Red Hat JBoss Fuse 6.0

Developing RESTful Web Services

Standards based RESTful service development

Last Updated: 2017-10-13

Red Hat JBoss Fuse 6.0 Developing RESTful Web Services

Standards based RESTful service development

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Abstract

This guide describes how to use the JAX-RS APIs to implement Web services.

Table of Contents

CHAPTER 1. INTRODUCTION TO RESTFUL WEB SERVICES

Abstract

Representational State Transfer(REST) is a software architecture style that centers around the transmission of data over HTTP, using only the four basic HTTP verbs. It also eschews the use of any additional wrappers such as a SOAP envelope and the use of any state data.

OVERVIEW

Representational State Transfer(REST) is an architectural style first described in a doctoral dissertation by a researcher named Roy Fielding. In RESTful systems, servers expose resources using a URI, and clients access these resources using the four HTTP verbs. As clients receive representations of a resource they are placed in a state. When they access a new resource, typically by following a link, they change, or transition, their state. In order to work, REST assumes that resources are capable of being represented using a pervasive standard grammar.

The World Wide Web is the most ubiquitous example of a system designed on REST principles. Web browsers act as clients accessing resources hosted on Web servers. The resources are represented using HTML or XML grammars that all Web browsers can consume. The browsers can also easily follow the links to new resources.

The advantages of RESTful systems is that they are highly scalable and highly flexible. Because the resources are accessed and manipulated using the four HTTP verbs, the resources are exposed using a URIs, and the resources are represented using standard grammars, clients are not as affected by changes to the servers. Also, RESTful systems can take full advantage of the scalability features of HTTP such as caching and proxies.

BASIC REST PRINCIPLES

RESTful architectures adhere to the following basic principles:

- Application state and functionality are divided into resources.
- Resources are addressable using standard URIs that can be used as hypermedia links.
- All resources use only the four HTTP verbs.
  - **DELETE**
  - **GET**
  - **POST**
  - **PUT**
- All resources provide information using the MIME types supported by HTTP.
- The protocol is stateless.
- Responses are cacheable.
- The protocol is layered.

RESOURCES

Resources are central to REST. A resource is a source of information that can be addressed using a URI. In the early days of the Web, resources were largely static documents. In the modern Web, a resource can be any source of information. For example a Web service can be a resource if it can be accessed using a URI.

RESTful endpoints exchange *representations* of the resources they address. A representation is a document containing the data provided by the resource. For example, the method of a Web service that provides access to a customer record would be a resource, the copy of the customer record exchanged between the service and the consumer is a representation of the resource.

REST BEST PRACTICES

When designing RESTful Web services it is helpful to keep in mind the following:

- Provide a distinct URI for each resource you wish to expose.

  For example, if you are building a system that deals with driving records, each record should have a unique URI. If the system also provides information on parking violations and speeding fines, each type of resource should also have a unique base. For example, speeding fines could be accessed through /speedingfines/*driverID* and parking violations could be accessed through /parkingfines/*driverID*.

- Use nouns in your URIs.

  Using nouns highlights the fact that resources are things and not actions. URIs such as /ordering imply an action, whereas /orders implies a thing.

- Methods that map to **GET** should not change any data.

- Use links in your responses.

  Putting links to other resources in your responses makes it easier for clients to follow a chain of data. For example, if your service returns a collection of resources, it would be easier for a client to access each of the individual resources using the provided links. If links are not included, a client needs to have additional logic to follow the chain to a specific node.

- Make your service stateless.

  Requiring the client or the service to maintain state information forces a tight coupling between the two. Tight couplings make upgrading and migrating more difficult. Maintaining state can also make recovery from communication errors more difficult.

DESIGNING A RESTFUL WEB SERVICE

Regardless of the framework you use to implement a RESTful Web service, there are a number of steps that should be followed:

1. Define the resources the service will expose.

   In general, a service will expose one or more resources that are organized as a tree. For example, a driving record service could be organized into three resources:

   - /license/*driverID*

- /license/*driverID*/speedingfines
- /license/*driverID*/parkingfines

2. Define what actions you want to be able to perform on each resource.

   For example, you may want to be able to update a diver's address or remove a parking ticket from a driver's record.

3. Map the actions to the appropriate HTTP verbs.

Once you have defined the service, you can implement it using Apache CXF.

IMPLEMENTING REST WITH APACHE CXF

Apache CXF provides an implementation of the *Java API for RESTFul Web Services*(JAX-RS). JAX-RS provides a standardized way to map POJOs to resources using annotations.

When moving from the abstract service definition to a RESTful Web service implemented using JAX-RS, you need to do the following:

1. Create a root resource class for the resource that represents the top of the service's resource tree.

   See Section 2.3, “Root resource classes”.

2. Map the service's other resources into sub-resources.

   See Section 2.5, “Working with sub-resources”.

3. Create methods to implement each of the HTTP verbs used by each of the resources.

   See Section 2.4, “Working with resource methods”.

NOTE

Apache CXF continues to support the old HTTP binding to map Java interfaces into RESTful Web services. The HTTP binding provides basic functionality and has a number of limitations. Developers are encouraged to update their applications to use JAX-RS.

DATA BINDINGS

By default, Apache CXF uses Java Architecture for XML Binding(JAXB) objects to map the resources and their representations to Java objects. Provides clean, well defined mappings between Java objects and XML elements.

The Apache CXF JAX-RS implementation also supports exchanging data using *JavaScript Object Notation*(JSON). JSON is a popular data format used by Ajax developers. The marshaling of data between JSON and JAXB is handled by the Apache CXF runtime.

CHAPTER 2. CREATING RESOURCES

Abstract

In RESTful Web services all requests are handled by resources. The JAX-RS APIs implement resources as a Java class. A resource class is a Java class that is annotated with one, or more, RAX-RS annotations. The core of a RESTful Web service implemented using JAX-RS is a root resource class. The root resource class is the entry point to the resource tree exposed by a service. It may handle all requests itself, or it may provide access to sub-resources that handle requests.

2.1. INTRODUCTION

Overview

RESTful Web services implemented using JAX-RS APIs provide responses as representations of a resource implemented by Java classes. A *resource class* is a class that uses JAX-RS annotations to implement a resource. For most RESTful Web services, there is a collection of resources that need to be accessed. The resource class' annotations provide information such as the URI of the resources and which HTTP verb each operation handles.

Types of resources

The JAX-RS APIs allow you to create two basic types of resources:

- A Section 2.3, “Root resource classes” is the entry point to a service's resource tree. It is decorated with the **`@Path`** annotation to define the base URI for the resources in the service.
- Section 2.5, “Working with sub-resources” are accessed through the root resource. They are implemented by methods that are decorated with the **`@Path`** annotation. A sub-resource's **`@Path`** annotation defines a URI relative to the base URI of a root resource.

Example

Example 2.1, “Simple resource class” shows a simple resource class.

Example 2.1. Simple resource class

```
package demo.jaxrs.server;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;

1 @Path("/customerservice")
  public class CustomerService
{
  public CustomerService()
  {
  }

2   @GET
    public Customer getCustomer(@QueryParam("id") String id)
```

```
    {
      ...
    }

    ...
}
```

Two items make the class defined in Example 2.1, “Simple resource class” a resource class:

1. The **@Path** annotation specifies the base URI for the resource.
2. The **@GET** annotation specifies that the method implements the HTTP **GET** method for the resource.

2.2. BASIC JAX-RS ANNOTATIONS

Overview

The most basic pieces of information required by a RESTful Web service implementation are:

- the URI of the service's resources
- how the class' methods are mapped to the HTTP verbs

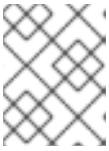
JAX-RS defines a set of annotations that provide this basic information. All resource classes must have at least one of these annotations.

Setting the path

The **@Path** annotation specifies the URI of a resource. The annotation is defined by the **javax.ws.rs.Path** interface and it can be used to decorate either a resource class or a resource method. It takes a string value as its only parameter. The string value is a URI template that specifies the location of an implemented resource.

The URI template specifies a relative location for the resource. As shown in Example 2.2, “URI template syntax”, the template can contain the following:

- unprocessed path components
- parameter identifiers surrounded by **{ }**



NOTE

Parameter identifiers can include regular expressions to alter the default path processing.

Example 2.2. URI template syntax

```
@Path("resourceName/{param1}/../{paramN}")
```

For example, the URI template widgets/{color}/{number} would map to widgets/blue/12. The value of the ***color*** parameter is assigned to blue. The value of the ***number*** parameter is assigned 12.

How the URI template is mapped to a complete URI depends on what the **@Path** annotation is decorating. If it is placed on a root resource class, the URI template is the root URI of all resources in the tree and it is appended directly to the URI at which the service is published. If the annotation decorates a sub-resource, it is relative to the root resource URI.

Specifying HTTP verbs

JAX-RS uses five annotations for specifying the HTTP verb that will be used for a method:

- **javax.ws.rs.DELETE** specifies that the method maps to a **DELETE**.
- **javax.ws.rs.GET** specifies that the method maps to a **GET**.
- **javax.ws.rs.POST** specifies that the method maps to a **POST**.
- **javax.ws.rs.PUT** specifies that the method maps to a **PUT**.
- **javax.ws.rs.HEAD** specifies that the method maps to a **HEAD**.

When you map your methods to HTTP verbs, you must ensure that the mapping makes sense. For example, if you map a method that is intended to submit a purchase order, you would map it to a **PUT** or a **POST**. Mapping it to a **GET** or a **DELETE** would result in unpredictable behavior.

2.3. ROOT RESOURCE CLASSES

Overview

A root resource class is the entry point into a JAX-RS implemented RESTful Web service. It is decorated with a **@Path** that specifies the root URI of the resources implemented by the service. Its methods either directly implement operations on the resource or provide access to sub-resources.

Requirements

In order for a class to be a root resource class it must meet the following criteria:

- The class must be decorated with the **@Path** annotation.

  The specified path is the root URI for all of the resources implemented by the service. If the root resource class specifies that its path is widgets and one of its methods implements the **GET** verb, then a **GET** on widgets invokes that method. If a sub-resource specifies that its URI is {id}, then the full URI template for the sub-resource is widgets/{id} and it will handle requests made to URIs like widgets/12 and widgets/42.

- The class must have a public constructor for the runtime to invoke.

  The runtime must be able to provide values for all of the constructor's parameters. The constructor's parameters can include parameters decorated with the JAX-RS parameter annotations. For more information on the parameter annotations see Chapter 3, *Passing Information into Resource Classes and Methods*.

- At least one of the classes methods must either be decorated with an HTTP verb annotation or the **@Path** annotation.

Example

Example 2.3, "Root resource class" shows a root resource class that provides access to a sub-resource.

Example 2.3. Root resource class

```
package demo.jaxrs.server;

import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Response;

1 @Path("/customerservice/")
  public class CustomerService
{
2   public CustomerService()
    {
    ...
  }

3   @GET
    public Customer getCustomer(@QueryParam("id") String id)
  {
    ...
  }

  @DELETE
  public Response deleteCustomer(@QueryParam("id") String id)
  {
    ...
  }

  @PUT
  public Response updateCustomer(Customer customer)
  {
    ...
  }

  @POST
  public Response addCustomer(Customer customer)
  {
    ...
  }

4   @Path("/orders/{orderId}/")
    public Order getOrder(@PathParam("orderId") String orderId)
  {
```

```
        ...
    }

}
```

The class in Example 2.3, "Root resource class" meets all of the requirements for a root resource class.

1. The class is decorated with the **@Path** annotation. The root URI for the resources exposed by the service is customerservice.
2. The class has a public constructor. In this case the no argument constructor is used for simplicity.
3. The class implements each of the four HTTP verbs for the resource.
4. The class also provides access to a sub-resource through the **getOrder()** method. The URI for the sub-resource, as specified using the the **@Path** annotation, is customerservice/order/*id*. The sub-resource is implemented by the **Order** class.

   For more information on implementing sub-resources see Section 2.5, "Working with sub-resources".

2.4. WORKING WITH RESOURCE METHODS

Overview

Resource methods are annotated using JAX-RS annotations. They have one of the HTTP method annotation specifying the types of requests that the method processes. JAX-RS places several constraints on resource methods.

General constraints

All resource methods must meet the following conditions:

- It must be public.
- It must be decorated with one of the HTTP method annotations described in the section called "Specifying HTTP verbs".
- It must not have more than one entity parameter as described in the section called "Parameters".

Parameters

Resource method parameters take two forms:

- **entity parameters**—Entity parameters are not annotated. Their value is mapped from the request entity body. An entity parameter can be of any type for which your application has an entity provider. Typically they are JAXB objects.

IMPORTANT

A resource method can have *only one* entity parameter.

For more information on entity providers see Chapter 7, *Entity Support*.

- **annotated parameters**—Annotated parameters use one of the JAX-RS annotations that specify how the value of the parameter is mapped from the request. Typically, the value of the parameter is mapped from portions of the request URI.

  For more information about using the JAX-RS annotations for mapping request data to method parameters see Chapter 3, *Passing Information into Resource Classes and Methods*

Example 2.4, "Resource method with a valid parameter list" shows a resource method with a valid parameter list.

Example 2.4. Resource method with a valid parameter list

```
@POST
@Path("disaster/monster/giant/{id}")
public void addDaikaiju(Kaiju kaiju,
                        @PathParam("id") String id)
{
  ...
}
```

Example 2.5, "Resource method with an invalid parameter list" shows a resource method with an invalid parameter list. It has two parameters that are not annotated.

Example 2.5. Resource method with an invalid parameter list

```
@POST
@Path("disaster/monster/giant/")
public void addDaikaiju(Kaiju kaiju,
                        String id)
{
  ...
}
```

Return values

Resource methods can return one of the following:

- **`void`**
- any Java class for which the application has an entity provider

  For more information on entity providers see Chapter 7, *Entity Support*.

- a **Response** object

  For more information on **Response** objects see Section 4.2, "Fine tuning an application's responses".

- a **`GenericEntity<T>`** object

For more information on **GenericEntity<*T*>** objects see Section 4.3, "Returning entities with generic type information".

All resource methods return an HTTP status code to the requester. When the return type of the method is **void** or the value being returned is **null**, the resource method sets the HTTP status code to **200**. When the resource method returns any value other than **null**, it sets the HTTP status code to **204**.

2.5. WORKING WITH SUB-RESOURCES

Overview

It is likely that a service will need to be handled by more than one resource. For example, in an order processing service best-practices suggests that each customer would be handled as a unique resource. Each order would also be handled as a unique resource.

Using the JAX-RS APIs, you would implement the customer resources and the order resources as *sub-resources*. A sub-resource is a resource that is accessed through a root resource class. They are defined by adding a **@Path** annotation to a resource class' method. Sub-resources can be implemented in one of two ways:

- *Sub-resource method*—directly implements an HTTP verb for a sub-resource and is decorated with one of the annotations described in the section called "Specifying HTTP verbs".
- *Sub-resource locator*—points to a class that implements the sub-resource.

Specifying a sub-resource

Sub-resources are specified by decorating a method with the **@Path** annotation. The URI of the sub-resource is constructed as follows:

1. Append the value of the sub-resource's **@Path** annotation to the value of the sub-resource's parent resource's **@Path** annotation.

   The parent resource's **@Path** annotation maybe located on a method in a resource class that returns an object of the class containing the sub-resource.

2. Repeat the previous step until the root resource is reached.

3. The assembled URI is appended to the base URI at which the service is deployed.

For example the URI of the sub-resource shown in Example 2.6, "Order sub-resource" could be *baseURI*/customerservice/order/12.

Example 2.6. Order sub-resource

```
...
@Path("/customerservice/")
public class CustomerService
{
  ...
  @Path("/orders/{orderId}/")
  @GET
  public Order getOrder(@PathParam("orderId") String orderId)
```

```
  {
    ...
  }
}
```

Sub-resource methods

A sub-resource method is decorated with both a **@Path** annotation and one of the HTTP verb annotations. The sub-resource method is directly responsible for handling a request made on the resource using the specified HTTP verb.

Example 2.7, "Sub-resource methods" shows a resource class with three sub-resource methods:

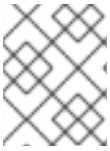
- **getOrder()** handles HTTP **GET** requests for resources whose URI matches /customerservice/orders/{orderId}/.
- **updateOrder()** handles HTTP **PUT** requests for resources whose URI matches /customerservice/orders/{orderId}/.
- **newOrder()** handles HTTP **POST** requests for the resource at /customerservice/orders/.

Example 2.7. Sub-resource methods

```
...
@Path("/customerservice/")
public class CustomerService
{
  ...
  @Path("/orders/{orderId}/")
  @GET
  public Order getOrder(@PathParam("orderId") String orderId)
  {
    ...
  }

  @Path("/orders/{orderId}/")
  @PUT
  public Order updateOrder(@PathParam("orderId") String orderId,
                           Order order)
  {
    ...
  }

  @Path("/orders/")
  @POST
  public Order newOrder(Order order)
  {
    ...
  }
}
```



NOTE

Sub-resource methods with the same URI template are equivalent to resource class returned by a sub-resource locator.

Sub-resource locators

Sub-resource locators are not decorated with one of the HTTP verb annotations and do not directly handle are request on the sub-resource. Instead, a sub-resource locator returns an instance of a resource class that can handle the request.

In addition to not having an HTTP verb annotation, sub-resource locators also cannot have any entity parameters. All of the parameters used by a sub-resource locator method must use one of the annotations described in Chapter 3, *Passing Information into Resource Classes and Methods*

As shown in Example 2.8, "Sub-resource locator returning a specific class" , sub-resource locator allows you to encapsulate a resource as a reusable class instead of putting all of the methods into one super class. The **`processOrder()`** method is a sub-resource locator. When a request is made on a URI matching the URI template /orders/{orderId}/ it returns an instance of the **`Order`** class. The **`Order`** class has methods that are decorated with HTTP verb annotations. A **PUT** request is handled by the **`updateOrder()`** method.

Example 2.8. Sub-resource locator returning a specific class

```
...
@Path("/customerservice/")
public class CustomerService
{
  ...
  @Path("/orders/{orderId}/")
  public Order processOrder(@PathParam("orderId") String orderId)
  {
    ...
  }

  ...
}

public class Order
{
  ...
  @GET
  public Order getOrder(@PathParam("orderId") String orderId)
  {
    ...
  }

  @PUT
  public Order updateOrder(@PathParam("orderId") String orderId,
                           Order order)
  {
    ...
  }

}
```

Sub-resource locators are processed at runtime so that they can support polymorphism. The return value of a sub-resource locator can be a generic `Object`, an abstract class, or the top of a class hierarchy. For example, if your service needed to process both PayPal orders and credit card orders, the `processOrder()` method's signature from Example 2.8, "Sub-resource locator returning a specific class" could remain unchanged. You would simply need to implement two classes, `ppOrder` and `ccOder`, that extended the `Order` class. The implementation of `processOrder()` would instantiate the desired implementation of the sub-resource based on what ever logic is required.

2.6. RESOURCE SELECTION METHOD

Overview

It is possible for a given URI to map to one or more resource methods. For example the URI customerservice/12/ma could match the templates `@Path("customerservice/{id}")` or `@Path("customerservice/{id}/{state}")`. JAX-RS specifies a detailed algorithm for matching a resource method to a request. The algorithm compares the normalized URI, the HTTP verb, and the media types of the request and response entities to the annotations on the resource classes.

The basic selection algorithm

The JAX-RS selection algorithm is broken down into three stages:

1. Determine the root resource class.

   The request URI is matched against all of the classes decorated with the `@Path` annotation. The classes whose `@Path` annotation matches the request URI are determined.

   If the value of the resource class' `@Path` annotation matches the entire request URI, the class' methods are used as input into the third stage.

2. Determine the object will handle the request.

   If the request URI is longer than the value of the selected class' `@Path` annotation, the values of the resource methods' `@Path` annotations are used to look for a sub-resource that can process the request.

   If one or more sub-resource methods match the request URI, these methods are used as input for the third stage.

   If the only matches for the request URI are sub-resource locaters, the resource methods of the object created by the sub-resource locater to match the request URI. This stage is repeated until a sub-resource method matches the request URI.

3. Select the resource method that will handle the request.

   The resource method whose HTTP verb annotation matches the HTTP verb in the request. In addition, the selected resource method must accept the media type of the request entity body and be capable of producing a response that conforms to the media type(s) specified in the request.

Selecting from multiple resource classes

The first two stages of the selection algorithm determine the resource that will handle the request. In

some cases the resource is implemented by a resource class. In other cases, it is implemented by one or more sub-resources that use the same URI template. When there are multiple resources that match a request URI, resource classes are preferred over sub-resources.

If more than one resource still matches the request URI after sorting between resource classes and sub-resources, the following criteria are used to select a single resource:

1. Prefer the resource with the most literal characters in its URI template.

   Literal characters are characters that are not part of a template variable. For example, /widgets/{id}/{color} has ten literal characters and /widgets/1/{color} has eleven literal characters. So, the request URI /widgets/1/red would be matched to the resource with /widgets/1/{color} as its URI template.

NOTE

A trailing slash (/) counts as a literal character. So /joefred/ will be preferred over /joefred.

2. Prefer the resource with the most variables in its URI template.

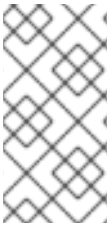
   The request URI /widgets/30/green could match both /widgets/{id}/{color} and /widgets/{amount}/. However, the resource with the URI template /widgets/{id}/{color} will be selected because it has two variables.

3. Prefer the resource with the most variables containing regular expressions.

   The request URI /widgets/30/green could match both /widgets/{number}/{color} and /widgets/{id:.+}/{color}. However, the resource with the URI template /widgets/{id:.+}/{color} will be selected because it has a variable containing a regular expression.

Selecting from multiple resource methods

In many cases, selecting a resource that matches the request URI results in a single resource method that can process the request. The method is determined by matching the HTTP verb specified in the request with a resource method's HTTP verb annotation. In addition to having the appropriate HTTP verb annotation, the selected method must also be able to handle the request entity included in the request and be able to produce the proper type of response specified in the request's metadata.



NOTE

The type of request entity a resource method can handle is specified by the **@Consumes** annotation. The type of responses a resource method can produce are specified using the **@Produces** annotation. For more information see Chapter 8, *Customizing the Media Types Handled by a Resource*.

When selecting a resource produces multiple methods that can handle a request the following criteria is used to select the resource method that will handle the request:

1. Prefer resource methods over sub-resources.

2. Prefer sub-resource methods over sub-resource locaters.

3. Prefer methods that use the most specific values in the **@Consumes** annotation and the **@Produces** annotation.

For example, a method that has the annotation **@Consumes(text/xml)** would be preferred over a method that has the annotation **@Consumes(text/*)**. Both methods would be preferred over a method without an **@Consumes** annotation or the annotation **@Consumes(*/*)**.

4. Prefer methods that most closely match the content type of the request body entity.

   TIP

   The content type of the request body entity is specified in the HTTP Content-Type property.

5. Prefer methods that most closely match the content type accepted as a response.

   TIP

   The content types accepted as a response are specified in the HTTP Accept property.

Customizing the selection process

In some cases, developers have reported the algorithm being somewhat restrictive in the way multiple resource classes are selected. For example, if a given resource class has been matched and if this class has no matching resource method, then the algorithm stops executing. It never checks the remaining matching resource classes.

Apache CXF provides the **org.apache.cxf.jaxrs.ext.ResourceComparator** interface which can be used to customize how the runtime handles multiple matching resource classes. The **ResourceComparator** interface, shown in Example 2.9, "Interface for customizing resource selection", has to methods that need to be implemented. One compares two resource classes and the other compares two resource methods.

Example 2.9. Interface for customizing resource selection

```
package org.apache.cxf.jaxrs.ext;

import org.apache.cxf.jaxrs.model.ClassResourceInfo;
import org.apache.cxf.jaxrs.model.OperationResourceInfo;
import org.apache.cxf.message.Message;

public interface ResourceComparator
{
  int compare(ClassResourceInfo cri1,
              ClassResourceInfo cri2,
              Message message);

  int compare(OperationResourceInfo oper1,
              OperationResourceInfo oper2,
              Message message);
}
```

Custom implementations select between the two resources as follows:

- Return **1** if the first parameter is a better match than the second parameter

- Return `-1` if the second parameter is a better match than the first parameter

If `0` is returned then the runtime will proceed with the default selection algorithm

You register a custom `ResourceComparator` implementation by adding a `resourceComparator` child to the service's `jaxrs:server` element.

CHAPTER 3. PASSING INFORMATION INTO RESOURCE CLASSES AND METHODS

Abstract

JAX-RS specifies a number of annotations that allow the developer to control where the information passed into resources come from. The annotations conform to common HTTP concepts such as matrix parameters in a URI. The standard APIs allow the annotations to be used on method parameters, bean properties, and resource class fields. Apache CXF provides an extension that allows for the injection of a sequence of parameters to be injected into a bean.

3.1. BASICS OF INJECTING DATA

Overview

Parameters, fields, and bean properties that are initialized using data from the HTTP request message have their values injected into them by the runtime. The specific data that is injected is specified by a set of annotations described in Section 3.2, “Using JAX-RS APIs”.

The JAX-RS specification places a few restrictions on when the data is injected. It also places a few restrictions on the types of objects into which request data can be injected.

When data is injected

Request data is injected into objects when they are instantiated due to a request. This means that only objects that directly correspond to a resource can use the injection annotations. As discussed in Chapter 2, *Creating Resources*, these objects will either be a root resource decorated with the `@Path` annotation or an object returned from a sub-resource locator method.

Supported data types

The specific set of data types that data can be injected into depends on the annotation used to specify the source of the injected data. However, all of the injection annotations support at least the following set of data types:

- primitives such as int, char, or long
- Objects that have a constructor that accepts a single String argument
- Objects that have a static `valueOf()` method that accepts a single String argument
- `List<T>`, `Set<T>`, or `SortedSet<T>` objects where *T* satisfies the other conditions in the list

TIP

Where injection annotations have different requirements for supported data types, the differences will be highlighted in the discussion of the annotation.

3.2. USING JAX-RS APIS

The standard JAX-RS API specifies annotations that can be used to inject values into fields, bean properties, and method parameters. The annotations can be split up into three distinct types:

- annotations that inject information from the request URI
- annotations that inject information from the HTTP message header
- annotations that inject information from HTML forms

3.2.1. Injecting data from a request URI

Overview

One of the best practices for designing a RESTful Web service is that each resource should have a unique URI. A developer can use this principle to provide a good deal of information to the underlying resource implementation. When designing URI templates for a resource, a developer can build the templates to include parameter information that can be injected into the resource implementation. Developers can also leverage query and matrix parameters for feeding information into the resource implementations.

Getting data from the URI's path

One of the more common mechanisms for getting information about a resource is through the variables used in creating the URI templates for a resource. This is accomplished using the **`javax.ws.rs.PathParam`** annotation. The **`@PathParam`** annotation has a single parameter that identifies the URI template variable from which the data will be injected.

In Example 3.1, "Injecting data from a URI template variable" the **`@PathParam`** annotation specifies that the value of the URI template variable **`color`** is injected into the **`itemColor`** field.

Example 3.1. Injecting data from a URI template variable

```
import javax.ws.rs.Path;
import javax.ws.rs.PathParam
...

@Path("/boxes/{shape}/{color}")
class Box
{
  ...

  @PathParam("color")
  String itemColor;

  ...
}
```

The data types supported by the **`@PathParam`** annotation are different from the ones described in the section called "Supported data types". The entity into which the **`@PathParam`** annotation injects data must be of one of the following types:

- **`PathSegment`**

The value will be the final segment of the matching part of the path.

- **`List<PathSegment>`**

  The value will be a list of **`PathSegment`** objects corresponding to the path segment(s) that matched the named template parameter.

- primitives such as int, char, or long
- Objects that have a constructor that accepts a single String argument
- Objects that have a static **`valueOf()`** method that accepts a single String argument

Using query parameters

A common way of passing information on the Web is to use *query parameters* in a URI. Query parameters appear at the end of the URI and are separated from the resource location portion of the URI by a question mark(**?**). They consist of one, or more, name value pairs where the name and value are separated by an equal sign(**=**). When more than one query parameter is specified, the pairs are separated from each other by either a semicolon(**;**) or an ampersand(**&**). Example 3.2, "URI with a query string" shows the syntax of a URI with query parameters.

Example 3.2. URI with a query string

```
http://fusesource.org?name=value;name2=value2;...
```

NOTE

You can use **either** the semicolon or the ampersand to separate query parameters, but not both.

The **`javax.ws.rs.QueryParam`** annotation extracts the value of a query parameter and injects it into a JAX-RS resource. The annotation takes a single parameter that identifies the name of the query parameter from which the value is extracted and injected into the specified field, bean property, or parameter. The **`@QueryParam`** annotation supports the types described in the section called "Supported data types".

Example 3.3, "Resource method using data from a query parameter" shows a resource method that injects the value of the query parameter ***`id`*** into the method's ***`id`*** parameter.

Example 3.3. Resource method using data from a query parameter

```
import javax.ws.rs.QueryParam;
import javax.ws.rs.PathParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...

@Path("/monstersforhire/")
public class MonsterService
{
  ...
  @POST
```

```
  @Path("\{type}")
  public void updateMonster(@PathParam("type") String type,
                            @QueryParam("id") String id)
  {
    ...
  }
  ...
}
```

To process an HTTP **POST** to /monstersforhire/daikaiju?id=jonas the **`updateMonster()`** method's ***`type`*** is set to **`daikaiju`** and the ***`id`*** is set to **`jonas`**.

Using matrix parameters

URI matrix parameters, like URI query parameters, are name/value pairs that can provide additional information selecting a resource. Unlike query parameters, matrix parameters can appear anywhere in a URI and they are separated from the hierarchical path segments of the URI using a semicolon(**`;`**). /mostersforhire/daikaiju;id=jonas has one matrix parameter called ***`id`*** and /monstersforhire/japan;type=daikaiju/flying;wingspan=40 has two matrix parameters called ***`type`*** and ***`wingspan`***.

NOTE

Matrix parameters are not evaluated when computing a resource's URI. So, the URI used to locate the proper resource to handle the request URI /monstersforhire/japan;type=daikaiju/flying;wingspan=40 is /monstersforhire/japan/flying.

The value of a matrix parameter is injected into a field, parameter, or bean property using the **`javax.ws.rs.MatrixParam`** annotation. The annotation takes a single parameter that identifies the name of the matrix parameter from which the value is extracted and injected into the specified field, bean property, or parameter. The **`@MatrixParam`** annotation supports the types described in the section called “Supported data types”.

Example 3.4, “Resource method using data from matrix parameters” shows a resource method that injects the value of the matrix parameters ***`type`*** and ***`id`*** into the method's parameters.

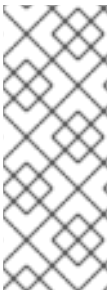
Example 3.4. Resource method using data from matrix parameters

```
import javax.ws.rs.MatrixParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...

@Path("/monstersforhire/")
public class MonsterService
{
  ...
  @POST
  public void updateMonster(@MatrixParam("type") String type,
                            @MatrixParam("id") String id)
  {
    ...
```

```
    }
    ...
}
```

To process an HTTP **POST** to /monstersforhire;type=daikaiju;id=whale the **updateMonster()** method's ***type*** is set to **daikaiju** and the ***id*** is set to **whale**.



NOTE

JAX-RS evaluates all of the matrix parameters in a URI at once, so it cannot enforce constraints on a matrix parameters location in a URI. For example /monstersforhire/japan;type=daikaiju/flying;wingspan=40 , /monstersforhire/japan/flying;type=daikaiju;wingspan=40, and /monstersforhire/japan;type=daikaiju;wingspan=40/flying are all treated as equivalent by a RESTful Web service implemented using the JAX-RS APIs.

Disabling URI decoding

By default all request URIs are decoded. So the URI /monster/night%20stalker and the URI /monster/night stalker are equivalent. The automatic URI decoding makes it easy to send characters outside of the ASCII character set as parameters.

If you do not wish to have URI automatically decoded, you can use the **javax.ws.rs.Encoded** annotation to deactivate the URI decoding. The annotation can be used to deactivate URI decoding at the following levels:

- class level—Decorating a class with the **@Encoded** annotation deactivates the URI decoding for all parameters, field, and bean properties in the class.
- method level—Decorating a method with the **@Encoded** annotation deactivates the URI decoding for all parameters of the class.
- parameter/field level—Decorating a parameter or field with the **@Encoded** annotation deactivates the URI decoding for all parameters of the class.

Example 3.5, “Disabling URI decoding” shows a resource whose **getMonster()** method does not use URI decoding. The **addMonster()** method only disables URI decoding for the ***type*** parameter.

Example 3.5. Disabling URI decoding

```
@Path("/monstersforhire/")
public class MonsterService
{
  ...

  @GET
  @Encoded
  @Path("\{type}")
  public Monster getMonster(@PathParam("type") String type,
                            @QueryParam("id") String id)
  {
    ...
  }
```

```
  @PUT
  @Path("\{id}")
  public void addMonster(@Encoded @PathParam("type") String type,
                         @QueryParam("id") String id)
  {
    ...
  }
  ...
}
```

Error handling

If an error occurs when attempting to inject data using one of the URI injection annotations a **`WebApplicationException`** exception wraps the original exception is generated. The **`WebApplicationException`** exception's status is set to **`404`**.

3.2.2. Injecting data from the HTTP message header

Overview

In normal usage the HTTP headers in a request message pass along generic information about the message, how it is to be handled in transit, and details about the expected response. While a few standard headers are commonly recognized and used, the HTTP specification allows for any name/value pair to be used as an HTTP header. The JAX-RS APIs provide an easy mechanism for injecting HTTP header information into a resource implementation.

One of the most commonly used HTTP headers is the cookie. Cookies allow HTTP clients and servers to share static information across multiple request/response sequences. The JAX-RS APIs provide an annotation inject data directly from a cookie into a resource implementation.

Injecting information from the HTTP headers

The **`javax.ws.rs.HeaderParam`** annotation is used to inject the data from an HTTP header field into a parameter, field, or bean property. It has a single parameter that specifies the name of the HTTP header field from which the value is extracted and injected into the resource implementation. The associated parameter, field, or bean property must conform to the data types described in the section called “Supported data types”.

Example 3.6, “Injecting the If-Modified-Since header” shows code for injecting the value of the HTTP **`If-Modified-Since`** header into a class' **`oldestDate`** field.

Example 3.6. Injecting the If-Modified-Since header

```
import javax.ws.rs.HeaderParam;
...
class RecordKeeper
{
  ...
  @HeaderParam("If-Modified-Since")
  String oldestDate;
  ...
}
```

Injecting information from a cookie

Cookies are a special type of HTTP header. They are made up of one or more name/value pairs that are passed to the resource implementation on the first request. After the first request, the cookie is passes back and forth between the provider and consumer with each message. Only the consumer, because they generate requests, can change the cookie. Cookies are commonly used to maintain session across multiple request/response sequences, storing user settings, and other data that can persist.

The **javax.ws.rs.CookieParam** annotation extracts the value from a cookie's field and injects it into a resource implementation. It takes a single parameter that specifies the name of the cookie's field from which the value is to be extracted. In addition to the data types listed in the section called "Supported data types", entities decorated with the **@CookieParam** can also be a **Cookie** object.

Example 3.7, "Injecting a cookie" shows code for injecting the value of the **handle** cookie into a field in the **CB** class.

Example 3.7. Injecting a cookie

```
import javax.ws.rs.CookieParam;
...
class CB
{
  ...
  @CookieParam("handle")
  String handle;
  ...
}
```

Error handling

If an error occurs when attempting to inject data using one of the HTTP message injection annotations a **WebApplicationException** exception wrapping the original exception is generated. The **WebApplicationException** exception's status is set to **400**.

3.2.3. Injecting data from HTML forms

Overview

HTML forms are an easy means of getting information from a user and they are also easy to create. Form data can be used for HTTP **GET** requests and HTTP **POST** requests:

GET

When form data is sent as part of an HTTP **GET** request the data is appended to the URI as a set of query parameters. Injecting data from query parameters is discussed in the section called "Using query parameters".

POST

When form data is sent as part of an HTTP **POST** request the data is placed in the HTTP message

body. The form data can be handled using a regular entity parameter that supports the form data. It can also be handled by using the **@FormParam** annotation to extract the data and inject the pieces into resource method parameters.

Using the @FormParam annotation to inject form data

The **javax.ws.rs.FormParam** annotation extracts field values from form data and injects the value into resource method parameters. The annotation takes a single parameter that specifies the key of the field from which it extracts the values. The associated parameter must conform to the data types described in the section called "Supported data types".

IMPORTANT

The JAX-RS API Javadoc states that the **@FormParam** annotation can be placed on fields, methods, and parameters. However, the **@FormParam** annotation is only meaningful when placed on resource method parameters.

Example

Example 3.8, "Injecting form data into resource method parameters" shows a resource method that injects form data into its parameters. The method assumes that the client's form includes three fields —**title**, **tags**, and **body**—that contain string data.

Example 3.8. Injecting form data into resource method parameters

```
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;

...
@POST
public boolean updatePost(@FormParam("title") String title,
                          @FormParam("tags") String tags,
                          @FormParam("body") String post)
{
  ...
}
```

3.2.4. Specifying a default value to inject

Overview

To provide for a more robust service implementation, you may want to ensure that any optional parameters can be set to a default value. This can be particularly useful for values that are taken from query parameters and matrix parameters since entering long URI strings is highly error prone. You may also want to set a default value for a parameter extracted from a cookie since it is possible for a requesting system not have the proper information to construct a cookie with all the values.

The **javax.ws.rs.DefaultValue** annotation can be used in conjunction with the following injection annotations:

- **@PathParam**

- **@QueryParam**
- **@MatrixParam**
- **@FormParam**
- **@HeaderParam**
- **@CookieParam**

The **@DefaultValue** annotation specifies a default value to be used when the data corresponding to the injection annotation is not present in the request.

Syntax

Example 3.9, "Syntax for setting the default value of a parameter" shows the syntax for using the **@DefaultValue** annotation.

Example 3.9. Syntax for setting the default value of a parameter

```
import javax.ws.rs.DefaultValue;
  ...
  void resourceMethod(@MatrixParam("matrix")
                      @DefaultValue("value)
                      int someValue, ... )
  ...
```

The annotation must come before the parameter, bean, or field, it will effect. The position of the **@DefaultValue** annotation relative to the accompanying injection annotation does not matter.

The **@DefaultValue** annotation takes a single parameter. This parameter is the value that will be injected into the field if the proper data cannot be extracted based on the injection annotation. The value can be any String value. The value should be compatible with type of the associated field. For example, if the associated field is of type int, a default value of **blue** results in an exception.

Dealing with lists and sets

If the type of the annotated parameter, bean or field is **List**, **Set**, or **SortedSet** then the resulting collection will have a single entry mapped from the supplied default value.

Example

Example 3.10, "Setting default values" shows two examples of using the **@DefaultValue** to specify a default value for a field whose value is injected.

Example 3.10. Setting default values

```
import javax.ws.rs.DefaultValue;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
```

```
@Path("/monster")
public class MonsterService
{

  @Get
  public Monster getMonster(@QueryParam("id") @DefaultValue("42") int
id,
                            @QueryParam("type")
@DefaultValue("bogeyman") String type)
  {
    ...
  }

  ...
}
```

The **getMonster()** method in Example 3.10, "Setting default values" is invoked when a **GET** request is sent to *baseURI*/monster. The method expects two query parameters, ***id*** and ***type***, appended to the URI. So a **GET** request using the URI *baseURI*/monster?id=1&type=fomóiri would return the Fomóiri with the id of one.

Because the **@DefaultValue** annotation is placed on both parameters, the **getMonster()** method can function if the query parameters are omitted. A **GET** request sent to *baseURI*/monster is equivalent to a **GET** request using the URI *baseURI*/monster?id=42&type=bogeyman.

3.3. USING APACHE CXF EXTENSIONS

Overview

Apache CXF provides an extension to the standard JAX-WS injection mechanism that allows developers to replace a sequence of injection annotations with a single annotation. The single annotation is place on a bean containing fields for the data that is extracted using the annotation. For example, if a resource method is expecting a request URI to include three query parameters called ***id***, ***type***, and ***size***, it could use a single **@QueryParam** annotation to inject all of the parameters into a bean with corresponding fields.

Supported injection annotations

This extension does not support all of the injection parameters. It only supports the following ones:

- **@PathParam**
- **@QueryParam**
- **@MatrixParam**
- **@FormParam**

Syntax

To indicate that an annotation is going to use serial injection into a bean, you need to do two things:

1. Specify the annotation's parameter as an empty string. For example **`@PathParam("")`** specifies that a sequence of URI template variables are to be serialized into a bean.

2. Ensure that the annotated parameter is a bean with fields that match the values being injected.

Example

Example 3.11, "Injecting query parameters into a bean" shows an example of injecting a number of Query parameters into a bean. The resource method expect the request URI to include two query parameters: ***`type`*** and ***`id`***. Their values are injected into the corresponding fields of the **`Monster`** bean.

Example 3.11. Injecting query parameters into a bean

```
import javax.ws.rs.QueryParam;
import javax.ws.rs.PathParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...

@Path("/monstersforhire/")
public class MonsterService
{
  ...
  @POST
  public void updateMonster(@QueryParam("") Monster bean)
  {
    ...
  }
  ...
}

public class Monster
{
  String type;
  String id;

  ...
}
```

CHAPTER 4. RETURNING INFORMATION TO THE CONSUMER

Abstract

RESTful requests require that at least an HTTP response code be returned to the consumer. In many cases, a request can be satisfied by returning a plain JAXB object or a **`GenericEntity`** object. When the resource method needs to return additional metadata along with the response entity, JAX-RS resource methods can return a **`Response`** object containing any needed HTTP headers or other metadata.

The information returned to the consumer determines the exact type of object a resource method returns. This may seem obvious, but the mapping between Java return objects and what is returned to a RESTful consumer is not one-to-one. At a minimum, RESTful consumers need to be returned a valid HTTP return code in addition to any response entity body. The mapping of the data contained within a Java object to a response entity is effected by the MIME types a consumer is willing to accept.

To address the issues involved in mapping Java object to RESTful response messages, resource methods are allowed to return four types of Java constructs:

- common Java types return basic information with HTTP return codes determined by the JAX-RS runtime.
- JAXB objects return complex information with HTTP return codes determined by the JAX-RS runtime.
- JAX-RS return complex information with a programmatically determined HTTP return status. The **`Response`** object also allows HTTP headers to be specified.
- JAX-RS return complex information with HTTP return codes determined by the JAX-RS runtime. The **`GenericEnitity`** object provides more information to the runtime components serializing the data.

4.1. RETURNING PLAIN JAVA CONSTRUCTS

Overview

In many cases a resource class can return a standard Java type, a JAXB object, or any object for which the application has an entity provider. In these cases the runtime determines the MIME type information using the Java class of the object being returned. The runtime also determines the appropriate HTTP return code to send to the consumer.

Returnable types

Resource methods can return **`void`** or any Java type for which an entity writer is provided. By default, the runtime has providers for the following:

- the Java primitives
- the **`Number`** representations of the Java primitives
- JAXB objects

the section called "Natively supported types" lists all of the return types supported by default. the section called "Custom writers" describes how to implement a custom entity writer.

MIME types

The runtime determines the MIME type of the returned entity by first checking the resource method and resource class for a **@Produces** annotation. If it finds one, it uses the MIME type specified in the annotation. If it does not find one specified by the resource implementation, it relies on the entity providers to determine the proper MIME type.

By default the runtime assign MIME types as follows:

- Java primitives and their **Number** representations are assigned a MIME type of **application/octet-stream**.
- JAXB objects are assigned a MIME type of **application/xml**.

Applications can use other mappings by implementing custom entity providers as described in the section called "Custom writers".

Response codes

When resource methods return plain Java constructs, the runtime automatically sets the response's status code if the resource method completes without throwing an exception. The status code is set as follows:

- **204**(No Content)—the resource method's return type is **void**
- **204**(No Content)—the value of the returned entity is **null**
- **200**(OK)—the value of the returned entity is not **null**

If an exception is thrown before the resource method completes the return status code is set as described in Chapter 5, *Handling Exceptions*.

4.2. FINE TUNING AN APPLICATION'S RESPONSES

4.2.1. Basics of building responses

Overview

RESTful services often need more precise control over the response returned to a consumer than is allowed when a resource method returns a plain Java construct. The JAX-RS **Response** class allows a resource method to have some control over the return status sent to the consumer and to specify HTTP message headers and cookies in the response.

Response objects wrap the object representing the entity that is returned to the consumer. **Response** objects are instantiated using the **ResponseBuilder** class as a factory.

The **ResponseBuilder** class also has many of the methods used to manipulate the response's metadata. For instance the **ResonseBuilder** class contains the methods for setting HTTP headers and cache control directives.

Relationship between a response and a response builder

The **Response** class has a protected constructor, so they cannot be instantiated directly. They are created using the **ResponseBuilder** class enclosed by the **Response** class. The **ResponseBuilder** class is a holder for all of the information that will be encapsulated in the response created from it. The **ResponseBuilder** class also has all of the methods responsible for setting HTTP header properties on the message.

The **Response** class does provide some methods that ease setting the proper response code and wrapping the entity. There are methods for each of the common response status codes. The methods corresponding to status that include an entity body, or required metadata, include versions that allow for directly setting the information into the associated response builder.

The **ResponseBuilder** class' **build()** method returns a response object containing the information stored in the response builder at the time the method is invoked. After the response object is returned, the response builder is returned to a clean state.

Getting a response builder

There are two ways to get a response builder:

- Using the static methods of the **Response** class as shown in Example 4.1, "Getting a response builder using the **Response** class".

  Example 4.1. Getting a response builder using the Response class

  ```
  import javax.ws.rs.core.Response;

  Response r = Response.ok().build();
  ```

  When getting a response builder this way you do not get access to an instance you can manipulate in multiple steps. You must string all of the actions into a single method call.

- Using the Apache CXF specific **ResponseBuilderImpl** class. This class allows you to work directly with a response builder. However, it requires that you manually set all of the response builders information manually.

  Example 4.2, "Getting a response builder using the **ResponseBuilderImpl** class" shows how Example 4.1, "Getting a response builder using the **Response** class" could be rewritten using the **ResponseBuilderImpl** class.

  Example 4.2. Getting a response builder using the ResponseBuilderImpl class

  ```
  import javax.ws.rs.core.Response;
  import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

  ResponseBuilderImpl builder = new ResponseBuilderImpl();
  builder.status(200);
  Response r = builder.build();
  ```

TIP

You could also simply assign the **ResponseBuilder** returned from a **Response** class' method to a **ResponseBuilderImpl** object.

More information

For more information about the **Response** class see the **Response** class' Javadoc.

For more information about the **ResponseBuilder** class see the **ResponseBuilder** class' Javadoc.

For more information on the Apache CXF **ResponseBuilderIml** class see the **ResponseBuilderImpl** Javadoc.

4.2.2. Creating responses for common use cases

Overview

The **Response** class provides shortcut methods for handling the more common responses that a RESTful service will need. These methods handle setting the proper headers using either provided values or default values. They also handle populating the entity body when appropriate.

Creating responses for successful requests

When a request is successfully processed the application needs to send a response to acknowledge that the request has been fulfilled. That response may contain an entity.

The most common response when successfully completing a response is **OK**. An **OK** response typically contains an entity that corresponds to the request. The **Response** class has an overloaded **ok()** method that sets the response status to **200** and adds a supplied entity to the enclosed response builder. There are five versions of the **ok()** method. The most commonly used variant are:

- **Response.ok()**—creates a response with a status of **200** and an empty entity body.
- **Response.ok(java.lang.Object entity)**—creates a response with a status of **200**, stores the supplied object in the responses entity body, and determines the entities media type by introspecting the object.

Example 4.3, "Creating a response with an **200** response" shows an example of creating a response with an **OK** status.

Example 4.3. Creating a response with an 200 response

```
import javax.ws.rs.core.Response;
import demo.jaxrs.server.Customer;
...

Customer customer = new Customer("Jane", 12);

return Response.ok(customer).build();
```

For cases where the requester is not expecting an entity body, it may be more appropriate to send a **204 No Content** status instead of an **200 OK** status. The **Response.noContent()** method will create an appropriate response object.

Example 4.4, "Creating a response with a 204 status" shows an example of creating a response with an **204** status.

Example 4.4. Creating a response with a 204 status

```
import javax.ws.rs.core.Response;

return Response.noContent().build();
```

Creating responses for redirection

The **Response** class provides methods for handling three of the redirection response statuses.

303 See Other

The **303 See Other** status is useful when the requested resource needs to permanently redirect the consumer to a new resource to process the request.

The **Response** classes **seeOther()** method creates a response with a **303** status and places the new resource URI in the message's **Location** field. The **seeOther()** method takes a single parameter that specifies the new URI as a **java.net.URI** object.

304 Not Modified

The **304 Not Modified** status can be used for different things depending on the nature of the request. It can be used to signify that the requested resource has not changed since a previous **GET** request. It can also be used to signify that a request to modify the resource did not result in the resource being changed.

The **Response** classes **notModified()** methods creates a response with a **304** status and sets the modified date property on the HTTP message. There are three versions of the **notModified()** method:

- **notModified();**
- **notModified(javax.ws.rs.core.Entity tag);**
- **notModified(java.lang.String tag);**

307 Temporary Redirect

The **307 Temporary Redirect** status is useful when the requested resource needs to direct the consumer to a new resource, but wants the consumer to continue using this resource to handle future requests.

The **Response** classes **temporaryRedirect()** method creates a response with a **307** status and places the new resource URI in the message's **Location** field. The **temporaryRedirect()** method takes a single parameter that specifies the new URI as a **java.net.URI** object.

Example 4.5, "Creating a response with a `304` status" shows an example of creating a response with an **304** status.

Example 4.5. Creating a response with a `304` status

```
import javax.ws.rs.core.Response;

return Response.notModified().build();
```

Creating responses to signal errors

The **Response** class provides methods to create responses for two basic processing errors:

- **`serverError()();`**—creates a response with a status of **`500 Internal Server Error`**.
- **`notAcceptable()(java.util.List<javax.ws.rs.core.Variant> variants);`**—creates a response with a **`406 Not Acceptable`** status and an entity body containing a list of acceptable resource types.

Example 4.6, "Creating a response with a `500` status" shows an example of creating a response with an **500** status.

Example 4.6. Creating a response with a `500` status

```
import javax.ws.rs.core.Response;

return Response.serverError().build();
```

4.2.3. Handling more advanced responses

Overview

The **Response** class methods provide short cuts for creating responses for common cases. When you need to address more complicated cases such as specifying cache control directives, adding custom HTTP headers, or sending a status not handled by the **Response** class, you need to use the **`ResponseBuilder`** classes methods to populate the response before using the **`build()`** method to generate the response object.

TIP

As discussed in the section called "Getting a response builder" , you can use the Apache CXF **`ResponseBuilderImpl`** class to create a response builder instance that can be manipulated directly.

Adding custom headers

Custom headers are added to a response using the **`ResponseBuilder`** class' **`header()`** method. The **`header()`** method takes two parameters:

- *name*—a string specifying the name of the header

- ***value***—a Java object containing the data stored in the header

You can set multiple headers on the message by calling the **`header()`** method repeatedly.

Example 4.7, "Adding a header to a response" shows code for adding a header to a response.

Example 4.7. Adding a header to a response

```
import javax.ws.rs.core.Response;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.header("username", "joe");
Response r = builder.build();
```

Adding a cookie

Custom headers are added to a response using the **`ResponseBuilder`** class' **`cookie()`** method. The **`cookie()`** method takes one or more cookies. Each cookie is stored in a **`javax.ws.rs.core.NewCookie`** object. The easiest of the **`NewCookie`** class' contructors to use takes two parameters:

- ***name***—a string specifying the name of the cookie
- ***value***—a string specifying the value of the cookie

You can set multiple cookies by calling the **`cookie()`** method repeatedly.

Example 4.8, "Adding a cookie to a response" shows code for adding a cookie to a response.

Example 4.8. Adding a cookie to a response

```
import javax.ws.rs.core.Response;
import javax.ws.rs.core.NewCookie;

NewCookie cookie = new NewCookie("username", "joe");

Response r = Response.ok().cookie(cookie).build();
```

WARNING

Calling the **`cookie()`** method with a **`null`** parameter list erases any cookies already associated with the response.

Setting the response status

When you want to return a status other than one of the statuses supported by the **Response** class' helper methods, you can use the **ResponseBuilder** class' **status()** method to set the response's status code. The **status()** method has two variants. One takes an int that specifies the response code. The other takes a **Response.Status** object to specify the response code.

The **Response.Status** class is an enumeration enclosed in the **Response** class. It has entries for most of the defined HTTP response codes.

Example 4.9, "Adding a header to a response" shows code for setting the response status to **404 Not Found**.

Example 4.9. Adding a header to a response

```
import javax.ws.rs.core.Response;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.status(404);
Response r = builder.build();
```

Setting cache control directives

The **ResponseBuilder** class' **cacheControl()** method allows you to set the cache control headers on the response. The **cacheControl()** method takes a **javax.ws.rs.CacheControl** object that specifies the cache control directives for the response.

The **CacheControl** class has methods that correspond to all of the cache control directives supported by the HTTP specification. Where the directive is a simple on or off value the setter method takes a boolean value. Where the directive requires a numeric value, such as the **max-age** directive, the setter takes an int value.

Example 4.10, "Adding a header to a response" shows code for setting the **no-store** cache control directive.

Example 4.10. Adding a header to a response

```
import javax.ws.rs.core.Response;
import javax.ws.rs.core.CacheControl;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

CacheControl cache = new CacheControl();
cache.setNoCache(true);

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.cacheControl(cache);
Response r = builder.build();
```

4.3. RETURNING ENTITIES WITH GENERIC TYPE INFORMATION

Overview

There are occasions where the application needs more control over the MIME type of the returned object or the entity provider used to serialize the response. The JAX-RS **`javax.ws.rs.core.GenericEntity<T>`** class provides finer control over the serializing of entities by providing a mechanism for specifying the generic type of the object representing the entity.

Using a `GenericEntity<T>` object

One of the criteria used for selecting the entity provider that serializes a response is the generic type of the object. The generic type of an object represents the Java type of the object. When a common Java type or a JAXB object is returned, the runtime can use Java reflection to determine the generic type. However, when a JAX-RS **`Response`** object is returned, the runtime cannot determine the generic type of the wrapped entity and the actual Java class of the object is used as the Java type.

To ensure that the entity provider is provided with correct generic type information, the entity can be wrapped in a **`GenericEntity<T>`** object before being added to the **`Response`** object being returned.

Resource methods can also directly return a **`GenericEntity<T>`** object. In practice, this approach is rarely used. The generic type information determined by reflection of an unwrapped entity and the generic type information stored for an entity wrapped in a **`GenericEntity<T>`** object are typically the same.

Creating a `GenericEntity<T>` object

There are two ways to create a **`GenericEntity<T>`** object:

1. Create a subclass of the **`GenericEntity<T>`** class using the entity being wrapped. Example 4.11, "Creating a **`GenericEntity<T>`** object using a subclass" shows how to create a **`GenericEntity<T>`** object containing an entity of type **`List<String>`** whose generic type will be available at runtime.

   Example 4.11. Creating a `GenericEntity<T>` object using a subclass

   ```
   import javax.ws.rs.core.GenericEntity;

   List<String> list = new ArrayList<String>();
   ...
   GenericEntity<List<String>> entity =
     new GenericEntity<List<String>>(list) {};
   Response response = Response.ok(entity).build();
   ```

   TIP

   The subclass used to create a **`GenericEntity<T>`** object is typically anonymous.

2. Create an instance directly by supplying the generic type information with the entity. Example 4.12, "Directly instantiating a **`GenericEntity<T>`** object" shows how to create a response containing an entity of type **`AtomicInteger`**.

   Example 4.12. Directly instantiating a `GenericEntity<T>` object

   ```
   import javax.ws.rs.core.GenericEntity;
   ```

```
AtomicInteger result = new AtomicInteger(12);
GenericEntity<AtomicInteger> entity =
  new GenericEntity<AtomicInteger>(result,

result.getClass().getGenericSuperclass());
Response response = Response.ok(entity).build();
```

CHAPTER 5. HANDLING EXCEPTIONS

Abstract

When possible, exceptions caught by a resource method should cause a useful error to be returned to the requesting consumer. JAX-RS resource methods can throw a **WebApplicaitonException** exception. You can also provide **ExceptionMapper<E>** implementations to map exceptions to appropriate responses.

5.1. USING WEBAPPLICAITONEXCEPTION EXCEPTIONS TO REPORT ERRORS

Overview

The JAX-RS API introduced the **WebApplicationException** runtime exception to provide an easy way for resource methods to create exceptions that are appropriate for RESTful clients to consume. **WebApplicationException** exceptions can include a **Response** object that defines the entity body to return to the originator of the request. It also provides a mechanism for specifying the HTTP status code to be returned to the client if no entity body is provided.

Creating a simple exception

The easiest means of creating a **WebApplicaitonException** exception is to use either the no argument constructor or the constructor that wraps the original exception in a **WebApplicationException** exception. Both constructors create a **WebApplicaitonException** with an empty response.

When an exception created by either of these constructors is thrown, the runtime returns a response with an empty entity body and a status code of **500 Server Error**.

Setting the status code returned to the client

When you want to return an error code other than **500**, you can use one of the four **WebApplicaitonException** constructors that allow you to specify the status. Two of these constructors, shown in Example 5.1, "Creating a **WebApplicationException** with a status code", take the return status as an integer.

Example 5.1. Creating a WebApplicationException with a status code

```
WebApplicationException(int status);
WebApplicationException(java.lang.Throwable cause,
                        int status);
```

The other two, shown in Example 5.2, "Creating a **WebApplicationException** with a status code" take the response status as an instance of **Response.Status**.

Example 5.2. Creating a WebApplicationException with a status code

```
WebApplicationException(javax.ws.rs.core.Response.Status status);
WebApplicationException(java.lang.Throwable cause,
                        javax.ws.rs.core.Response.Status status);
```

When an exception created by either of these constructors is thrown, the runtime returns a response with an empty entity body and the specified status code.

Providing an entity body

If you want a message to be sent along with the exception, you can use one of the **`WebApplicationException`** constructors that takes a **`Response`** object. The runtime uses the **`Response`** object to create the response sent to the client. The entity stored in the response is mapped to the entity body of the message and the status field of the response is mapped to the HTTP status of the message.

Example 5.3, "Sending a message with an exception" shows code for returning a text message to a client containing the reason for the exception and sets the HTTP message status to **`409 Conflict`**.

Example 5.3. Sending a message with an exception

```
import javax.ws.rs.core.Response;
import javax.ws.rs.WebApplicationException;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

...
ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.status(Response.Status.CONFLICT);
builder.entity("The requested resource is conflicted.");
Response response = builder.build();
throw WebApplicationException(response);
```

Extending the generic exception

It is possible to extend the **`WebApplicationException`** exception. This would allow you to create custom exceptions and eliminate some boiler plate code.

Example 5.4, "Extending **`WebApplicationException`**" shows a new exception that creates a similar response to the code in Example 5.3, "Sending a message with an exception" .

Example 5.4. Extending `WebApplicationException`

```
public class ConflicteddException extends WebApplicationException
{
  public ConflictedException(String message)
  {
    ResponseBuilderImpl builder = new ResponseBuilderImpl();
    builder.status(Response.Status.CONFLICT);
    builder.entity(message);
    super(builder.build());
     }
}

...
throw ConflictedException("The requested resource is conflicted.");
```

5.2. MAPPING EXCEPTIONS TO RESPONSES

Overview

There are instances where throwing a **`WebApplicationException`** exception is impractical or impossible. For example, you may not want to catch all possible exceptions and then create a **`WebApplicationException`** for them. You may also want to use custom exceptions that make working with your application code easier.

To handle these cases the JAX-RS API allows you to implement a custom exception provider that generates a **`Response`** object to send to a client. Custom exception providers are created by implementing the **`ExceptionMapper<E>`** interface. When registered with the Apache CXF runtime, the custom provider will be used whenever an exception of type E is thrown.

How exception mappers are selected

Exception mappers are used in two cases:

- When a **`WebApplicationException`**, or one of its subclasses, with an empty entity body is thrown, the runtime will check to see if there is an exception mapper that handles **`WebApplicationException`** exceptions. If there is the exception mapper is used to create the response sent to the consumer.
- When any exception other than a **`WebApplicationException`** exception, or one of its subclasses, is thrown, the runtime will check for an appropriate exception mapper. An exception mapper is selected if it handles the specific exception thrown. If there is not an exception mapper for the specific exception that was thrown, the exception mapper for the nearest superclass of the exception is selected.

If an exception mapper is not found for an exception, the exception is wrapped in an **`ServletException`** exception and passed onto the container runtime. The container runtime will then determine how to handle the exception.

Implementing an exception mapper

Exception mappers are created by implementing the **`javax.ws.rs.ext.ExceptionMapper<E>`** interface. As shown in Example 5.5, “Exception mapper interface” , the interface has a single method, **`toResponse()`**, that takes the original exception as a parameter and returns a **`Response`** object.

Example 5.5. Exception mapper interface

```
public interface ExceptionMapper<E extends java.lang.Throwable>
{
  public Response toResponse(E exception);
}
```

The **`Response`** object created by the exception mapper is processed by the runtime just like any other **`Response`** object. The resulting response to the consumer will contain the status, headers, and entity body encapsulated in the **`Response`** object.

Exception mapper implementations are considered providers by the runtime. Therefor they must be decorated with the **@Provder** annotation.

If an exception occurs while the exception mapper is building the **Response** object, the runtime will a response with a status of **500 Server Error** to the consumer.

Example 5.6, "Mapping an exception to a response" shows an exception mapper that intercepts Spring **AccessDeniedException** exceptions and generates a response with a **403 Forbidden** status and an empty entity body.

Example 5.6. Mapping an exception to a response

```
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;

import org.springframework.security.AccessDeniedException;

@Provider
public class SecurityExceptionMapper implements
ExceptionMapper<AccessDeniedException>
{

  public Response toResponse(AccessDeniedException exception)
  {
    return Response.status(Response.Status.FORBIDDEN).build();
  }

}
```

The runtime will catch any **AccessDeniedException** exceptions and create a **Response** object with no entity body and a status of **403**. The runtime will then process the **Response** object as it would for a normal response. The result is that the consumer will receive an HTTP response with a status of **403**.

Registering an exception mapper

Before a JAX-RS application can use an exception mapper, the exception mapper must be registered with the runtime. Exception mappers are registered with the runtime using the **jaxrs:providers** element in the application's configuration file.

The **jaxrs:providers** element is a child of the **jaxrs:server** element and contains a list of **bean** elements. Each **bean** element defines one exception mapper.

Example 5.7, "Registering exception mappers with the runtime" shows a JAX-RS server configured to use a an exception mapper.

Example 5.7. Registering exception mappers with the runtime

```
<beans ...>
  <jaxrs:server id="customerService" address="/">
    ...
    <jaxrs:providers>
      <bean id="securityException"
class="com.bar.providers.SecurityExceptionMapper"/>
```

```
    </jaxrs:providers>
  </jaxrs:server>
</beans>
```

CHAPTER 6. PUBLISHING A SERVICE

Abstract

How you publish a RESTful Web service depends on the runtime environment. Apache CXF allows you to publish RESTful Web services as standalone applications. You can also publish them using Spring, a servlet container, or an OSGi container.

CHAPTER 7. ENTITY SUPPORT

Abstract

The Apache CXF runtime supports a limited number of mappings between MIME types and Java objects out of the box. Developers can extend the mappings by implementing custom readers and writers. The custom readers and writers are registered with the runtime at start-up.

OVERVIEW

The runtime relies on JAX-RS `MessageBodyReader` and `MessageBodyWriter` implementations to serialize and de-serialize data between the HTTP messages and their Java representations. The readers and writers can restrict the MIME types they are capable of processing.

The runtime provides readers and writers for a number of common mappings. If an application requires more advanced mappings, a developer can provide custom implementations of the `MessageBodyReader` interface and/or the `MessageBodyWriter` interface. Custom readers and writers are registered with the runtime when the application is started.

NATIVELY SUPPORTED TYPES

Table 7.1, “Natively supported entity mappings” lists the entity mappings provided by Apache CXF out of the box.

Table 7.1. Natively supported entity mappings

Java Type	MIME Type
primitive types	`text/plain`
`java.lang.Number`	`text/plain`
byte[]	`*/*`
`java.lang.String`	`*/*`
`java.io.InputStream`	`*/*`
`java.io.Reader`	`*/*`
`java.io.File`	`*/*`
`javax.activation.DataSource`	`*/*`
`javax.xml.transform.Source`	`text/xml`, `application/xml`, `application/*+xml`
`javax.xml.bind.JAXBElement`	`text/xml`, `application/xml`, `application/*+xml`

Java Type	MIME Type
JAXB annotated objects	**text/xml**, **application/xml**, **application/*+xml**
javax.ws.rs.core.MultivaluedMap<String, String>	**application/x-www-form-urlencoded** [a]
javax.ws.rs.core.StreamingOutput	***/*** [b]

[a] This mapping is used for handling HTML form data.

[b] This mapping is only supported for returning data to a consumer.

CUSTOM READERS

Custom entity readers are responsible for mapping incoming HTTP requests into a Java type that a service's implementation can manipulate. They implement the **javax.ws.rs.ext.MessageBodyReader** interface.

The interface, shown in Example 7.1, "Message reader interface", has two methods that need implementing:

Example 7.1. Message reader interface

```
package javax.ws.rs.ext;

public interface MessageBodyReader<T>
{
  public boolean isReadable(java.lang.Class<?> type,
                            java.lang.reflect.Type genericType,
                            java.lang.annotation.Annotation[]
annotations,
                            javax.ws.rs.core.MediaType mediaType);

  public T readFrom(java.lang.Class<T> type,
                    java.lang.reflect.Type genericType,
                    java.lang.annotation.Annotation[] annotations,
                    javax.ws.rs.core.MediaType mediaType,
                    javax.ws.rs.core.MultivaluedMap<String, String>
httpHeaders,
                    java.io.InputStream entityStream)
  throws java.io.IOException, WebApplicationException;
}
```

isReadable()

The **isReadable()** method determines if the reader is capable of reading the data stream and creating the proper type of entity representation. If the reader can create the proper type of entity the method returns **true**.

Table 7.2, “Parameters used to determine if a reader can produce an entity” describes the `isReadable()` method's parameters.

Table 7.2. Parameters used to determine if a reader can produce an entity

Parameter	Type	Description
type	`Class<T>`	Specifies the actual Java class of the object used to store the entity.
genericType	`Type`	Specifies the Java type of the object used to store the entity. For example, if the message body is to be converted into a method parameter, the value will be the type of the method parameter as returned by the `Method.getGenericParameterTypes()` method.
annotations	`Annotation[]`	Specifies the list of annotations on the declaration of the object created to store the entity. For example if the message body is to be converted into a method parameter, this will be the annotations on that parameter returned by the `Method.getParameterAnnotations()` method.
mediaType	`MediatType`	Specifies the MIME type of the HTTP entity.

`readFrom()`

The `readFrom()` method reads the HTTP entity and coverts it into the desired Java object. If the reading is successful the method returns the created Java object containing the entity. If an error occurs when reading the input stream the method should throw an `IOException` exception. If an error occurs that requires an HTTP error response, an `WebApplicationException` with the HTTP response should be thrown.

Table 7.3, “Parameters used to read an entity” describes the `readFrom()` method's parameters.

Table 7.3. Parameters used to read an entity

Parameter	Type	Description
type	`Class<T>`	Specifies the actual Java class of the object used to store the entity.

Parameter	Type	Description
genericType	**Type**	Specifies the Java type of the object used to store the entity. For example, if the message body is to be converted into a method parameter, the value will be the type of the method parameter as returned by the **Method.getGenericParameterTypes()** method.
annotations	**Annotation[]**	Specifies the list of annotations on the declaration of the object created to store the entity. For example if the message body is to be converted into a method parameter, this will be the annotations on that parameter returned by the **Method.getParameterAnnotations()** method.
mediaType	**MediatType**	Specifies the MIME type of the HTTP entity.
httpHeaders	**MultivaluedMap<String, String>**	Specifies the HTTP message headers associated with the entity.
entityStream	**InputStream**	Specifies the input stream containing the HTTP entity.

IMPORTANT

This method should not close the input stream.

Before an **MessageBodyReader** implementation can be used as an entity reader, it must be decorated with the **javax.ws.rs.ext.Provider** annotation. The **@Provider** annotation alerts the runtime that the supplied implementation provides additional functionality. The implementation must also be registered with the runtime as described in the section called “Registering readers and writers”.

By default a custom entity provider handles all MIME types. You can limit the MIME types that a custom entity reader will handle using the **javax.ws.rs.Consumes** annotation. The **@Consumes** annotation specifies a comma separated list of MIME types that the custom entity provider reads. If an entity is not of a specified MIME type, the entity provider will not be selected as a possible reader.

Example 7.2, “XML source entity reader” shows an entity reader the consumes XML entities and stores them in a **Source** object.

Example 7.2. XML source entity reader

```
import java.io.IOException;
import java.io.InputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

import javax.ws.rs.Consumes;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.ext.MessageBodyReader;
import javax.ws.rs.ext.Provider;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Source;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamSource;

import org.w3c.dom.Document;
import org.apache.cxf.jaxrs.ext.xml.XMLSource;

@Provider
@Consumes({"application/xml", "application/*+xml", "text/xml",
"text/html" })
public class SourceProvider implements MessageBodyReader<Object>
{
  public boolean isReadable(Class<?> type,
                            Type genericType,
                            Annotation[] annotations,
                            MediaType mt)
  {
    return Source.class.isAssignableFrom(type) ||
XMLSource.class.isAssignableFrom(type);
  }

  public Object readFrom(Class<Object> source,
                         Type genericType,
                         Annotation[] annotations,
                         MediaType mediaType,
                         MultivaluedMap<String, String> httpHeaders,
                         InputStream is)
  throws IOException
  {
    if (DOMSource.class.isAssignableFrom(source))
    {
      Document doc = null;
      DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
      DocumentBuilder builder;
      try
      {
        builder = factory.newDocumentBuilder();
        doc = builder.parse(is);
      }
      catch (Exception e)
      {
        IOException ioex = new IOException("Problem creating a Source
```

```
object");
        ioex.setStackTrace(e.getStackTrace());
        throw ioex;
       }

       return new DOMSource(doc);
    }
    else if (StreamSource.class.isAssignableFrom(source) ||
Source.class.isAssignableFrom(source))
    {
      return new StreamSource(is);
    }
    else if (XMLSource.class.isAssignableFrom(source))
    {
      return new XMLSource(is);
    }

    throw new IOException("Unrecognized source");
  }
}
```

CUSTOM WRITERS

Custom entity writers are responsible for mapping Java types into HTTP entities. They implement the **`javax.ws.rs.ext.MessageBodyWriter`** interface.

The interface, shown in Example 7.3, “Message writer interface” , has three methods that need implementing:

Example 7.3. Message writer interface

```
package javax.ws.rs.ext;

public interface MessageBodyWriter<T>
{
  public boolean isWriteable(java.lang.Class<?> type,
                             java.lang.reflect.Type genericType,
                             java.lang.annotation.Annotation[]
annotations,
                             javax.ws.rs.core.MediaType mediaType);

  public long getSize(T t,
                      java.lang.Class<?> type,
                      java.lang.reflect.Type genericType,
                      java.lang.annotation.Annotation[] annotations,
                      javax.ws.rs.core.MediaType mediaType);

  public void writeTo(T t,
                      java.lang.Class<?> type,
                      java.lang.reflect.Type genericType,
                      java.lang.annotation.Annotation[] annotations,
                      javax.ws.rs.core.MediaType mediaType,
                      javax.ws.rs.core.MultivaluedMap<String, Object>
httpHeaders,
```

```
                        java.io.OutputStream entityStream)
    throws java.io.IOException, WebApplicationException;
}
```

isWriteable()

The **isWriteable()** method determines if the entity writer can map the Java type to the proper entity type. If the writer can do the mapping, the method returns **true**.

Table 7.4, "Parameters used to read an entity" describes the **isWritable()** method's parameters.

Table 7.4. Parameters used to read an entity

Parameter	Type	Description
type	**Class<T>**	Specifies the Java class of the object being written.
genericType	**Type**	Specifies the Java type of object to be written, obtained either by reflection of a resource method return type or via inspection of the returned instance. The **GenericEntity** class, described in Section 4.3, "Returning entities with generic type information", provides support for controlling this value.
annotations	**Annotation[]**	Specifies the list of annotations on the method returning the entity.
mediaType	**MediatType**	Specifies the MIME type of the HTTP entity.

getSize()

The **getSize()** method is called before the **writeTo()**. It returns the length, in bytes, of the entity being written. If a positive value is returned the value is written into the HTTP message's **Content-Length** header.

Table 7.5, "Parameters used to read an entity" describes the **getSize()** method's parameters.

Table 7.5. Parameters used to read an entity

Parameter	Type	Description
t	generic	Specifies the instance being written.

Parameter	Type	Description
type	`Class<T>`	Specifies the Java class of the object being written.
genericType	`Type`	Specifies the Java type of object to be written, obtained either by reflection of a resource method return type or via inspection of the returned instance. The `GenericEntity` class, described in Section 4.3, "Returning entities with generic type information", provides support for controlling this value.
annotations	`Annotation[]`	Specifies the list of annotations on the method returning the entity.
mediaType	`MediatType`	Specifies the MIME type of the HTTP entity.

`writeTo()`

The `writeTo()` method converts a Java object into the desired entity type and writes the entity to the output stream. If an error occurs when writing the entity to the output stream the method should throw an `IOException` exception. If an error occurs that requires an HTTP error response, an `WebApplicationException` with the HTTP response should be thrown.

Table 7.6, "Parameters used to read an entity" describes the `writeTo()` method's parameters.

Table 7.6. Parameters used to read an entity

Parameter	Type	Description
t	generic	Specifies the instance being written.
type	`Class<T>`	Specifies the Java class of the object being written.

Parameter	Type	Description
genericType	**Type**	Specifies the Java type of object to be written, obtained either by reflection of a resource method return type or via inspection of the returned instance. The **GenericEntity** class, described in Section 4.3, "Returning entities with generic type information", provides support for controlling this value.
annotations	**Annotation[]**	Specifies the list of annotations on the method returning the entity.
mediaType	**MediatType**	Specifies the MIME type of the HTTP entity.
httpHeaders	**MultivaluedMap<String, Object>**	Specifies the HTTP response headers associated with the entity.
entityStream	**OutputStream**	Specifies the output stream into which the entity is written.

Before a **MessageBodyWriter** implementation can be used as an entity writer, it must be decorated with the **javax.ws.rs.ext.Provider** annotation. The **@Provider** annotation alerts the runtime that the supplied implementation provides additional functionality. The implementation must also be registered with the runtime as described in the section called "Registering readers and writers".

By default a custom entity provider handles all MIME types. You can limit the MIME types that a custom entity writer will handle using the **javax.ws.rs.Produces** annotation. The **@Produces** annotation specifies a comma separated list of MIME types that the custom entity provider generates. If an entity is not of a specified MIME type, the entity provider will not be selected as a possible writer.

Example 7.4, "XML source entity writer" shows an entity writer that takes **Source** objects and produces XML entities.

Example 7.4. XML source entity writer

```
import java.io.IOException;
import java.io.OutputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

import javax.ws.rs.Produces;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
```

```
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.ext.MessageBodyWriter;
import javax.ws.rs.ext.Provider;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;

import org.apache.cxf.jaxrs.ext.xml.XMLSource;

@Provider
@Produces({"application/xml", "application/*+xml", "text/xml" })
public class SourceProvider implements MessageBodyWriter<Source>
{

  public boolean isWriteable(Class<?> type,
                             Type genericType,
                             Annotation[] annotations,
                             MediaType mt)
  {
    return Source.class.isAssignableFrom(type);
  }

  public void writeTo(Source source,
                      Class<?> clazz,
                      Type genericType,
                      Annotation[] annotations,
                      MediaType mediatype,
                      MultivaluedMap<String, Object> httpHeaders,
                      OutputStream os)
  throws IOException
  {
    StreamResult result = new StreamResult(os);
    TransformerFactory tf = TransformerFactory.newInstance();
    try
    {
      Transformer t = tf.newTransformer();
      t.transform(source, result);
    }
    catch (TransformerException te)
    {
      te.printStackTrace();
      throw new WebApplicationException(te);
    }
  }

  public long getSize(Source source,
                      Class<?> type,
                      Type genericType,
                      Annotation[] annotations,
                      MediaType mt)
  {
```

```
    return -1;
  }
}
```

REGISTERING READERS AND WRITERS

Before a JAX-RS application can use any custom entity providers, the custom providers must be registered with the runtime. Providers are registered with the runtime using either the **jaxrs:providers** element in the application's configuration file or using the **JAXRSServerFactoryBean** class.

The **jaxrs:providers** element is a child of the **jaxrs:server** element and contains a list of **bean** elements. Each **bean** element defines one entity provider.

Example 7.5, "Registering entity providers with the runtime" show a JAX-RS server configured to use a set of custom entity providers.

Example 7.5. Registering entity providers with the runtime

```
<beans ...>
  <jaxrs:server id="customerService" address="/">
    ...
    <jaxrs:providers>
      <bean id="isProvider"
class="com.bar.providers.InputStreamProvider"/>
      <bean id="longProvider" class="com.bar.providers.LongProvider"/>
    </jaxrs:providers>
  </jaxrs:server>
</beans>
```

The **JAXRSServerFactoryBean** class is a Apache CXF extension that provides access to the configuration APIs. It has a **setProvider()** method that allows you to add instantiated entity providers to an application. Example 7.6, "Programmatically registering an entity provider" shows code for registering an entity provider programmatically.

Example 7.6. Programmatically registering an entity provider

```
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
...
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
...
SourceProvider provider = new SourceProvider();
sf.setProvider(provider);
...
```

CHAPTER 8. CUSTOMIZING THE MEDIA TYPES HANDLED BY A RESOURCE

Abstract

By default, resources process all requests using the media type specifier `*/*`. You can restrict the media types a resource will process using annotations.

CHAPTER 9. GETTING AND USING CONTEXT INFORMATION

Abstract

Context information includes detailed information about a resource's URI, the HTTP headers, and other details that are not readily available using the other injection annotations. Apache CXF provides special class that amalgamates the all possible context information into a single object.

9.1. INTRODUCTION TO CONTEXTS

Context annotation

You specify that context information is to be injected into a field or a resource method parameter using the `javax.ws.rs.core.Context` annotation. Annotating a field or parameter of one of the context types will instruct the runtime to inject the appropriate context information into the annotated field or parameter.

Types of contexts

Table 9.1, “Context types” lists the types of context information that can be injected and the objects that support them.

Table 9.1. Context types

Object	Context information
`UriInfo`	The full request URI
`HttpHeaders`	The HTTP message headers
`Request`	Information that can be used to determine the best representation variant or to determine if a set of preconditions have been set
`SecurityContext`	Information about the security of the requester including the authentication scheme in use, if the request channel is secure, and the user principle

Where context information can be used

Context information is available to the following parts of a JAX-RS application:

- resource classes
- resource methods
- entity providers
- exception mappers

Scope

All context information injected using the **@Context** annotation is specific to the current request. This is true in all cases including entity providers and exception mappers.

Adding contexts

The JAX-RS framework allows developers to extend the types of information that can be injected using the context mechanism. You add custom contexts by implementing a **Context<T>** object and registering it with the runtime.

For information on creating custom contexts see Section 9.8, “Adding custom contexts”.

9.2. WORKING WITH THE FULL REQUEST URI

The request URI contains a significant amount of information. Most of this information can be accessed using method parameters as described in Section 3.2.1, “Injecting data from a request URI”, however using parameters forces certain constraints on how the URI is processed. Using parameters to access the segments of a URI also does not provide a resource access to the full request URI.

You can provide access to the complete request URI by injecting the URI context into a resource. The URI is provided as a **UriInfo** object. The **UriInfo** interface provides functions for decomposing the URI in a number of ways. It can also provide the URI as a **UriBuilder** object that allows you to construct URIs to return to clients.

9.2.1. Injecting the URI information

Overview

When a class field or method parameter that is a **UriInfo** object is decorated with the **@Context** annotation, the URI context for the current request is injected into the **UriInfo** object.

Example

Example 9.1, “Injecting the URI context into a class field” shows a class with a field populated by injecting the URI context.

Example 9.1. Injecting the URI context into a class field

```
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.Path;
...
@Path("/monstersforhire/")
public class MonsterService
{
  @Context
  UriInfo requestURI;
  ...
}
```

9.2.2. Working with the URI

Overview

One of the main advantages of using the URI context is that it provides access to the base URI of the service and the path segment of the URI for the selected resource. This information can be useful for a number of purposes such as making processing decisions based on the URI or calculating URIs to return as part of the response. For example if the base URI of the request contains a .com extension the service may decide to use US dollars and if the base URI contains a .co.uk extension is may decide to us British Pounds.

The **`UriInfo`** interface provides methods for accessing the parts of the URI:

- the base URI
- the resource path
- the full URI

Getting the Base URI

The *base URI* is the root URI on which the service is published. It does not contain any portion of the URI specified in any of the service's **`@Path`** annotations. For example if a service implementing the resource defined in Example 3.5, "Disabling URI decoding" were published to http://fusesource.org and a request was made on http://fusesource.org/montersforhire/nightstalker?12 the base URI would be http://fusesource.org.

Table 9.2, "Methods for accessing a resource's base URI" describes the methods that return the base URI.

Table 9.2. Methods for accessing a resource's base URI

Method	Desription
`URI getBaseUri();`	Returns the service's base URI as a **`URI`** object.
`UriBuilder getBaseUriBuilder();`	Returns the base URI as a **`javax.ws.rs.core.UriBuilder`** object. The **`UriBuilder`** class is useful for creating URIs for other resources implemented by the service.

Getting the path

The *path* portion of the request URI is the portion of the URI that was used to select the current resource. It does not include the base URI, but does include any URI template variable and matrix parameters included in the URI.

The value of the path depends on the resource selected. For example, the paths for the resources defined in Example 9.2, "Getting a resource's path" would be:

- **`rootPath`** — /monstersforhire/
- **`getterPath`** — /mostersforhire/nightstalker

The **GET** request was made on /monstersforhire/nightstalker.

- **putterPath** — /mostersforhire/911

  The **PUT** request was made on /monstersforhire/911.

Example 9.2. Getting a resource's path

```
@Path("/monstersforhire/")
public class MonsterService
{
  @Context
  UriInfo rootUri;

  ...

  @GET
  public List<Monster> getMonsters(@Context UriInfo getUri)
  {
    String rootPath = rootUri.getPath();
    ...
  }

  @GET
  @Path("\{type}")
  public Monster getMonster(@PathParam("type") String type,
                            @Context UriInfo getUri)
  {
    String getterPath = getUri.getPath();
    ...
  }

  @PUT
  @Path("\{id}")
  public void addMonster(@Encoded @PathParam("type") String type,
                         @Context UriInfo putUri)
  {
    String putterPath = putUri.getPath();
    ...
  }
  ...
}
```

Table 9.3, “Methods for accessing a resource's path” describes the methods that return the resource path.

Table 9.3. Methods for accessing a resource's path

Method	Desription
`String getPath();`	Returns the resource's path as a decoded URI.

Method	Desription
`String getPath(boolean decode);`	Returns the resource's path. Specifying `false` disables URI decoding.
`List<PathSegment> getPathSegments();`	Returns the decoded path as a list of `javax.ws.rs.core.PathSegment` objects. Each portion of the path, including matrix parameters, is placed into a unique entry in the list. For example the resource path box/round#tall would result in a list with three entries: `box`, `round`, and `tall`.
`List<PathSegment> getPathSegments(boolean decode);`	Returns the path as a list of `javax.ws.rs.core.PathSegment` objects. Each portion of the path, including matrix parameters, is placed into a unique entry in the list. Specifying `false` disables URI decoding. For example the resource path box#tall/round would result in a list with three entries: `box`, `tall`, and `round`.

Getting the full request URI

Table 9.4, "Methods for accessing the full request URI" describes the methods that return the full request URI. You have the option of returning the request URI or the absolute path of the resource. The difference is that the request URI includes the any query parameters appended to the URI and the absolute path does not include the query parameters.

Table 9.4. Methods for accessing the full request URI

Method	Desription
`URI getRequestUri();`	Returns the complete request URI, including query parameters and matrix parameters, as a `java.net.URI` object.
`UriBuilder getRequestUriBuilder();`	Returns the complete request URI, including query parameters and matrix parameters, as a `javax.ws.rs.UriBuilder` object. The `UriBuilder` class is useful for creating URIs for other resources implemented by the service.
`URI getAbsolutePath();`	Returns the complete request URI, including matrix parameters, as a `java.net.URI` object. The absolute path does not include query parameters.

Method	Desription
`UriBuilder getAbsolutePathBuilder();`	Returns the complete request URI, including matrix parameters, as a `javax.ws.rs.UriBuilder` object. The absolute path does not include query parameters.

For a request made using the URI http://fusesource.org/montersforhire/nightstalker?12, the `getRequestUri()` methods would return http://fusesource.org/montersforhire/nightstalker?12. The `getAbsolutePath()` method would return http://fusesource.org/montersforhire/nightstalker.

9.2.3. Getting the value of URI template variables

Overview

As described in the section called "Setting the path", resource paths can contain variable segments that are bound to values dynamically. Often these variable path segments are used as parameters to a resource method as described in the section called "Getting data from the URI's path" . You can, however, also access them through the URI context.

Methods for getting the path parameters

The `UriInfo` interface provides two methods, shown in Example 9.3, "Methods for returning path parameters from the URI context", that return a list of the path parameters.

Example 9.3. Methods for returning path parameters from the URI context

```
MultivaluedMap<java.lang.String, java.lang.String> getPathParameters();
MultivaluedMap<java.lang.String,
java.lang.String> getPathParameters(boolean decode);
```

The `getPathParameters()` method that does not take any parameters automatically decodes the path parameters. If you want to disable URI decoding use `getPathParameters(false)`.

The values are stored in the map using their template identifiers as keys. For example if the URI template for the resource is /{color}/box/{note} the returned map will have two entries with the keys `color` and `note`.

Example

Example 9.4, "Extracting path parameters from the URI context" shows code for retrieving the path parameters using the URI context.

Example 9.4. Extracting path parameters from the URI context

```
import javax.ws.rs.Path;
import javax.ws.rs.Get;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.core.MultivaluedMap;
```

```
@Path("/monstersforhire/")
public class MonsterService

  @GET
  @Path("\{type}\{size}")
  public Monster getMonster(@Context UriInfo uri)
  {
    MultivaluedMap paramMap = uri.getPathParameters();
    String type = paramMap.getFirst("type");
    String size = paramMap.getFirst("size");
  }
}
```

9.2.4. Getting the query parameters

9.2.5. Getting the matrix parameters

9.3. WORKING WITH THE HTTP HEADERS

9.4. WORKING WITH SECURITY INFORMATION

9.5. WORKING WITH PRECONDITIONS

9.6. WORKING WITH SERVLET CONTEXTS

9.7. WORKING WITH THE APACHE CXF CONTEXT OBJECT

9.8. ADDING CUSTOM CONTEXTS

CHAPTER 10. ANNOTATION INHERITANCE

Abstract

JAX-RS annotations can be inherited by subclasses and classes implementing annotated interfaces. The inheritance mechanism allows for subclasses and implementation classes to override the annotations inherited from its parents.

OVERVIEW

Inheritance is one of the more powerful mechanisms in Java because it allows developers to create generic objects that can then be specialized to meet particular needs. JAX-RS keeps this power by allowing the annotations used in mapping classes to resources to be inherited from super classes.

JAX-RS's annotation inheritance also extends to support for interfaces. Implementation classes inherit the JAX-RS annotations used in the interface they implement.

The JAX-RS inheritance rules do provide a mechanism for overriding inherited annotations. However, it is not possible to completely remove JAX-RS annotations from a construct that inherits them from a super class or interface.

INHERITANCE RULES

Resource classes inherit any JAX-RS annotations from the interface(s) it implements. Resource classes also inherit any JAX-RS annotations from any super classes they extend. Annotations inherited from a super class take precedence over annotations inherited from am interface.

In the code sample shown in Example 10.1, “Annotation inheritance”, the **`Kaijin`** class' **`getMonster()`** method inherits the **`@Path`**, **`@GET`**, and **`@PathParam`** annotations from the **`Kaiju`** interface.

Example 10.1. Annotation inheritance

```
public interface Kaiju
{
  @GET
  @Path("/{id}")
  public Monster getMonster(@PathParam("id") int id);
  ...
}

@Path("/kaijin")
public class Kaijin implements Kaiju
{
  public Monster getMonster(int id)
  {
    ...
  }
  ...
}
```

OVERRIDING INHERITED ANNOTATIONS

Overriding inherited annotations is as easy as providing new annotations. If the subclass, or implementation class, provides any of its own JAX-RS annotations for a method then all of the JAX-RS annotations for that method are ignored.

In the code sample shown in Example 10.2, "Overriding annotation inheritance", the **`Kaijin`** class' **`getMonster()`** method does not inherit any of the annotations from the **`Kaiju`** interface. The implementation class overrides the **`@Produces`** annotation which causes all of the annotations from the interface to be ignored.

Example 10.2. Overriding annotation inheritance

```
public interface Kaiju
{
  @GET
  @Path("/{id}")
  @Produces("text/xml");
  public Monster getMonster(@PathParam("id") int id);
  ...
}

@Path("/kaijin")
public class Kaijin implements Kaiju
{

  @GET
  @Path("/{id}")
  @Produces("application/octect-stream");
  public Monster getMonster(@PathParam("id") int id)
  {
    ...
  }
  ...
}
```

INDEX

Symbols

N

O

P

Q

R

S

T

U

W