JAX-RS Injection

Pathparam

- @PathParam allows us to inject the value of named URI path parameters that were defined in @Path expressions.

```
@Path("/customers")
public class CustomerResource {
...
@Path("{id}")
@GET
@Produces("application/xml")
public StreamingOutput getCustomer(@PathParam("id") int id) {
...
}
}
```

- In this example, we want to route HTTP GET requests to the relative URI pattern /customers/{id}. Our getCustomer() method extracts a Customer ID from the URI using @PathParam. The value of the @PathParam annotation, "id", matches the path parameter,{id}, that we defined in the @Path expression of getCustomer().
- While {id} represents a string segment in the request's URI, JAX-RS automatically converts the value to an int before it invokes the getCustomer() method. If the URI path parameter cannot be converted to an integer, the request is considered a client error and the client will receive a 404, "Not Found" response from the server.
- More Than One Path Parameter we can reference more than one URI path parameter in your Java methods. For instance, let's say we are using first and last name to identify a customer in our CustomerResource:

```
@Path("/customers")
public class CustomerResource {
...
@Path("{first}-{last}")
@GET
@Produces("application/xml")
public StreamingOutput getCustomer(@PathParam("first") String firstName,
@PathParam("last") String lastName) {
...
}
}
```

- Here, we have the URI path parameters {first} and {last}. If our HTTP request is GET /customers/bill-burke, bill will be injected into the firstName parameter and burke will be injected into the lastName parameter.
- Sometimes a named URI path parameter will be repeated by different @Path expressions that compose the full URI matching pattern of a resource method. The path parameter could be repeated by the class's @Path expression or by a subresource locator. In these cases, the @PathParam annotation will always reference the final path parameter. For example:

```
@Path("/customers/{id}")
public class CustomerResource {
@Path("/address/{id}")
@Produces("text/plain")
@GET
```

```
public String getAddress(@PathParam("id") String addressId) {...}
}
```

If our HTTP request was GET /customers/123/address/456, the addressId parameter in the getAddress() method would have the 456 value injected.

PathSegment and Matrix Parameters

@PathParam can not only inject the value of a path parameter, it can also inject instances of javax.ws.rs.core.PathSegment. The PathSegment class is an abstraction of a specific URI path segment:

```
package javax.ws.rs.core;
public interface PathSegment {
String getPath();
MultivaluedMap<String, String> getMatrixParameters();
}
```

The getPath() method is the string value of the actual URI segment minus any matrix parameters. The more interesting method here is getMatrixParameters(). This returns a map of the entire matrix parameters applied to a particular URI segment. In combination with @PathParam, you can get access to the matrix parameters applied to your request's URI. For example:

```
@Path("/cars/{make}")
public class CarResource {
@GET
@Path("/{model}/{year}")
@Produces("image/jpeg")
public Jpeg getPicture(@PathParam("make") String make,
@PathParam("model") PathSegment car,
@PathParam("year") String year) {
String carColor = car.getMatrixParameters().getFirst("color");
...
}
```

In this example, we have a CarResource that allows us to get pictures of cars in our database. The getPicture() method returns a JPEG image of cars that match the make, model, and year that we specify. The color of the vehicle is expressed as a matrix parameter of the model. For example:

GET /cars/mercedes/e55;color=black/2006

Here, our make is mercedes, the model is e55 with a color attribute of black, and the year is 2006. While the make, model, and year information can be injected into our getPicture() method directly, we need to do some processing to obtain information about the color of the vehicle.

Instead of injecting the model information as a Java string, we inject the path parameter as a PathSegment into the car parameter. We then use this PathSegment instance to obtain the color matrix parameter's value.

Matching with multiple PathSegments

Sometimes a particular path parameter matches to more than one URI segment. In these cases, you can inject a list of PathSegments. For example, let's say a model in our CarResource could be represented by more than one URI segment. Here's how the getPicture() method might change:

```
@Path("/cars/{make}")
public class CarResource {
@GET
@Path("/{model : .+}/year/{year}")
@Produces("image/jpeg")
public Jpeg getPicture(@PathParam("make") String make,
```

```
@PathParam("model") List<PathSegment> car,
@PathParam("year") String year) {
58 | Chapter 5: JAX-RS Injection
Download at WoweBook.Com
}
}
```

In this example, if our request was GET /cars/mercedes/e55/amg/year/2006, the car parameter would have a list of two PathSegments injected into it, one representing the e55 segment and the other representing the amg segment. We could then query and pull in matrix parameters as needed from these segments.

Programmatic URI Information

All this à la carte injection of path parameter data with the @PathParam annotation is perfect most of the time. Sometimes, though, you need a more general raw API to query and browse information about the incoming request's URI. The interface javax.ws.rs.core.UriInfo provides such an API:

```
public interface UriInfo {
public String getPath();
public String getPath(boolean decode);
public List<PathSegment> getPathSegments();
public List<PathSegment> getPathSegments(boolean decode);
public MultivaluedMap<String, String> getPathParameters();
public MultivaluedMap<String, String> getPathParameters(boolean decode);
...
}
```

The getPath() methods allow you obtain the relative path JAX-RS used to match the incoming request. You can receive the path string decoded or encoded. The getPath Segments() methods break up the entire relative path into a series of PathSegment objects. Like getPath(), you can receive this information encoded or decoded. Finally, getPathParameters() returns a map of all the path parameters defined for all matching @Path expressions.

You can obtain an instance of the UriInfo interface by using the @javax.ws.rs.core.Con text injection annotation. Here's an example:

```
@Path("/cars/{make}")
public class CarResource {
@GET
@Path("/{model}/{year}")
@Produces("image/jpeg")
public Jpeg getPicture(@Context UriInfo info) {
String make = info.getPathParameters().getFirst("make");
PathSegment model = info.getPathSegments().get(1);
String color = model.getMatrixParameters().getFirst("color");
...
}
}
```

In this example, we inject an instance of UriInfo into the getPicture() method's info parameter. We then use this instance to extract information out of the URI.

@MatrixParam

Instead of injecting and processing PathSegment objects to obtain matrix parameter values, the JAX-RS specification allows you to inject matrix parameter values directly through the @javax.ws.rs.MatrixParam annotation. Let's change our CarResource example from the previous section to reflect using this annotation:

```
@Path("/{make}")
public class CarResource {
@GET
@Path("/{model}/{year}")
@Produces("image/jpeg")
public Jpeg getPicture(@PathParam("make") String make,
@PathParam("model") String model,
@MatrixParam("color") String color) {
...
}
```

Using the @MatrixParam annotation shrinks our code and provides a bit more readability. The only downside of @MatrixParam is that sometimes you might have a repeating matrix parameter that is applied to many different path segments in the URI. For example, what if color shows up multiple times in our car service example?

GET /mercedes/e55;color=black/2006/interior;color=tan

Here, the color attribute shows up twice: once with the model and once with the interior. Using @MatrixParam("color") in this case would be ambiguous and we would have to go back to processing PathSegments to obtain this matrix parameter.

@QueryParam

The @javax.ws.rs.QueryParam annotation allows you to inject individual URI query parameters into your Java parameters. For example, let's say we wanted to query a customer database and retrieve a subset of all customers in the database. Our URI might look like this:

GET /customers?start=0&size=10

The start query parameter represents the customer index we want to start with and the size query parameter represents how many customers we want returned. The JAX-RS service that implemented this might look like this:

```
@Path("/customers")
public class CustomerResource {
@GET
@Produces("application/xml")
public String getCustomers(@QueryParam("start") int start,
@QueryParam("size") int size) {
}
}
```

Here, we use the @QueryParam annotation to inject the URI query parameters "start" and "size" into the Java parameters start and size. As with other annotation injection, JAX-RS automatically converts the query parameter's string into an integer.

Programmatic Query Parameter Information

You may have the need to iterate through all query parameters defined on the request URI. The javax.ws.rs.core.UriInfo interface has a getQueryParameters() method that gives you a map containing all query parameters:

```
public interface UriInfo {
...
public MultivaluedMap<String, String> getQueryParameters();
public MultivaluedMap<String, String> getQueryParameters(
boolean decode);
```

```
...
}
```

You can inject instances of UriInfo using the @javax.ws.rs.core.Context annotation. Here's an example of injecting this class and using it to obtain the value of a few query parameters:

```
@Path("/customers")
public class CustomerResource {
@GET
@Produces("application/xml")
public String getCustomers(@Context UriInfo info) {
String start = info.getQueryParameters().getFirst("start");
String size = info.getQueryParameters().getFirst("size");
...
}
}
```

@FormParam

The @javax.ws.rs.FormParam annotation is used to access application/x-www-formurlencoded request bodies. In other words, it's used to access individual entries posted by an HTML form document. For example, let's say we set up a form on our website to register new customers:

```
<FORM action="http://example.com/customers" method="post">
<P>
First name: <INPUT type="text" name="firstname"><BR>
Last name: <INPUT type="text" name="lastname"><BR>
<INPUT type="submit" value="Send">
</P>
</FORM>
...
```

We could post this form directly to a JAX-RS backend service described as follows:

```
@Path("/customers")
public class CustomerResource {
@POST
public void createCustomer(@FormParam("firstname") String first,
@FormParam("lastname") String last) {
...
}
}
```

Here, we are injecting firstname and lastname from the HTML form into the Java parameters first and last. Form data is URL-encoded when it goes across the wire. When using @FormParam, JAX-RS will automatically decode the form entry's value before injecting it.

@HeaderParam

The @javax.ws.rs.HeaderParam annotation is used to inject HTTP request header values. For example, what if your application was interested in the web page that referred to or linked to your web service? You could access the HTTP Referer header using the @HeaderParam annotation:

```
@Path("/myservice")
public class MyService {
@GET
@Produces("text/html")
```

```
public String get(@HeaderParam("Referer") String referer) {
...
}
}
```

The @HeaderParam annotation is pulling the Referer header directly from the HTTP request and injecting it into the referer method parameter.

Raw Headers

Sometimes you need programmatic access to view all headers within the incoming request. For instance, you may want to log them. The JAX-RS specification provides the javax.ws.rs.core.HttpHeaders interface for such scenarios.

```
public interface HttpHeaders {
public List<String> getRequestHeader(String name);
public MultivaluedMap<String, String> getRequestHeaders();
...
}
```

The getRequestHeader() method allows you to get access to one particular header, and getRequestHeaders() gives you a map that represents all headers.

As with UriInfo, you can use the @Context annotation to obtain an instance of HttpHeaders. Here's an example:

```
@Path("/myservice")
public class MyService {
@GET
@Produces("text/html")
public String get(@Context HttpHeaders headers) {
String referer = headers.getRequestHeader("Referer").get(0);
for (String header : headers.getRequestHeaders().keySet())
{
System.out.println("This header was set: " + header);
}
...
}
}
```

@CookieParam

Servers can store state information in cookies on the client, and can retrieve that information when the client makes its next request. Many web applications use cookies to set up a session between the client and the server. They also use cookies to remember identity and user preferences between requests. These cookie values are transmitted back and forth between the client and server via cookie headers.

The @javax.ws.rs.CookieParam annotation allows you to inject cookies sent by a client request into your JAX-RS resource methods. For example, let's say our applications push a customerId cookie to our clients so that we can track users as they invoke and interact with our web services. Code to pull in this information might look like this:

```
@Path("/myservice")
public class MyService {
@GET
@Produces("text/html")
public String get(@CookieParam("customerId") int custId) {
...
}
}
```

The use of @CookieParam here makes the JAX-RS provider search all cookie headers for the customerId cookie value. It then converts it into an int and injects it into the

custId parameter.
If you need more information about the cookie other than its base value, you can instead inject a javax.ws.rs.core.Cookie object:

```
@Path("/myservice")
public class MyService {
@Produces("text/html")
public String get(@CookieParam("customerId") Cookie custId) {
...
}
}
```

The Cookie class has additional contextual information about the cookie beyond its name and value:

```
package javax.ws.rs.core;
public class Cookie
{
public String getName() {...}
public String getValue() {...}
public int getVersion() {...}
public String getDomain() {...}
public String getPath() {...}
...
}
```

The getName() and getValue() methods correspond to the string name and value of the cookie you are injecting. The getVersion() method defines the format of the cookie header, specifically, which version of the cookie specification* the header follows. The getDomain() method specifies the DNS name that the cookie matched. The getPath() method corresponds to the URI path that was used to match the cookie to the incoming request. All these attributes are defined in detail by the IETF cookie specification.
You can also obtain a map of all cookies sent by the client by injecting a reference to javax.ws.rs.core.HttpHeaders:

```
public interface HttpHeaders {
...
public Map<String, Cookie> getCookies();
}
```

As you saw in the previous section, you use the @Context annotation to get access to HttpHeaders. Here's an example of logging all cookies sent by the client:

```
@Path("/myservice")
public class MyService {
@GET
@Produces("text/html")
public String get(@Context HttpHeaders headers) {
for (String name : headers.getCookies().keySet())
{
Cookie cookie = headers.getCookies().get(name);
System.out.println("Cookie: " +
name + "=" + cookie.getValue());
}
...
}
}
```

@DefaultValue

In many types of JAX-RS services, you may have parameters that are optional. When a client does not provide this optional information within the request, JAX-RS will, by default, inject a null value for Object types and a zero value for primitive types. Many times, though, a null or zero value may not work as a default value for your injection. To solve this problem, you can define your own default value for optional parameters by using the @javax.ws.rs.DefaultValue annotation.
For instance, let's look back again at the @QueryParam example given earlier in this chapter. In that example, we wanted to pull down a set of customers from a customer database. We used the start and size query parameters to specify the beginning index and the number of customers desired. While we do want to control the amount of customers sent back as a response, we do not want to require the client to send these query parameters when making a request. We can use @DefaultValue to set a base index and dataset size:

```
import java.util.List;
@Path("/customers")
public class CustomerResource {
@GET
@Produces("application/xml")
public String getCustomers(@DefaultValue("0") @QueryParam("start") int start,
@DefaultValue("10") @QueryParam("size") int size) {
...
}
}
```

Here, we've used @DefaultValue to specify a default start index of 0 and a default dataset size of 10. JAX-RS will use the string conversion rules to convert the string value of the @DefaultValue annotation into the desired Java type.

@Encoded

URI template, matrix, query, and form parameters must all be encoded by the HTTP specification. By default, JAX-RS decodes these values before converting them into the desired Java types. Sometimes, though, you may want to work with the raw encoded values. Using the @javax.ws.rs.Encoded annotation gives you the desired effect:

```
@GET
@Produces("application/xml")
public String get(@Encoded @QueryParam("something") String str) {...}
```

Here, we've used the @Encoded annotation to specify that we want the encoded value of the something query parameter be injected into the str Java parameter. If you want to work solely with encoded values within your Java method or even your entire class, you can annotate the method or class with @Encoded and only encoded values will be used.

Annotation used in JAX-RS

Annotation	Target	Description
Consumes	Type or method	Specifies a list of media types that can be consumed.
Produces	Type or method	Specifies a list of media types that can be produced.
GET	Method	Specifies that the annotated method handles HTTP GET requests.
POST	Method	Specifies that the annotated method handles HTTP POST requests.
PUT	Method	Specifies that the annotated method handles HTTP PUT requests.
DELETE	Method	Specifies that the annotated method handles HTTP DELETE requests.
HEAD	Method	Specifies that the annotated method handles HTTP HEAD requests. Note that HEAD may be automatically handled,
Path	Type or method	Specifies a relative path for a resource. When used on a class this annotation identifies that class as a root resource. When used on a method this annotation identifies a sub-resource method or locator.
PathParam	Parameter, field or method	Specifies that the value of a method parameter, class field,or bean property is to be extracted from the request URI path. The value of the annotation identifies the name of a URI template parameter.
QueryParam	Parameter, field or method	Specifies that the value of a method parameter, class field, or bean property is to be extracted from a URI query parameter. The value of the annotation identifies the name of a query parameter.
FormParam	Parameter, field or method	Specifies that the value of a method parameter is to be extracted from a form

		parameter in a request entity body. The value of the annotation identifies the name of a form parameter. Note that whilst the annotation target allows use on fields and methods, the specification only requires support for use on resource method parameters.
MatrixParam	Parameter, field or method	Specifies that the value of a method parameter, class field, or bean property is to be extracted from a URI matrix parameter. The value of the annotation identifies the name of a matrix parameter.
CookieParam	Parameter, field or method	Specifies that the value of a method parameter, class field, or bean property is to be extracted from a HTTP cookie. The value of the annotation identifies the name of a the cookie.
HeaderParam	Parameter, field or method	Specifies that the value of a method parameter, class field, or bean property is to be extracted from a HTTP header. The value of the annotation identifies the name of a HTTP header.
Encoded	Type, constructor, method, field or parameter	Disables automatic URI decoding for path, query, form and matrix parameters.
DefaultValue	Parameter, field or method	Specifies a default value for a field, property or method parameter annotated with @QueryParam, @MatrixParam, @CookieParam, @FormParam or @HeaderParam. The specified value will be used if the corresponding query or matrix parameter is not present in the request URI, if the corresponding form parameter is not in the request entity body, or if the corresponding HTTP header is not included in the request.

Context	Field, method or parameter	Identifies an injection target for one of the types listed in section 5.2 or the applicable section of chapter 6.
HttpMethod	Annotation	Specifies the HTTP method for a request method designator annotation.
Provider	Type	Specifies that the annotated class implements a JAX-RS extension interface.