Advanced JAX-RS Features

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 19, 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
- Subresources and Runtime Resource Resolution
- Integrating JAX-RS with EJB Technology and CDI
- Conditional HTTP Requests
- Runtime Content Negotiation
- Using JAX-RS With JAXB

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 the following table.

Table 20-1 Advanced JAX-RS Annotations

Annotation	Description
`@Context`	Injects information into a class field, bean property, or method parameter
`@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@mycompany.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 only consist 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 a 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 (requests on the subresource are then handled by something else ...).

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@oracle.com
```

Subresource Locators

A subresource locator returns an object that will handle a 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("id") 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 enterprise beans 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.

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
content-type: text/plain
500 Oracle Parkway, Redwood Shores,
CA
```

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

```
GET /employee
content-type: text/xml
<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`,`encodings`, or `languages` 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) enables you to access XML documents from Java applications.

JAXB provides annotations to map Java classes into XML.

JAX-RS has built-in support for JAXB.

You can add JAXB annotations to the JPA entity manager, and there is no need to use converter classes.

The JAXB annotations within a RESTful web service indicate the XML structure that should be generated from that code.