Address"/>
      <xs:element name="email" type="xs:string"/>
      <xs:element name="phone" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:int" use="required"/>
  </xs:complexType>
</xs:schema>
```

You can modify the customer example as follows:

▼ To Modify the customer Example to Generate Java Entity Classes from an Existing XML Schema Definition

1 Create a JAXB binding to generate the entity Java classes from the schema definition. For example, in NetBeans IDE, follow these steps:

  a. Right click on the `customer` project and select New > Other...

  b. Under the XML folder, select JAXB Binding and click Next.

  c. In the Binding Name field, type `CustomerBinding`.

  d. Click Browse and choose the `.xsd` file from your file system.

  e. In the Package Name field, type `customer.xml`.

  f. Click Finish.

This procedure creates the `Customer` class, the `Address` class, and some JAXB auxiliary classes in the package `customer.xml`.

2 Modify the `CustomerService` class as follows:

  a. Replace the `customer.data.*` imports with `customer.xml.*` imports and import the `JAXBElement` and `ObjectFactory` classes:

```
import customer.xml.Customer;
import customer.xml.Address;
import customer.xml.ObjectFactory;
import javax.xml.bind.JAXBElement;
```

  b. Replace the return type of the `getCustomer` method:

```
public JAXBElement<Customer> getCustomer(
        @PathParam("id") String customerId) {
    ...
    return new ObjectFactory().createCustomer(customer);
}
```

3 **Modify the `CustomerClientXML` and `CustomerClientJSON` classes as follows:**

a. **Replace the `customer.data.*` imports with `customer.xml.*` imports and import the `JAXBElement` and `ObjectFactory` classes:**

```
import customer.xml.Address;
import customer.xml.Customer;
import customer.xml.ObjectFactory;
import javax.xml.bind.JAXBElement;
```

b. **Create an `ObjectFactory` instance and a `JAXBElement<Customer>` instance at the beginning of the `main` method:**

```
public static void main(String[] args) {
    Client client = Client.create();
    ObjectFactory factory = new ObjectFactory();
    WebResource webResource = ...;
    ...
    customer.setPhone("12341234");
    JAXBElement<Customer> customerJAXB = factory.createCustomer(customer);
    ClientResponse response = webResource.type("application/xml")
            .post(ClientResponse.class, customerJAXB);
    ...
}
```

c. **Modify the `GET` request after testing the `DELETE` method:**

```
response = webResource.path("1").accept(MediaType.APPLICATION_XML)
        .get(ClientResponse.class);
entity = response.getEntity(Customer.class);
logger.log(Level.INFO, "GET status: {0}", response.getStatus());
try {
    logger.info(entity.getAddress().getCity());
} catch (NullPointerException ne) {
    // null after deleting the only customer
    logger.log(Level.INFO, "After DELETE, city is: {0}", ne.getCause());
}
```

The instructions for building, deploying, and running the example are the same for the original `customer` example and for the modified version using this procedure.

Running the `customer` Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the `customer` application.

▼ To Build, Package, and Deploy the `customer` Example Using NetBeans IDE

This procedure builds the application into the *tut-install*`/examples/jax-rs/customer/build/web/` directory. The contents of this directory are deployed to the GlassFish Server.

1 **From the File menu, choose Open Project.**

2 **In the Open Project dialog, navigate to:**

tut-install`/examples/jaxrs/`

3 **Select the `customer` folder.**

4 **Select the Open as Main Project check box.**

5 **Click Open Project.**

It may appear that there are errors in the source files, because the files refer to JAXB classes that will be generated when you build the application. You can ignore these errors.

6 **In the Projects tab, right-click the `customer` project and select Deploy.**

▼ To Build, Package, and Deploy the `customer` Example Using Ant

1 **In a terminal window, go to:**

tut-install`/examples/jaxrs/customer/`

2 **Type the following command:**

```
ant
```

This command calls the `default` target, which builds and packages the application into a WAR file, `customer.war`, located in the `dist` directory.

3 **Type the following command:**

```
ant deploy
```

Typing this command deploys `customer.war` to the GlassFish Server.

▼ To Run the `customer` Example Using the Jersey Client

1 **In NetBeans IDE, expand the Source Packages node.**

2 **Expand the `customer.rest.client` node.**

3 **Right-click the `CustomerClientXML.java` file and select Run File.**

The output of the client looks like this:

```
run:
Jun 12, 2012 2:40:20 PM customer.rest.client.CustomerClientXML main
INFO: POST status: 201
Jun 12, 2012 2:40:20 PM customer.rest.client.CustomerClientXML main
INFO: POST succeeded
Jun 12, 2012 2:40:20 PM customer.rest.client.CustomerClientXML main
```