JAX-RS

JAX-RS: Java API for RESTful Web Services is a Java programming language API that provides support in creating web services according to the Representational State Transfer (REST) architectural style.

REST

REST (Representational State Transfer) is a style of architecture for building distributed applications using HTTP (Hyper Text Transfer Protocol) protocol as the transport and application protocol. This style of architecture is characterized by following key facts of the architecture style, each of which is briefly explained in the sections that follow.

- Resource
- URIs & Addressability
- Representation
- Uniform Interface
- Stateless

- Connectedness
- Cacheable
- Layered System Design

Resource
In REST architecture style everything that is referenced in the distributed application is a resource, so if I am building a system to expose employee entity for external applications to access then employee is a resource. If I am building a time entry system then assignments against which time is to be entered, time entry and employee entering time all are resources.

URIs & Addressability
URI represent the name and the address of the resource, a resource should have at least one URI. It is important to note that one URI cannot point to two resources. URI gives addressability to the resource and is an important facet of the REST architecture style.

Representation
The data exchanged between the client of the REST service and the service itself is not the resource itself but a representation of the same in a format that both parties understand example XML, JSON, etc.

Uniform Interface
HTTP exposes a few basic methods (see details below) that expose the resources exposed by REST service to its clients. They are as under.

- GET: to get representation of a resource.
- POST: to create a new resource
- PUT: to update an instance of an existing resource
- DELETE: to delete an instance of the resource
- HEAD: returns the headers of the GET request and not the representation of the resource itself, useful for writing business logic on meta-data of the resource.
- OPTIONS: returns a list of above mentioned HTTP methods supported by the
- resource

Stateless
One of the key facets of the REST architecture style is the statelessness of the REST service, i.e. the service does not maintain any state between two client requests. All state that is required by the service to service the request is carried with each request to the service; this implies that in this architecture style client is stateful if required but not the service. Since HTTP is a stateless protocol itself so it does not come as a surprise that this is a key requirement of REST.

Connectedness
The representation of the resource returned by the REST service not only contains the data about the resource only but also links to other related resources, for example a representation of the employee resource may have a link that point to its manager.

Cacheable
HTTP protocol has the concept of caches where in the client or an intermediary is allowed to cache the representations of resource returned by the service, this is a key facet of REST architecture style as well. Hence the REST service must provide instructions to the client/intermediary whether to cache the response or not and if yes then for how long. The response can be cached at the following locations.

1. Client cache (e.g. browser cache in cache of browser based client or custom client cache).
2. Proxy Server cache (this is client side proxy server, caches only public content).
3. Service Side Cache(e.g. cache on service side)

Layered Architecture
When REST clients connect to REST service they may not be connect to the service itself, the response may be returned by an intermediary which cached the response based on the instructions of the service, when it served the request for the first time. This layered architecture with stateless and cache facets increase the scalability of the service a lot.

Characteristics of REST web services

Here are the characteristics of REST:

- **Client-Server:** a pull-based interaction style: consuming components pull representations.
- **Stateless:** each request from client to server must contain all the information necessary to understand the request, and cannot take advantage of any stored context on the server.
- **Cache:** to improve network efficiency responses must be capable of being labeled as cacheable or non-cacheable.
- **Uniform interface:** all resources are accessed with a generic interface (e.g., HTTP GET, POST, PUT, and DELETE).
- **Named resources** - the system is comprised of resources which are named using a URL.
- **Interconnected resource representations** - the representations of the resources are interconnected using URLs, thereby enabling a client to progress from one state to another.
- **Layered components** - intermediaries, such as proxy servers, cache servers, gateways, etc, can be inserted between clients and resources to support performance, security, etc.

Principles of The RESTful Web Service Design

1. The key to creating Web Services in a REST network (i.e., the Web) is to identify all of the conceptual entities that you wish to expose as services. Above we saw some examples of resources: parts list, detailed part data, purchase order.
2. Create a URL to each resource. The resources should be nouns, not verbs. For example, do not use this:
   http://www.parts-depot.com/parts/getPart?id=00345
   Note the verb, getPart. Instead, use a noun:
   http://www.parts-depot.com/parts/00345

3. Categorize your resources according to whether clients can just receive a representation of the resource, or whether clients can modify (add to) the resource. For the former, make those resources accessible using an HTTP GET. For the later, make those resources accessible using HTTP POST, PUT, and/or DELETE.
4. All resources accessible via HTTP GET should be side-effect free. That is, the resource should just return a representation of the resource. Invoking the resource should not result in modifying the resource.
5. No man/woman is an island. Likewise, no representation should be an island. In other words, put hyperlinks within resource representations to enable clients to drill down for more information, and/or to obtain related information.
6. Design to reveal data gradually. Don't reveal everything in a single response document. Provide hyperlinks to obtain more details.
7. Specify the format of response data using a schema (DTD, W3C Schema, RelaxNG, or Schematron). For those services that require a POST or PUT to it, also provide a schema to specify the format of the response.
8. Describe how your services are to be invoked using either a WSDL document, or simply an HTML document.

The REST Architecture

Key components of a REST architecture:

- **Resources**, which are identified by logical URLs. Both *state* and *functionality* are represented using resources.
  - The logical URLs imply that the resources are universally addressable by other parts of the system.
  - **Resources are the key element of a true RESTful design**, as opposed to "methods" or "services" used in RPC and SOAP Web Services, respectively.

- **A web of resources**, meaning that a single resource should not be overwhelmingly large and contain too fine-grained details. Whenever relevant, a resource should contain *links* to additional information -- just as in web pages.
- The system has a **client-server**, but of course one component's server can be another component's client.
- There is **no connection state**; interaction is stateless (although the servers and resources can of course be stateful). Each new request should carry all the information required to complete it, and must not rely on previous interactions with the same client.
- Resources should be **cachable** whenever possible (with an expiration date/time).
- The protocol must allow the server to explicitly specify which resources may be cached, and for how long.

        - Since HTTP is universally used as the REST protocol, the HTTP cache control headers are used for this purpose.
        - Clients must respect the server's cache specification for each resource.
- **Proxy servers** can be used as part of the architecture, to improve performance and scalability. Any standard HTTP proxy can be used.

When to use The RESTful Web Service

Architects and developers need to decide when this particular style is an appropriate choice for their applications. A RESTful design may be appropriate when the web services are completely stateless. A good test is to consider whether the interaction can survive a restart of the server.

- A caching infrastructure can be leveraged for performance. If the data that the web service returns is not dynamically generated and can be cached, then the caching infrastructure that web servers and other intermediaries inherently provide can be leveraged to improve performance. However, the developer must take care because such caches are limited to the HTTP GET method for most servers.

- The service producer and service consumer have a mutual understanding of the context and content being passed along. Because there is no formal way to describe the web services interface, both parties must agree out of band on the schema that describe the data being exchanged and on ways to process it meaningfully. In the real world, most commercial applications that expose services as RESTful implementations also distribute so-called value-added toolkit that describe the interfaces to developers in popular programming languages.

- Bandwidth is particularly important and needs to be limited. REST is particularly useful for limited-profile devices such as PDAs and mobile phones, for which the overhead of headers and additional layers of SOAP elements on the XML payload must be restricted.

- Web service delivery or aggregation into existing web sites can be enabled easily with a RESTful style. Developers can use technologies such as Asynchronous JavaScript with XML (AJAX) and toolkit such as Direct Web Remoting (DWR) to consume the services in their web applications. Rather than starting from scratch, services can be exposed with XML and consumed by HTML pages without significantly refactoring the existing web site architecture. Existing developers will be more productive because they are adding to something they are already familiar with, rather than having to start from scratch with new technology.

Designing a RESTful web service

The underlying RESTful web service design principles can be summarized in the following four steps:

- **Requirements gathering**—this step is similar to traditional software requirement gathering practices.
- **Resource identification**—this step is similar to OOD where we identify objects, but we don't worry about messaging between objects.
- **Resource representation definition**—because we exchange representation between clients and servers, we should define what kind of representation we need to use. Typically, we use XML, but JSON has gained popularity. That's not to say that we can't use any other form of resource representation—on the contrary, we could use XHTML or any other form of binary representation, though we let the requirements guide our choices.
- **URI definition**—with resources in place, we need to define the API, which consists of URIs for clients and servers to exchange resources' representations.

HTTP Method and URI Matching

@Path

- The @javax.ws.rs.Path annotation in JAX-RS is used to define a URI matching pattern for incoming HTTP requests.
- It can be placed upon a class or on one or more Java methods. For a Java class to be eligible to receive any HTTP requests, the class must be annotated with at least the @Path("/") expression. These types of classes are called JAX-RS root resources.
- The value of the @Path annotation is an expression that denotes a relative URI to the context root of your JAX-RS application. For example, if you are deploying into a WAR archive of a servlet container, that WAR will have a base URI that browsers and remote clients use to access it. @Path expressions are relative to this URI.
- To receive a request, a Java method must have at least an HTTP method annotation like @javax.ws.rs.GET applied to it. This method is not required to have an @Path annotation on it, though. For example:

```
@Path("/orders")
public class OrderResource {
@GET
public String getAllOrders() {
...
}
}
```

- An HTTP request of GET /orders would dispatch to the getAllOrders() method. @Path can also be applied to your Java method. If you do this, the URI matching pattern is a concatenation of the class's @Path expression and that of the method's. For example:

```
@Path("/orders")
public class OrderResource {
@GET
@Path("unpaid")
public String getUnpaidOrders() {
...
}
}
```

So, the URI pattern for getUnpaidOrders() would be the relative URI /orders/unpaid.

- @Path expressions are not limited to simple wildcard matching expressions. For example,our getCustomer() method takes an integer parameter. We can change our @Path value to only match digits:

```
@Path("/customers")
public class CustomerResource {
@GET
@Path("{id : \\d+}")
public String getCustomer(@PathParam("id") int id) {
...
}
}
```

- We have taken our {id} expression and changed it to {id : \\d+}. The id string is the identifier of the expression. We'll see in the next chapter how you can reference it with@PathParam. The \\d+ string is a regular expression. It is delimited by a ":" character.

- Regular expressions are part of a string matching language that allows you to express complex string matching patterns. In this case, \\d+ matches one or more digits. We will not cover regular expressions in this book, but you can read up on how to use them directly from your JDK's javadocs within the java.util.regex.Pattern class page.† Regular expressions are not limited in matching one segment of a URI. For example:

```
@Path("/customers")
public class CustomerResource {
@GET
@Path("{id : .+}")
public String getCustomer(@PathParam("id") String id) {
...
}
@GET
@Path("{id : .+}/address")
public String getAddress(@PathParam("id") String id) {
...
}
}
```

- We've changed getCustomer()'s @Path expression to {id : .+}. The .+ is a regular expression that will match any stream of characters after /customers. So, the GET /customers/bill/burke request would be routed to getCustomer().
  The getAddress() method has a more specific expression. It will map any stream of characters after /customers and that ends with /address. So, the GET /customers/bill/burke/address request would be routed to the getAddress() method.

HTTP methods

Every HTTP request made to a web service is handled by an appropriately annotated method in a resource class, provided that the resource class has the annotation @Path defined. JAX-RS defines five annotations that map to specific HTTP operations:

- @javax.ws.rs.GET
- @javax.ws.rs.PUT
- @javax.ws.rs.POST
- @javax.ws.rs.DELETE
- @javax.ws.rs.HEAD

@GET

An HTTP GET request is handled by an annotated method that looks as follows:

```
@Path("/users")
public class UserResource {
@GET
public String handleGETRequest() {
}
}
```

The name of the method is not important. However, we should use descriptive names that are representative of the problem we are solving.

@POST

An HTTP POST request is handled by an annotated method that looks as follows:

```
@Path("/users")
public class UserResource {
@POST
```

```
public String handlePOSTRequest(String payload) {
}
}
```

The POST request has a payload that the framework intercepts and delivers to us in the parameter payload. The value of the payload could be an XML structure, so we use a String object. However, the payload could also be a binary stream of, say, MIME type image/jpg, so our object type for the payload must change accordingly (InputStream, for instance). Note that the name of the payload variable is arbitrary, just like the name of the method containing it.

@PUT

An HTTP PUT request is handled by an annotated method that looks as follows:

```
@Path("/users")
public class UserResource {
@PUT
public String handlePUTRequest(String payload) {
}
}
```

Similar to the POST request, the PUT request has a payload associated with it, which is stored in the payload variable.

@DELETE

An HTTP DELETE request is handled by an annotated method that looks as follows:

```
@Path("/users")
public class UserResource {
@DELETE
public void handleDELETERequest() {
}
}
```

For now, ignore the return type of each of these methods.

Input and output formats

We know how to handle requests made to a web service; we now need to figure out how to exchange inbound and outbound representations between clients and servers. For this, we have the following annotations.

@Consumes

- This annotation works together with @POST and @PUT.
- It tells the framework to which method to delegate the incoming request. Specifically, the client sets the *Content-Type* HTTP header and the framework delegates the request to the corresponding handling method.
- An example of this annotation together with @PUT looks as follows:

```
@Path("/users")
public class UserResource {
@PUT
@Consumes("application/xml")
public void updateUser(String representation) {
}
}
```

- In this example we're telling the framework that the updateUser() method accepts

an input stream of MIME type application/xml, which is stored in the variable representation. Therefore, a client connecting to the web service through the URI /users must send an HTTP PUT method request containing the HTTP header *Content-Type* with a value of application/xml.

- A resource can accept different types of payloads. Thus, we use the @PUT annotation with multiple methods to handle requests with different MIME types. So, we could have the following, a method to accept XML structures and a method to accept JSON structures:

```
@Path("/users")
public class UserResource {
@PUT
@Consumes("application/xml")
public void updateXML(String representation) {
}
@PUT
@Consumes("application/json")
public void updateJSON(String representation) {
}
}
```

- Again, the client must provide the MIME type it's sending to the web service. The job of the framework is to delegate the incoming HTTP request to the method matching the intended MIME type.

For @POST we use the same idea:

```
@Path("/users")
public class UserResource {
@POST
@Consumes("application/xml")
public void updateXML(String representation) {
}
@POST
@Consumes("application/json")
public void updateJSON(String representation) {
}
}
```

@Produces

- This annotation works with @GET, @POST, and @PUT.
- It lets the framework know what type of representation to send back to the client.
- Specifically, the client sends an HTTP request together with an Accept HTTP header that maps directly to the *Content-Type* the method produces. So, if the value of the HTTP header *Accept* is application/xml, the method handling the request returns a stream of MIME type application/xml. This annotation can also be used with multiple methods in the same resource class. An example that returns XML and JSON representations looks as follows:

```
@Path("/users")
public class UserResource {
@GET
@Produces("application/xml")
public String getXML(String representation) {
}
@GET
@Produces("application/json")
public String getJSON(String representation) {
```

```
}
}
```

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.