# Using JAX-RS 2.0 in

Java EE 7 with RESTEasy

JSR 311 [(http://jcp.org/en/jsr/detail?id=311](http://jcp.org/en/jsr/detail?id=311))) specifies the Java API for RESTful Web services (JAX-RS) for developing **REST** (**Representational State Transfer**) Web services with Java. REST is a protocol independent, loosely coupled, software architecture style for distributed systems. A RESTful Web service exposes a set of resources, which are simply sources of information, identified by **URI**s (**Uniform Resource Identifiers**) in HTTP. RESTful Web services follow these RESTful principles:

##### Every resource has a unique base URI

* For invoking Web service operations, the HTTP protocol methods such as

GET, PUT, POST, and DELETE are used

##### A client sends a request to a service, and the service returns a representation of a resource requested to the client

* Client sessions are not stored on the server, which makes it easier to scale the service with less data to replicate in a clustered environment

JSR 339 ([https://www.jcp.org/en/jsr/detail?id=339](http://www.jcp.org/en/jsr/detail?id=339))) develops the JAX-RS 2.0 version. JAX-RS 2.0 provides several new features, such as a Client API, support for **validation**, **filters** and **interceptors**, and **asynchronous** processing. We will discuss the salient new features in JAX RS 2.0 using the RESTEasy ([http://resteasy.](http://resteasy/) jboss.org/) implementation. This chapter has the following sections:

* Setting up the environment

##### The Client API

* Filters and interceptors
* Asynchronous processing

**[ 317 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

* Cancelling a request
* Session bean EJB resource
* Making an asynchronous call from the client

## Setting up the environment

##### We need to install the following software:

* **WildFly 8.1.0.Final**: Download wildfly-8.1.0.Final.zip from

[http://wildfly.org/downloads/.](http://wildfly.org/downloads/)

* **Eclipse IDE for Java EE Developers**: Download Eclipse Luna from

[https://www.eclipse.org/downloads/packages/release/Luna/SR1](http://www.eclipse.org/downloads/packages/release/Luna/SR1).

* **JBoss Tools (Luna) 4.2.0.Final (or the latest version)**: Install this as a plugin to Eclipse from Eclipse Marketplace (<http://tools.jboss.org/downloads/> installation.html).
* **Apache Maven**: Download version 3.05 or higher from [http://maven.](http://maven/) apache.org/download.cgi.
* **Java 7**: Download Java 7 from <http://www.oracle.com/technetwork/> java/javase/downloads/index.html?ssSourceSiteId=ocomcn.)

Set the environment variables JAVA\_HOME, JBOSS\_HOME, and MAVEN\_HOME. Add

%JAVA\_HOME%/bin, %MAVEN\_HOME%/bin, and %JBOSS\_HOME%/bin to the PATH

##### environment variable.

Create a WildFly 8.1.0 runtime as discussed in *Chapter 1*, *Getting Started with EJB*

*3.x*. Create a MySQL data source with the JNDI name java:jboss/datasources/ MySQLDS as explained in *Chapter 1*, *Getting Started with EJB 3.x*.

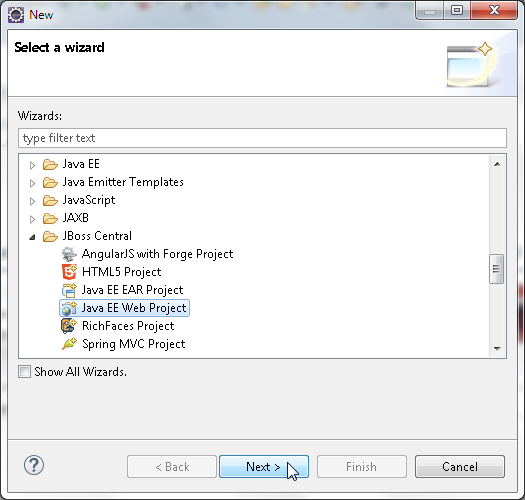
## Creating a Java EE web project

##### First, we need to create a Java EE web project for which you need to select

**File** | **New** | **Other**. In **New**, select **Web** | **Java EE Web Project**, as shown in the following screenshot:

**[ 318 ]**

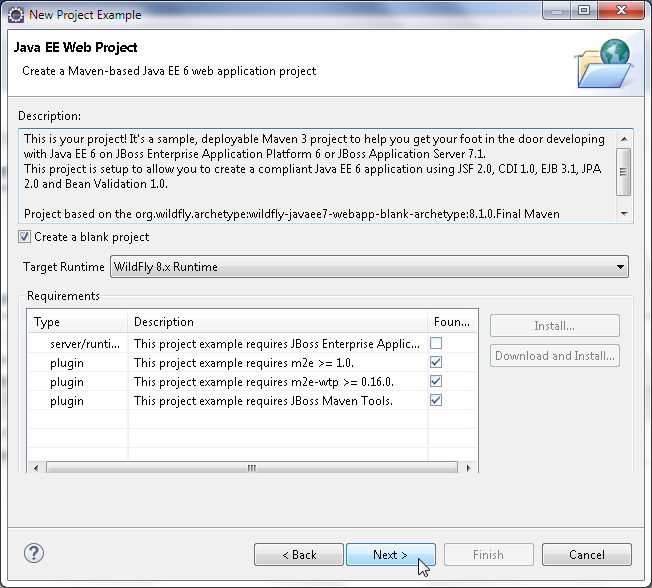
*Chapter 9*



In the **Java EE Web Project** wizard, select **Create a blank project** and select WildFly

8.x Runtime as **Target Runtime**, which is shown as follows. A test gets run to find

##### whether the required plugins are installed. Then, click on **Next**.

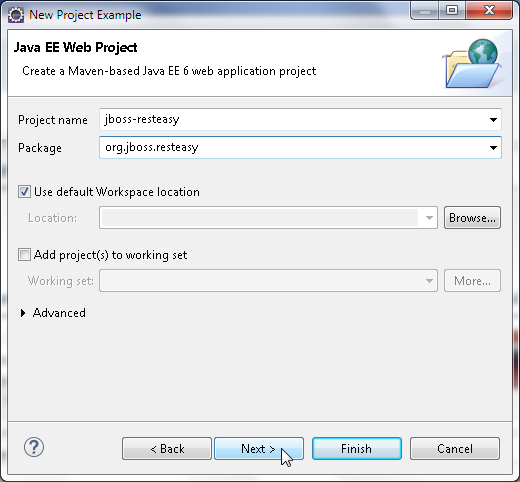


**[ 319 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

Specify **Project name** (jboss-resteasy) and **Package** (org.jboss.resteasy), and

##### click on **Next** as follows:

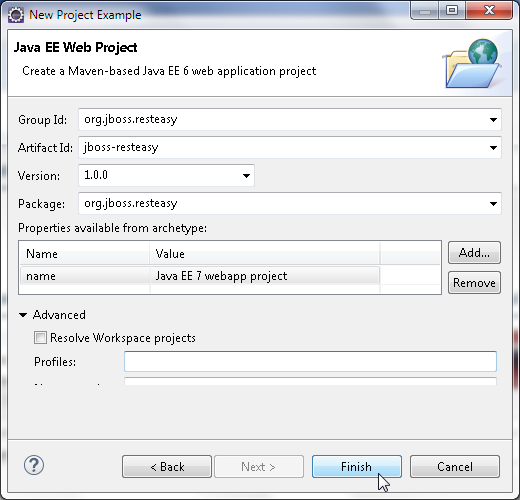


Specify **Group Id** (org.jboss.resteasy), **Artifact Id** (jboss-resteasy), **Version**

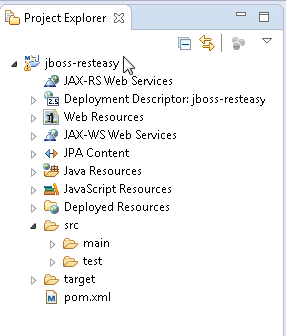
(1.0.0), and **Package** (org.jboss.resteasy), and click on **Next**, as shown here:

**[ 320 ]**

*Chapter 9*



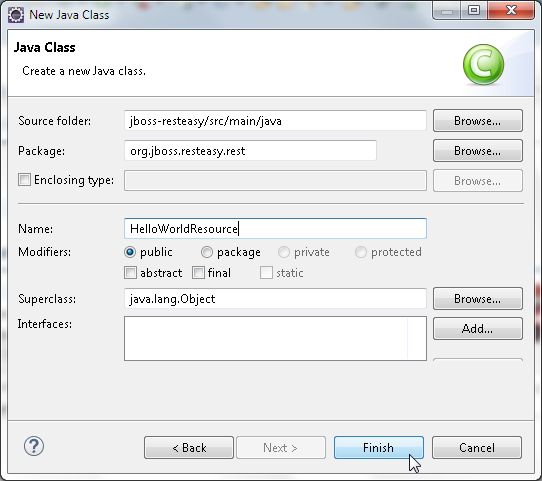
The jboss-resteasy Maven project gets created and gets added to **Project Explorer**, as shown here:



**[ 321 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

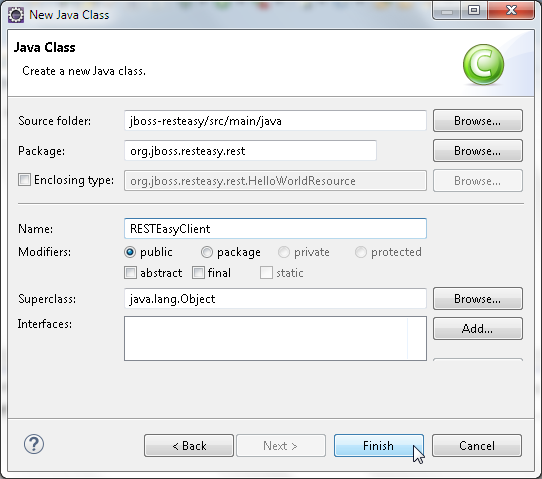
Next, add a JAX-RS resource class (HelloWorldResource), which is just a Java class. Select **File** | **New** | **Other**, and in **New**, select **Java** | **Class** and click on **Next**. Select **Source folder** (jboss-resteasy/src/main/java) and specify **Package** (org.jboss.resteasy.rest) and the class **Name** (HelloWorldResource), and click on **Finish**, as shown here:



**[ 322 ]**

Similarly, add a Java client class (RESTEasyClient) as follows:

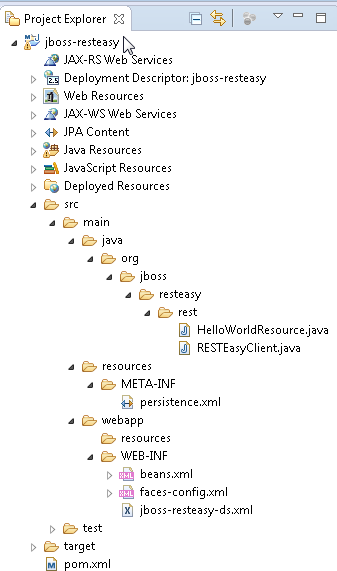
*Chapter 9*



**[ 323 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

##### The directory structure of the jboss-resteasy application is shown in the following screenshot:



Add the JAX-RS- and RESTEasy-related dependencies to pom.xml, as follows:

<dependencies>

<!-- Import the JAX-RS API, we use provided scope as the API is included in JBoss WildFly -->

<dependency>

<groupId>org.jboss.resteasy</groupId>

**[ 324 ]**

*Chapter 9*

<artifactId>jaxrs-api</artifactId>

<version>3.0.10.Final</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.jboss.resteasy</groupId>

<artifactId>resteasy-jackson-provider</artifactId>

<version>3.0.10.Final</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.jboss.resteasy</groupId>

<artifactId>resteasy-jaxrs</artifactId>

<version>3.0.10.Final</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpclient</artifactId>

<version>4.3.6</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpcore</artifactId>

<version>4.3.3</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.jboss.resteasy</groupId>

<artifactId>resteasy-client</artifactId>

<version>3.0.10.Final</version>

<scope>provided</scope>

</dependency>

##### To the build, add the Maven compiler plugin and the Maven WAR plugin. In the Maven WAR plugin configuration, specify the output directory to which the built application is to be deployed: the C:\wildfly-8.1.0.Final\standalone\ deployments directory:

<build>

<!-- Maven will append the version to the finalName (which is the name given to the generated war, and hence the context root) -->

**[ 325 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

<finalName>${project.artifactId}</finalName>

<plugins>

<plugin>

<artifactId>maven-war-plugin</artifactId>

<version>${version.war.plugin}</version>

<configuration>

<outputDirectory>C:\wildfly-8.1.0.Final\standalone\ deployments</outputDirectory>

<failOnMissingWebXml>false</failOnMissingWebXml>

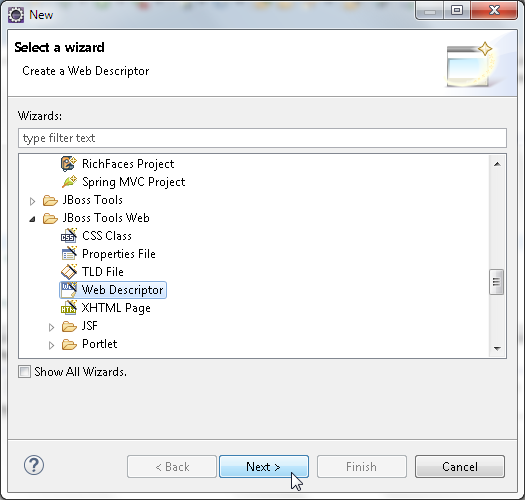
</configuration>

</plugin>

</plugins>

</build>

The complete pom.xml is available in the code download for this chapter. Next, create the web deployment descriptor web.xml. Select **File** | **New** | **Other**. In **New**, select **JBoss Tools Web** | **Web Descriptor** and click on **Next**, as shown here:

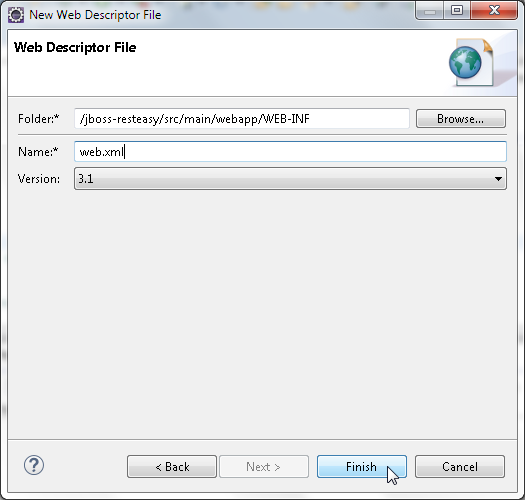


**[ 326 ]**

*Chapter 9*

In **New Web Descriptor File**, select **Folder** as the WEB-INF directory, specify **Name**

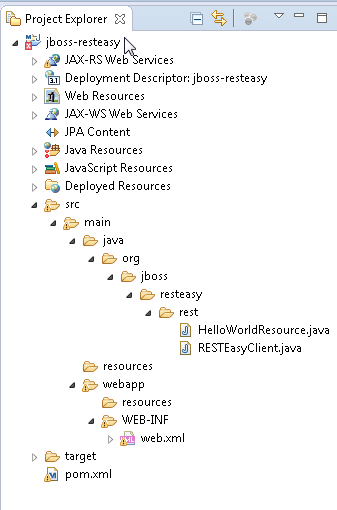
as web.xml, and select **Version** as 3.1, as shown here:



**[ 327 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

##### The web.xml deployment descriptor gets added to WEB-INF, which is shown as follows:



Add the RESTEasy dispatcher servlet to the web.xml file including its URL mapping. Add the context parameter required for RESTEasy to scan for JAX-RS classes. Also, add the context parameter for the RESTEasy servlet mapping the prefix. The web. xml file is listed as follows:

<?xml version="1.0" encoding="UTF-8"?>

<web-app [xmlns="http://xmlns.jcp.org/xml/ns/javaee"](http://xmlns.jcp.org/xml/ns/javaee) [xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) version="3.1" [xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee](http://xmlns.jcp.org/xml/ns/javaee) [http://xmlns.](http://xmlns/) jcp.org/xml/ns/javaee/web-app\_3\_1.xsd">

<context-param>

**[ 328 ]**

*Chapter 9*

<param-name>resteasy.servlet.mapping.prefix</param-name>

<param-value>/rest</param-value>

</context-param>

<context-param>

<param-name>resteasy.scan</param-name>

<param-value>true</param-value>

</context-param>

<listener>

<listener-class>org.jboss.resteasy.plugins.server.servlet.

ResteasyBootstrap</listener-class>

</listener>

<servlet>

<servlet-name>Resteasy</servlet-name>

<servlet-class>org.jboss.resteasy.plugins.server.servlet.

HttpServletDispatcher</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>Resteasy</servlet-name>

<url-pattern>/rest/\*</url-pattern>

</servlet-mapping>

</web-app>

## The Client API

##### The Client API is a high-level API used to access web resources and integrate with

JAX-RS providers and is included in the javax.ws.rs.client package.

A picture containing text, clipart

Description automatically generatedPreviously, different implementations provided the Client API,

but in JAX-RS 2.0, the Client API is provided as a core API.

### Creating a client instance

A Client instance is required to build and run client requests to access or consume web resources. In the RESTEasy client class RESTEasyClient.java, create a Client instance from ClientBuilder using the newClient() method as follows:

Client client = ClientBuilder.newClient();

Providers, filters, and features can be configured with the Client object using the register() method. For example, the org.jboss.resteasy.plugins.providers. JaxrsFormProvider.class provider class is registered as follows:

client.register(org.jboss.resteasy.plugins.providers. JaxrsFormProvider.class);

**[ 329 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

### Accessing a resource

##### The Client API is used to access a web resource as follows:

1. Create a WebTarget object from the resource URI using the overloaded target() method of the Client object. The path appended to the URI is to enable the REST service to handle multiple inputs:

WebTarget target = client.target("http://localhost:8080/jboss- resteasy/rest/helloworld");

##### Add one or more path elements to the WebTarget object if required using the

path() method, which returns a WebTarget object:

WebTarget target = target.path("text");

##### Create a request from the WebTarget object using the overloaded request() method, in which you need to define the accepted response media types. Invoke the HTTP GET method for the request using the overloaded get() method to obtain an invocation response as a Response object:

Response response=target.request("text/plain").get();

##### Obtain the message entity input stream as a String object:

String value = response.readEntity(String.class);

##### The fluent API can be used to build and submit the client request and obtain

a response by linking the method invocations:

String response = client.target("http://localhost:8080/jboss- resteasy/rest/helloworld").path("text").request("text/plain"). get(String.class);

The RESTEasyClient.java class is listed as follows:

package org.jboss.resteasy.rest;

import javax.ws.rs.client.Client;

import javax.ws.rs.client.ClientBuilder;

import javax.ws.rs.client.WebTarget; import javax.ws.rs.core.\*;

import org.jboss.resteasy.client.jaxrs.ResteasyClient; public class RESTEasyClient {

public static void main(String[] args) {

Client client = ClientBuilder.newClient();

**[ 330 ]**

*Chapter 9*

String response = client.target("http://localhost:8080/jboss- resteasy/rest/helloworld").path("text").request("text/plain"). get(String.class);

System.out.println(response);

}

}

Create a resource class hosted at the /helloworld URI path to test the Client API. Add a resource method at the relative URI path /text to return a Hello message from a name. The resource class HelloWorldResource is listed as follows:

package org.jboss.resteasy.rest;

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

@Path("/helloworld")

public class HelloWorldResource {

@GET

@Produces("text/plain") @Path("/text")

public String getClichedMessage() {

return "Hello John Smith";

}

}

##### To test the resource class and the client, compile, package, and deploy the

jboss-resteasy application to WildFly. We will add the output directory as the WildFly deployments directory to the configuration for the Maven WAR plugin, as shown here:

<plugin>

<artifactId>maven-war-plugin</artifactId>

<version>${version.war.plugin}</version>

<configuration>

<outputDirectory>C:\wildfly-8.1.0.Final\standalone\deployments</ outputDirectory>

<failOnMissingWebXml>false</failOnMissingWebXml>

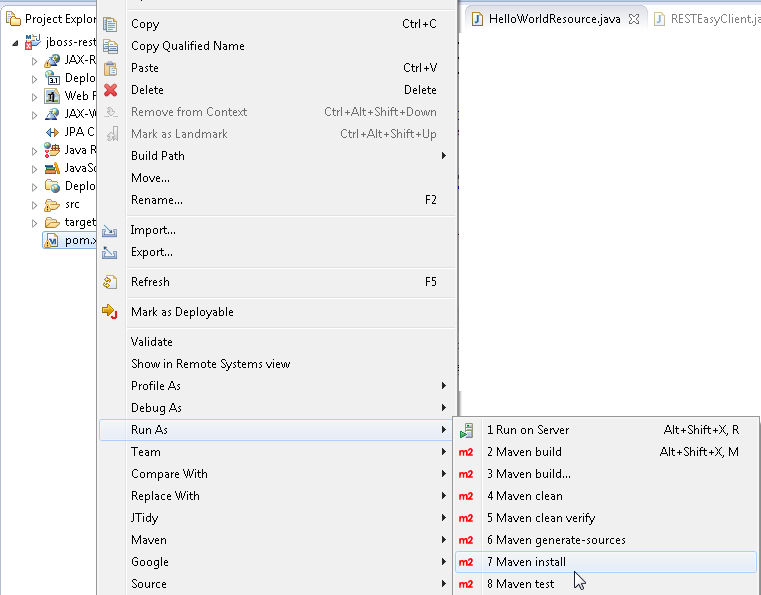
</configuration>

</plugin>

**[ 331 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

Right-click on pom.xml and select **Run As** | **Maven install**, which is shown as follows:



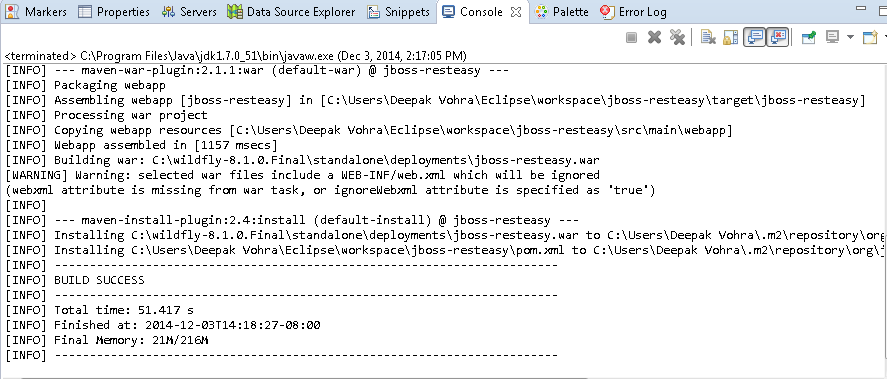
##### The jboss-resteasy application gets compiled, built, and outputted to the WildFly

* 1. deployments directory is indicated by the **BUILD SUCCESS** message in the

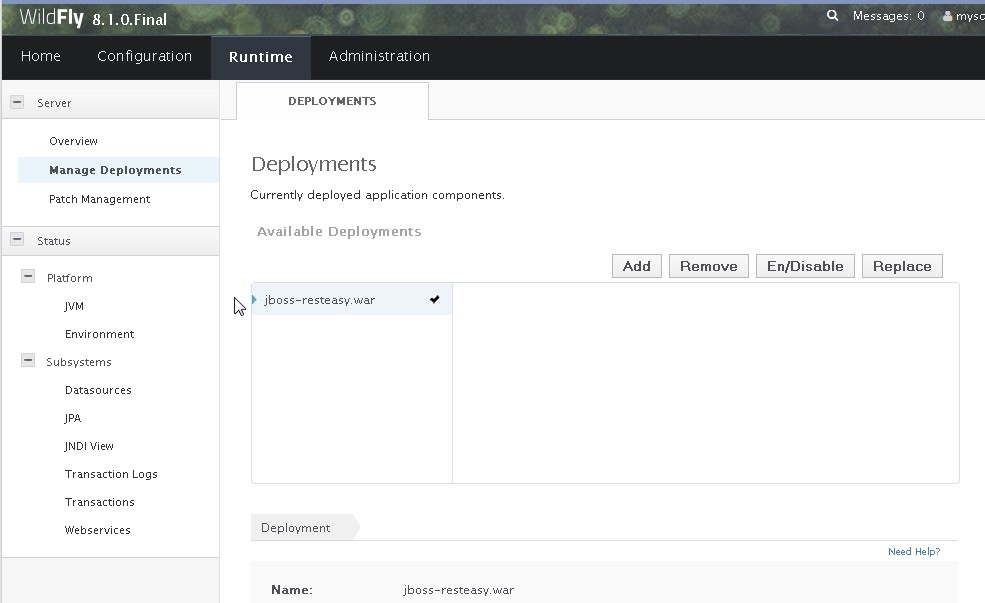
##### **Console**, which is shown as follows:

**[ 332 ]**

*Chapter 9*



Now, log in to the WildFly 8.1 **Administration Console** and click on **Manage Deployments**. The jboss-resteasy.war application should be listed as deployed, which is shown in the following screenshot:

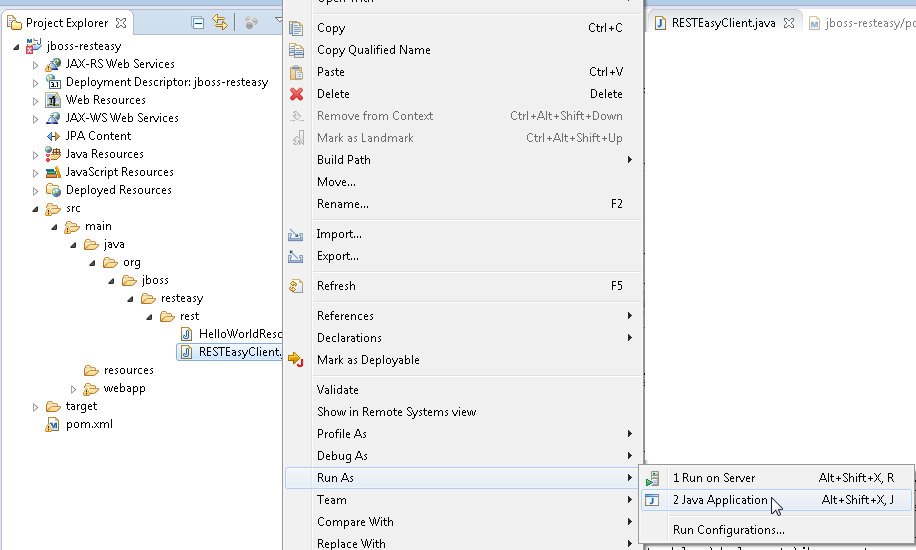


**[ 333 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

To test the client, right-click on the RESTEasyClient.java class in **Project Explorer**

and select **Run As** | **Java Application**, as shown here:



##### The client application runs to invoke the resource class and produces the output, as shown in the following **Console**:



**[ 334 ]**

### Setting a query parameter

*Chapter 9*

##### To invoke a resource method with parameters, the @QueryParam annotation can be used to bind request parameters to resource method parameters. In a variation of the resource class used in the previous subsection, add a String parameter to the resource method. Annotate the parameter declaration with @QueryParam and set its default value as DefaultValue:

@GET

@Produces("text/plain") @Path("/text")

public String getClichedMessage(@QueryParam("name") @ DefaultValue("John Smith") String name) {

return "Hello " +name;

}

##### In the client class, the query parameter can be sent in the request using the

queryParam() method as follows:

String response = client.target("http://localhost:8080/jboss-resteasy/ rest/helloworld").path("text").queryParam("name", "John Smith"). request("text/plain").get(String.class);

##### Alternatively, the query parameter can be included in the request URI, which is shown as follows:

String response = client.target("http://localhost:8080/jboss-resteasy/ rest/helloworld/text?name=John Smith").request("text/plain"). get(String.class);

### Setting a template parameter

##### The resource URI can also be built using template parameters. In a variation of HelloWorldResource, specify a template parameter {name} in the @Path annotating the resource method. Bind the template parameter to the resource method parameter using the @PathParam annotation:

@GET

@Produces("text/plain") @Path("/text/{name}")

public String getClichedMessage(@PathParam("name") String name) { return "Hello " +name;

}

**[ 335 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

##### In the RESTEasyClient class, include the value for the {name} template parameter in the resource URI as follows:

String response = client.target("http://localhost:8080/jboss-resteasy/ rest/helloworld/text/John Smith").request("text/plain").get(String. class);

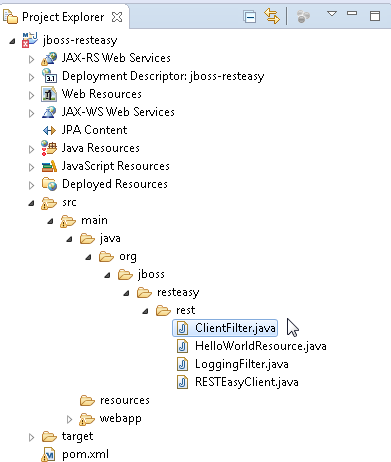
Redeploy the jboss-resteasy application and rerun the client to produce the same output.

## Filters and interceptors

##### Filters provide extended functionality such as logging and authentication. Interceptors provide extended functions such as entity compression. In this section, we will discuss the support for filters at specific extension points in JAX-RS 2.0 implementation. The two types of filters are provided in JAX-RS 2.0: **client filters** and **container filters**.

The client filters are on the client side and the container filters are on the container side. Interfaces corresponding to the client filters are included in the Client API and are javax.ws.rs.client.ClientRequestFilter and javax.ws.rs.client.

ClientResponseFilter. Interfaces for the container filters, which are included in the Server API, are javax.ws.rs.container.ContainerRequestFilter and javax. ws.rs.container.ContainerResponseFilter. To be discovered by the JAX-RS runtime, filters implementing the interfaces must be annotated with the @Provider annotation. Create Java classes LoggingFilter (for the container filter example), and ClientFilter (for the client filter example), as shown in **Project Explorer**.



**[ 336 ]**

*Chapter 9*

##### Before we discuss the client and container filters, we need to discuss the junctions at which the filters intercept communication between the client and the server:

* + 1. The ClientRequestFilter intercepts communication before the client HTTP request is sent over to the server.
    2. The ContainerRequestFilter intercepts after the client is sent over to the

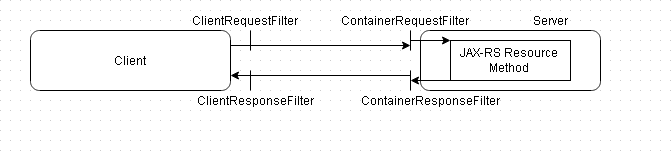
##### server but before the JAX-RS resource method is invoked.

* + 1. The ContainerResponseFilter intercepts after the JAX-RS resource method

is invoked but before the response is sent back to the client.

* + 1. The ClientResponseFilter is invoked after the server HTTP response is sent over to the client but before the response is unmarshalled.

The junctions of request/response interception are illustrated in the following diagram:



### Creating a client filter

First, we will discuss the client filters with an example. The ClientRequestFilter

##### is run in the invocation pipeline before the HTTP request is delivered to the network.

The ClientRequestFilter should be annotated by @Provider, which is the

marker that is discovered by the JAX-RS runtime during the scanning phase. The ClientResponseFilter is run after the response is received from the server and before the control is returned to the application. Make the ClientFilter class implement the ClientRequestFilter and ClientResponseFilter interfaces. Add implementation for the filter(ClientRequestContext arg0) and filter(ClientRequestContext arg0, ClientResponseContext arg1) methods. In the ClientRequestFilter implementation method filter(ClientRequestContext arg0), output some headers using the getHeaderString(String) method of ClientRequestContext.

For example, the Accept-Charset and Accept-Encoding headers give out the following output:

System.out.println("Accept-Charset: " + arg0.getHeaderString("Accept- Charset"));

System.out.println("Accept-Encoding: " + arg0.getHeaderString("Accept- Encoding"));

**[ 337 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

##### Set a new resource URI using the setUri(URI) method as follows:

arg0.setUri(new URI("http://localhost:8080/jboss-resteasy/rest/ helloworld/text/Smith,John"));

The ClientFilter class is listed as follows:

package org.jboss.resteasy.rest;

import java.io.IOException; import java.net.URI;

import java.net.URISyntaxException;

import javax.ws.rs.client.ClientRequestContext; import javax.ws.rs.client.ClientRequestFilter; import javax.ws.rs.client.ClientResponseContext; import javax.ws.rs.client.ClientResponseFilter;

import javax.ws.rs.core.Response; import javax.ws.rs.ext.Provider;

@Provider

public class ClientFilter implements ClientRequestFilter, ClientResponseFilter {

@Override

public void filter(ClientRequestContext arg0, ClientResponseContext arg1)

throws IOException {

}

@Override

public void filter(ClientRequestContext arg0) throws IOException {

System.out.println("Entity Class: " + arg0.getEntityClass()); System.out.println("Accept: " + arg0.getHeaderString("Accept")); System.out.println("Accept-Charset: "

+ arg0.getHeaderString("Accept-Charset")); System.out.println("Accept-Encoding: "

+ arg0.getHeaderString("Accept-Encoding")); System.out.println("Accept-Language: "

+ arg0.getHeaderString("Accept-Language")); System.out.println("Accept-Ranges: "

**[ 338 ]**

*Chapter 9*

+ arg0.getHeaderString("Accept-Ranges"));

System.out.println("Allow: " + arg0.getHeaderString("Allow")); System.out.println("Authorization: "

+ arg0.getHeaderString("Authorization")); System.out.println("Cache-Control: "

+ arg0.getHeaderString("Cache-Control")); System.out.println("Content-Encoding: "

+ arg0.getHeaderString("Content-Encoding")); System.out.println("Content-Location: "

+ arg0.getHeaderString("Content-Location")); System.out.println("Accept-Encoding: "

+ arg0.getHeaderString("Accept-Encoding")); System.out.println("Content-Type: "

+ arg0.getHeaderString("Content-Type")); System.out.println("Host: " + arg0.getHeaderString("Host")); System.out.println("Pragma: " + arg0.getHeaderString("Pragma")); System.out.println("Server: " + arg0.getHeaderString("Server")); System.out.println("User-Agent: " + arg0.getHeaderString("User-

Agent"));

try {

arg0.setUri(new URI(

"http://localhost:8080/jboss-resteasy/rest/helloworld/text/ Smith,John"));

} catch (URISyntaxException e) {

// TODO Auto-generated catch block e.printStackTrace();

}

//arg0.abortWith(Response.notAcceptable(null).build());

}

}

##### For a client filter issue that could occur, refer to the section *Fixing a Common Issue* at the end of this chapter. In the client class, RESTEasyClient registers the client filter with the client:

client.register(ClientFilter.class);

We will use the following root resource class HelloWorldResource to test the

##### client filter:

package org.jboss.resteasy.rest; import javax.ws.rs.GET;

import javax.ws.rs.PathParam;

**[ 339 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

import javax.ws.rs.Produces; import javax.ws.rs.Path;

@Path("/helloworld")

public class HelloWorldResource {

@GET

@Produces("text/plain") @Path("/text/{name}")

public String getClichedMessage(@PathParam("name") String name) { return "Hello " +name;

}

}

##### The client class RESTEasyClient to test the client filter with is listed as follows:

package org.jboss.resteasy.rest; import javax.ws.rs.client.Client;

import javax.ws.rs.client.ClientBuilder; import javax.ws.rs.client.WebTarget; import javax.ws.rs.core.\*;

public class RESTEasyClient {

public static void main(String[] args) { Client client = ClientBuilder.newClient(); client.register(ClientFilter.class);

String response = client.target("http://localhost:8080/jboss- resteasy/rest/helloworld/text/John Smith").request("text/plain"). get(String.class);

System.out.println("Text response "+ response);

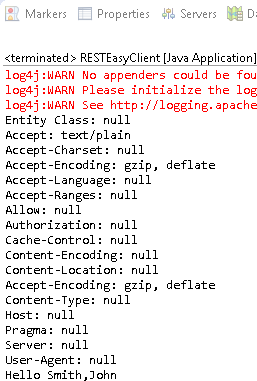
}

}

**[ 340 ]**

*Chapter 9*

Redeploy the jboss-resteasy application. To redeploy, right-click on pom.xml in **Project Explorer** and select **Run As** | **Maven clean**, and subsequently, right-click on pom.xml and select **Run As** | **Maven install**. Run the RESTEasyClient.java class to generate the following output in the **Console** screen shown as follows:



##### As we modified the resource URI in the client filter, the response message is not for John Smith as specified in the client class, but for **Smith, John**.

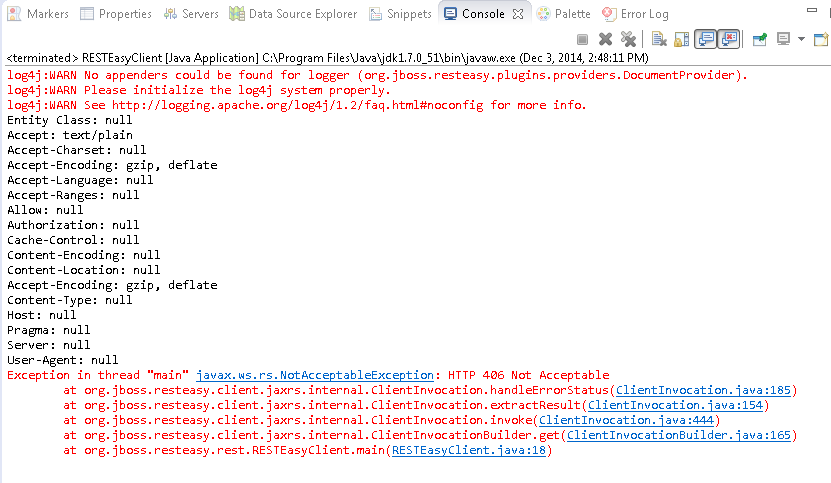
The filter chain processing may be aborted and response is returned to the client with the abortWith(Response response) method. The client response filters get applied before the client gets the response. As an example, break the filter chain and return a notAcceptable(null) response:

arg0.abortWith(Response.notAcceptable(null).build());

**[ 341 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

Keep RESTEasyClient and HelloWorldResource the same and redeploy the j boss-restaesy application. Rerun the RESTEasyClient class to generate the following output, which includes a NotAcceptableException shown as follows:



## Creating a container filter

##### Container filters are Server API filters. A ContainerRequestFilter filter is run after receiving a request from the client. A ContainerResponseFilter is run in the response pipeline before the HTTP response is delivered to the client. Next, we will create a container filter for logging/outputting some information about the request. Extension points before and after the match are provided in the ContainerRequestFilter interface. The pre-match filter is run before the request

has been matched with a resource method, and the post-match filter is applied after the resource method matching; the default is post-match. We will use pre-match with the @PreMatching annotation. Annotate the filter class with @Provider for the filter to be discovered by the JAX-RS runtime during the scanning phase.

**[ 342 ]**

*Chapter 9*

Make the example container filter LoggingFilter, implement the

ContainerRequestFilter, ContainerResponseFilter interfaces, and provide

implementation for the filter(ContainerRequestContext requestContext) and

filter(ContainerRequestContext requestContext, ContainerResponseContext responseContext) methods. In the ContainerRequestFilter implementation method filter(ContainerRequestContext requestContext), output the request method with the getMethod() method, request URI with the getUriInfo(). getAbsolutePath(), media type with getMediaType(), and acceptable media

types with getAcceptableMediaTypes(). Include a no-argument constructor in the LoggingFilter so that the filter may be instantiated. Register the LoggingFilter with the client configuration using the register method. Comment out the registration of the ClientFilter as we will apply only the LoggingFilter in the RESTEasyClient class:

Client client = ClientBuilder.newClient();

//client.register(ClientFilter.class); client.register(LoggingFilter.class);

##### By default, container filters are bound to all the resources the client request is sent to, but a resource-specific container filter can be applied using the @NameBinding annotation:

The LoggingFilter.java class is listed as follows:

package org.jboss.resteasy.rest;

import java.io.IOException; import java.util.Iterator; import java.util.List;

import javax.ws.rs.container.ContainerRequestContext; import javax.ws.rs.container.ContainerRequestFilter; import javax.ws.rs.container.ContainerResponseContext; import javax.ws.rs.container.ContainerResponseFilter; import javax.ws.rs.container.PreMatching;

import javax.ws.rs.core.MediaType; import javax.ws.rs.ext.Provider;

@Provider @PreMatching

public class LoggingFilter implements ContainerRequestFilter, ContainerResponseFilter {

**[ 343 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

public LoggingFilter() {

}

@Override

public void filter(ContainerRequestContext requestContext) throws IOException {

System.out.println("Request Method: " + requestContext. getMethod());

System.out.println("Request URI: "+ requestContext.getUriInfo(). getAbsolutePath());

System.out.println("Media Type : "+ requestContext. getMediaType());

List<MediaType> mediaTypes = requestContext. getAcceptableMediaTypes();

Iterator<MediaType> iter = mediaTypes.iterator(); System.out.println("Acceptable Media Types: "); while (iter.hasNext()) {

MediaType mediaType = iter.next(); System.out.println(mediaType.getType() + ", ");

}

}

@Override

public void filter(ContainerRequestContext requestContext, ContainerResponseContext responseContext) throws IOException {

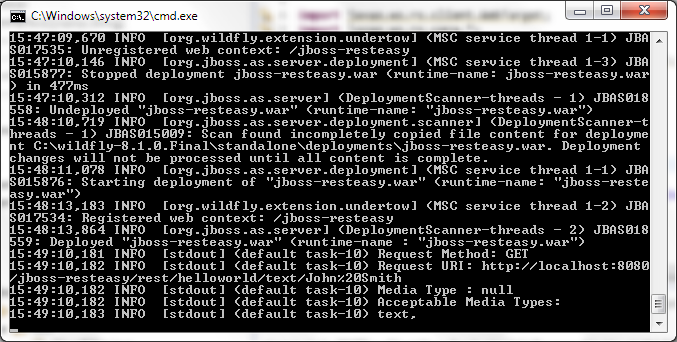
}

}

Keep RESTEasyClient and HelloWorldResource the same as the ClientFilter example and redeploy the jboss-restaesy application. Run the RESTEasyClient application to generate the output shown from the container filter, which is shown here:

**[ 344 ]**

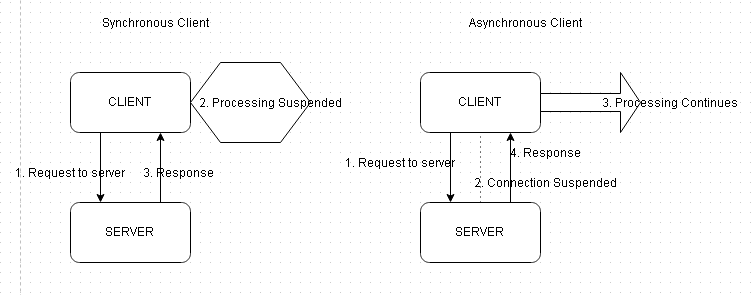
*Chapter 9*



## Asynchronous processing

##### JAX-RS 2.0 has added support for asynchronous processing in both the client API and the server API. By default, when a client sends a request to the server, it suspends all other processing till the response is received. With asynchronous

processing, the client suspends connection with the server and continues to process while a server response is being generated and sent back to the client. When the response is delivered to the client, the client re-establishes a connection with the server and accepts the response. The client-server model in synchronous and asynchronous request/response is illustrated in the following diagram:

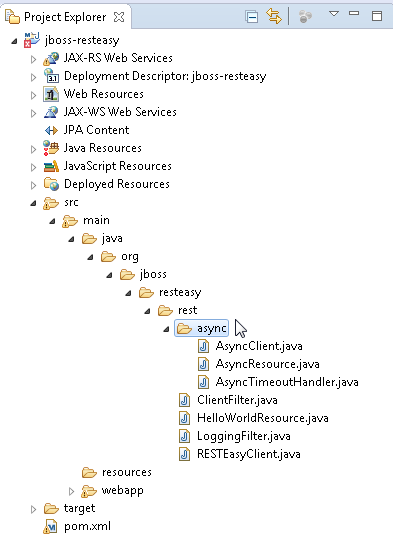


**[ 345 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

##### Similarly, by default a server thread blocks all other incoming client requests while waiting for an external process to complete one client request. With asynchronous processing, the server suspends connection with the client so that it may accept other client requests. When a response is available for a client request, the

server re-establishes a connection with the client and sends the request. In this section, we will discuss asynchronous processing with an example. Create Java classes AsyncResource (for a root resource class), AsyncClient(for a client), and AsyncTimeoutHandler (for a timeout handler). The directory structure of the async classes is shown in **Project Explorer** as follows:



**[ 346 ]**

*Chapter 9*

The server API has added the javax.ws.rs.container.AsyncResponse interface to represent an asynchronous response for server-side processing of an asynchronous response. The javax.ws.rs.container.Suspended interface is provided to inject

a suspended AsyncResponse instance into a resource method parameter. The AsyncResponse instance is bound to an active client request and can be used to provide a response asynchronously when a response is available. When a response is to be sent to the client, the AsyncResponse instance resumes the suspended request.

### Suspended response

##### A resource or subresource method that injects a suspended AsyncResponse using the @Suspended annotation must declare the return type as void. If the injected AsyncResponse instance does not cancel or resume a suspended asynchronous response, the response is suspended indefinitely. In the AsyncResource root resource class, add a resource method (called timeout for example), which has a suspended AsyncResponse instance injected into a resource method parameter using the @Suspended annotation, as shown in the following listing. A template parameter

{timeout} is included in the path URI for the resource method:

package org.jboss.resteasy.rest.async;

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

import javax.ws.rs.container.AsyncResponse; import javax.ws.rs.container.Suspended;

@Path("/helloworld")

public class AsyncResource {

@GET

@Path("/timeout/{timeout}") Produces("text/plain")

public void timeout(@PathParam("timeout") String timeoutStr,@ Suspended AsyncResponse ar) {}

}

**[ 347 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

##### In the AsyncClient, class includes a value of 60 for the {timeout} template parameter in the request URI, as shown in the following listing:

package org.jboss.resteasy.rest.async;

import javax.ws.rs.client.Client;

import javax.ws.rs.client.ClientBuilder; import javax.ws.rs.client.WebTarget; import javax.ws.rs.core.\*;

public class AsyncClient {

public static void main(String[] args) {

Client client = ClientBuilder.newClient();

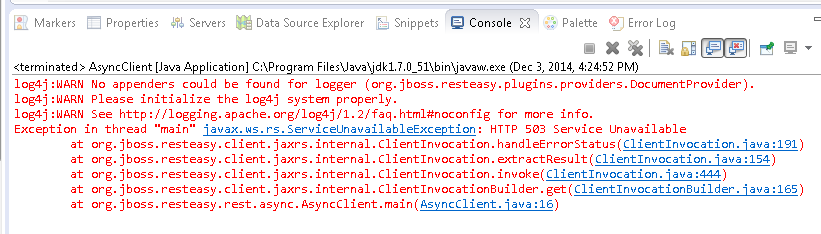
WebTarget target = client.target("http://localhost:8080/jboss- resteasy/rest/helloworld/timeout/60");

String response = target.request("text/plain").get(String.class); System.out.println("Text response: " + response);

}

}

##### Run the pom.xml file to deploy the jboss-resteasy application. When the AsyncClient application is run, the server does not return a response as the asynchronous response is suspended with the following exception being returned:



**[ 348 ]**

### Resuming request processing

*Chapter 9*

The suspended AsyncResponse may choose to resume the request processing, usually when a response is available, using the resume(Object) method. Build the response using the ResponseBuilder object, which may be obtained for the Response static method ok(Object). Set the media type for the response using the ResponseBuilder method type(MediaType) and create a Response object using the build() method. Resume the suspended request processing using the resume(Object) method to send the Response object. The AsyncResource root resource class is listed as follows:

package org.jboss.resteasy.rest.async;

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

import javax.ws.rs.container.AsyncResponse; import javax.ws.rs.container.Suspended; import javax.ws.rs.core.MediaType;

import javax.ws.rs.core.Response;

@Path("/helloworld")

public class AsyncResource {

@GET

@Path("/timeout/{timeout}") @Produces("text/plain")

public void timeout(@PathParam("timeout") String timeoutStr, @Suspended AsyncResponse ar) {

try {

Response hello = Response.ok("Hello John Smith").type(MediaType.

TEXT\_PLAIN).build(); ar.resume(hello);

} catch (Exception e) { System.out.println(e.getMessage());

}

}

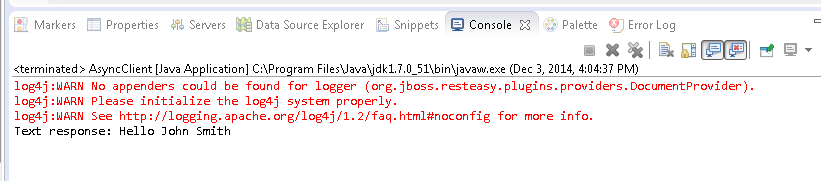
}

**[ 349 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

The client class is the same as listed in the *Suspended response* section of this chapter. To compile and package the jboss-resteasy application, right-click on pom.

xml and select **Run As** | **Maven Install**. Start the WildFly 8.1 server to deploy the application, and after the application has deployed, run the client class AsyncClient. Right-click on AsyncClient.java in **Package Explorer** and select **Run As** | **Java Application**. The client runs to produce the output, which is shown as follows:



The Response object to be sent may be a String literal. If a String literal is used in the resume(Object) as shown here, a **Hello after a timeout** message gets generated:

ar.resume("Hello after a timeout");

### Resuming a request with a suspend timeout handler

The AsyncResponse instance may choose to update the suspended set data to set a new suspend time-out. A new suspend time-out is set as follows using the setTimeout(long time, TimeUnit unit) method:

ar.setTimeout(timeout, TimeUnit.SECONDS);

##### The ar variable is the AsyncResponse object. The new suspended timeout value overrides the previous timeout value. At the first invocation of setTimeout, the suspend timeout has gone from being suspended indefinitely to being suspended for the specified timeout value. The javax.ws.rs.container.TimeoutHandler interface is used to provide custom resolution of timeout events. The default resolution of a timeout event is for the JAX-RS 2.0 runtime to generate a Service

unavailable exception. Set a suspend timeout handler using the setTimeoutHandle r(TimeoutHandler handler) method:

ar.setTimeoutHandler(new AsyncTimeoutHandler("Timeouted after " + timeout + " seconds"));

**[ 350 ]**

*Chapter 9*

##### The AsyncResource class to set a suspend timeout handler is listed as follows:

package org.jboss.resteasy.rest.async; import java.util.concurrent.TimeUnit;

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

import javax.ws.rs.container.AsyncResponse; import javax.ws.rs.container.Suspended; import javax.ws.rs.core.MediaType;

import javax.ws.rs.core.Response;

@Path("/helloworld")

public class AsyncResource {

@GET

@Path("/timeout/{timeout}") @Produces("text/plain")

public void timeout(@PathParam("timeout") String timeoutStr, @Suspended AsyncResponse ar) {

try {

long timeout = Long.parseLong(timeoutStr); System.out.println("timeout - enter with timeout=" + timeoutStr

+ "s");

ar.setTimeoutHandler(new AsyncTimeoutHandler("Timeouted after "

+ timeout + " seconds")); ar.setTimeout(timeout, TimeUnit.SECONDS);

} catch (Exception e) { System.out.println(e.getMessage());

}

}

}

Make the AsyncTimeoutHandler timeout handler class implement the TimeoutHandler interface. In the AsyncTimeoutHandler, implement the handleTimeout(AsyncResponse asyncResponse) method in which the suspended timeout can be handled with one of the following methods:

* The asynchronous response can be resumed using the resume(Object)

##### method

**[ 351 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

* The response can be resumed using the resume(Throwable) method to throw an exception

##### The response can be cancelled using the cancel() method

* The suspend timeout can be extended using another invocation of the

setTimeout(long time, TimeUnit unit) method

In the AsyncTimeoutHandler class, resume the asynchronous response using the resume(Object) method to return a response to the client. The AsyncTimeoutHandler class is listed as follows:

package org.jboss.resteasy.rest.async; import java.util.concurrent.TimeUnit; import javax.ws.rs.container.AsyncResponse;

import javax.ws.rs.container.TimeoutHandler;

public class AsyncTimeoutHandler implements TimeoutHandler { private String \_message;

boolean keepSuspended = false;

//boolean cancel = true; boolean cancel = false; int retryPeriod = 10;

AsyncTimeoutHandler(String message) {

\_message = message;

}

@Override

public void handleTimeout(AsyncResponse ar) { System.out.println("handleTimeout - enter"); if (keepSuspended) {

ar.setTimeout(10, TimeUnit.SECONDS);

} else if (cancel) { ar.cancel(retryPeriod);

} else { ar.resume(\_message);

}

/\*Response hello = Response.ok("Hello after a timeout"). type(MediaType.TEXT\_PLAIN).build();

ar.resume(hello);\*/

}

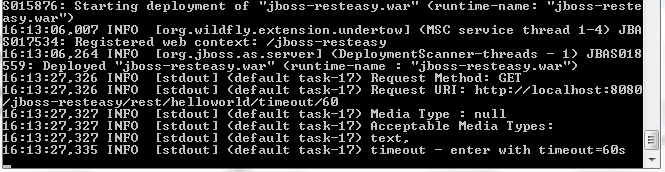
}

**[ 352 ]**

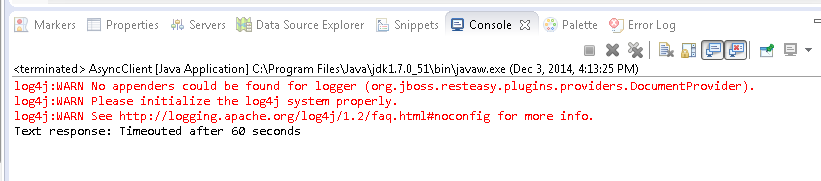
*Chapter 9*

##### Redeploy the application with Maven and rerun the AsyncClient class. The new suspended timeout gets applied and the response gets suspended for 60 seconds

as indicated by the message timeout- enter with timeout=60s, which is shown as follows:



##### When the request processing is resumed, the following response, as shown in the following screenshot, is sent to the client and output from the client class AsyncClient.java:



In the previous example, we resumed the request in the timeout handler. A request can be resumed in a resource method in which the new suspended timeout and the timeout handler are set before the new suspend timeout has run as shown in the resource method in the following listing:

@GET

@Path("/timeout/{timeout}") @Produces("text/plain")

public void timeout(@PathParam("timeout") String timeoutStr, @ Suspended AsyncResponse ar) {

try {

long timeout = Long.parseLong(timeoutStr); System.out.println("timeout - enter with timeout=" + timeoutStr +

"s");

ar.setTimeoutHandler(new AsyncTimeoutHandler("Timeouted after " + timeout + " seconds"));

ar.setTimeout(timeout, TimeUnit.SECONDS);

**[ 353 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

Response hello = Response.ok("Hello before the suspend timeout of

60 seconds has run").type(MediaType.TEXT\_PLAIN).build(); ar.resume(hello);

} catch (Exception e) { System.out.println(e.getMessage());

}

}

For a new suspend timeout to be applied, the request must be resumed in the timeout handler. If a new suspended timeout and a timeout handler are set and the suspended timeout handler does not take any action, the default resolution is for the request processing to be resumed with a ServiceUnAvailableException exception. To resume the request to send a response, the request has to be resumed explicitly using the resume(Object) method.

### Cancelling a request

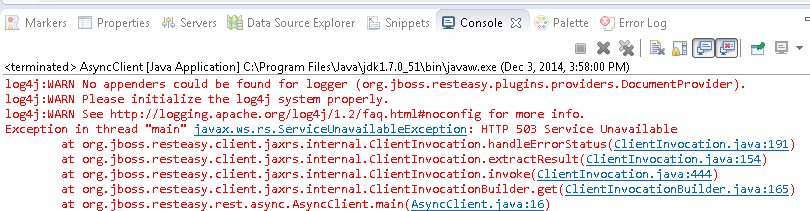
##### The AsyncResponse instance can cancel the response in the suspended timeout handler or the resource method using the overloaded cancel() method:

boolean cancel = true; int retryPeriod = 10; if (cancel) {

System.out.println("Cancel the suspeneded request processing"); ar.cancel(retryPeriod);

}

##### The client gets the following exception when a response is cancelled, which is shown as follows:



**[ 354 ]**

## Session bean EJB resource

*Chapter 9*

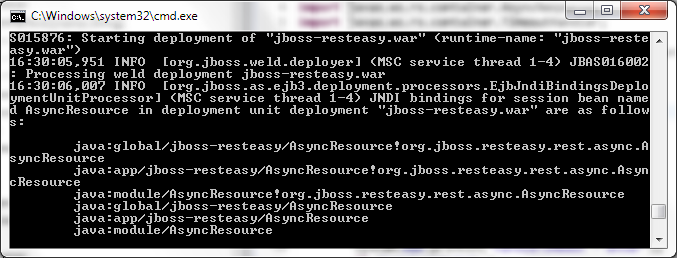
##### JAX-RS 2.0 supports stateless and singleton session beans as root resource classes. In this section, we will run the AsyncResource root resource class as a stateless session bean. We added an EJB-related dependency to pom.xml.

Annotate the AsyncResource with the Stateless annotation. The @Path annotation must also be applied to the class:

@Path("/helloworld") @Stateless

public class AsyncResource {}

When the application is run, the root resource class gets added to the JNDI just as any other session bean would. The JNDI binding for AsycnResource is shown as follows:



## Making an asynchronous call from

**the client**

##### We have as yet discussed only the asynchronous support in the Server API. The asynchronous request processing has also been made available in the client API. A client request can be sent asynchronously using the async() method.

**[ 355 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

To invoke a resource asynchronously implies that the call returns immediately. Optionally, a callback can be registered using the InvocationCallback interface. Implement the completed(RESPONSE response) and failed(Throwable throwable) methods. The completed(RESPONSE response) method is called when the invocation completes successfully and the failed(Throwable throwable) method is called when the invocation fails. A client request is made asynchronously in the AsyncClient.java client as follows:

WebTarget target = client.target("http://localhost:8080/jboss- resteasy/rest/helloworld/timeout/60");

AsyncInvoker asyncInvoker = target.request("text/plain").async(); asyncInvoker.get(new InvocationCallback<String>() {

@Override

public void completed(String response) { System.out.println("Invocation completed and response available");

}

@Override

public void failed(Throwable arg0) {}

});

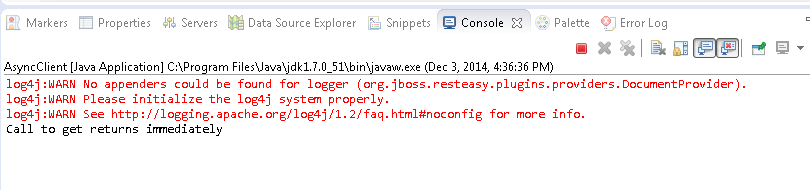
System.out.println("Call to get returns immediately");

##### Set a suspended timeout and timeout handler in the resource method and resume the request in the timeout handler as follows:

Response hello = Response.ok("Hello after a timeout").type(MediaType. TEXT\_PLAIN).build();

ar.resume(hello);

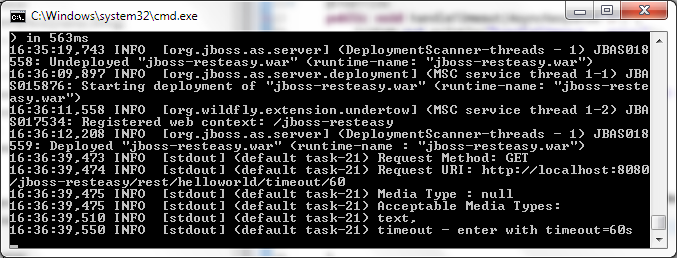
##### When the application is run, the call returns immediately with the message Call to get returns immediately (the message can get output and the processing continue without the message being noticed).



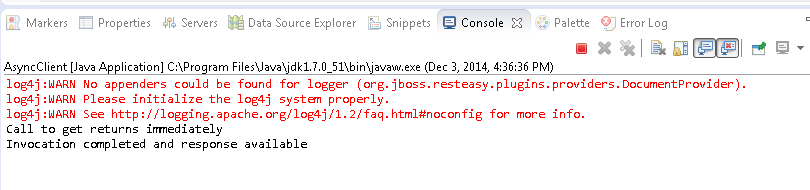
**[ 356 ]**

*Chapter 9*

And the suspend timeout starts with the message. This is shown in the following screenshot:



After the suspended timeout has run, the response is returned as shown in the following screenshot. The message output when the call returns immediately is also shown here:



## Fixing a common issue

##### A RESTEasy application may generate the following error:

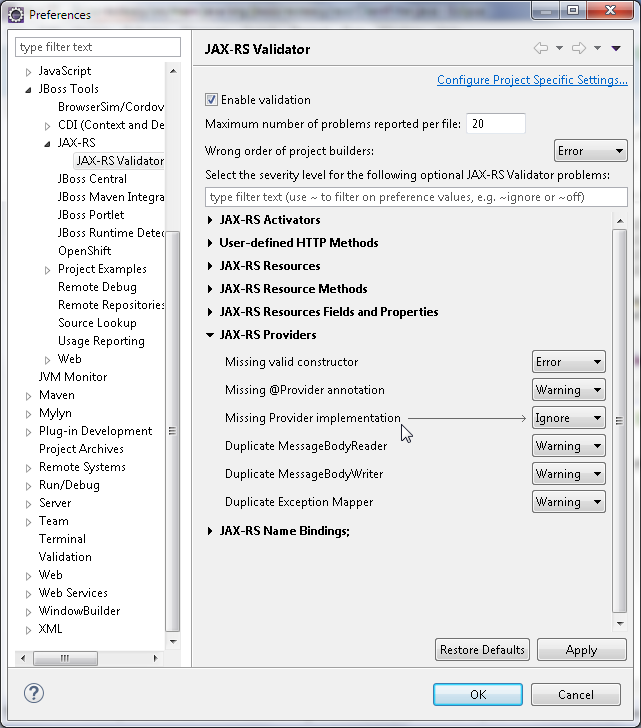
The Provider must implement at least one of the following interfaces: javax.ws.rs.ext.MessageBodyReader, javax.ws.rs.ext.MessageBodyWriter, javax.ws.rs.ext.ExceptionMapper or javax.ws.rs.ext.ContextResolver."

**[ 357 ]**

*Using JAX-RS 2.0 in Java EE 7 with RESTEasy*

To fix the error, select **Windows** | **Preferences**. In **Preferences,** select **JBoss Tools** | **JAX-RS** | **JAX-RS Validator**. Select **JAX-RS Providers** and set **Missing**

**Provider implementation** to **Ignore**, and click on **Apply** and **OK**, as shown in the following screenshot:



## Summary

##### In this chapter, we discussed the salient new features in JAX RS 2.0 with an example using the RESTEasy implementation. We discussed the new client API, the filters and interceptors, asynchronous processing, cancelling a request, and using EJB as a REST Web service resource.

In the next chapter, we will discuss another new feature introduced in Java EE 7, support for JSON processing.

**[ 358 ]**

Processing