Jersey 2.1 User Guide

Jersey 2.1 User Guide

Table of Contents

List of Tables

List of Examples

Preface

This is user guide for Jersey 2.1. We are trying to keep it up to date as we add new features. When reading the user guide, please consult also our Jersey API documentation [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/index.html] as an additional source of information about Jersey features and API.

If you would like to contribute to the guide or have questions on things not covered in our docs, please contact us atusers@jersey.java.net [mailto:users@jersey.java.net]. Similarly, in case you spot any errors in the Jersey documentation, please report them by filing a new issue in our Jersey JIRA Issue Tracker [http://java.net/jira/browse/JERSEY] under `docs` component. Please make sure to specify the version of the Jersey User Guide where the error has been spotted by selecting the proper value for the `Affected Version` field.

Text formatting conventions

First mention of any Jersey and JAX-RS API component in a section links to the API documentation of the referenced component. Any sub-sequent mentions of the component in the same chapter are rendered using a `monospace` font.

Emphasised font is used to a call attention to a newly introduce concept, when it first occurs in the text.

In some of the code listings, certain lines are too long to be displayed on one line for the available page width. In such case, the lines that exceed the available page width are broken up into multiple lines using a `'\'` at the end of each line to indicate that a break has been introduced to fit the line in the page. For example:

```
This is an overly long line that \
might not fit the available page \
width and had to be broken into \
multiple lines.

This line fits the page width.
```

Should read as:

```
This is an overly long line that might not fit the available page width and had to

This line fits the page width.
```

Chapter 1. Getting Started

This chapter provides a quick introduction on how to get started building RESTful services using Jersey. The example described here uses the lightweight Grizzly HTTP server. At the end of this chapter you will see how to implement equivalent functionality as a JavaEE web application you can deploy on any servlet container supporting Servlet 2.5 and higher.

1.1. Creating a New Project from Maven Archetype

Jersey project is build using Apache Maven [http://maven.apache.org/] software project build and management tool. All modules produced as part of Jersey project build are pushed to the Central Maven Repository [http://search.maven.org/]. Therefore it is very convenient to work with Jersey for any Maven-based project as all the released (non-SNAPSHOT) Jersey dependencies are readily available without a need to configure a special maven repository to consume the Jersey modules.

Note

In case you want to depend on the latest SNAPSHOT versions of Jersey modules, the following repository configuration needs to be added to your Maven project pom:

```
<repository>
    <id>snapshot-repository.java.net</id>
    <name>Java.net Snapshot Repository for Maven</name>
    <url>https://maven.java.net/content/repositories/snapshots/</url>
    <layout>default</layout>
</repository>
```

Since starting from a Maven project is the most convenient way for working with Jersey, let's now have a look at this approach. We will now create a new Jersey project that runs on top of a Grizzly [http://grizzly.java.net/] container. We will use a Jersey-provided maven archetype. To create the project, execute the following Maven command in the directory where the new project should reside:

```
mvn archetype:generate -DarchetypeArtifactId=jersey-quickstart-grizzly2 \
-DarchetypeGroupId=org.glassfish.jersey.archetypes -DinteractiveMode=false \
-DgroupId=com.example -DartifactId=simple-service -Dpackage=com.example \
-DarchetypeVersion=2.1
```

Feel free to adjust the `groupId`, `package` and/or `artifactId` of your new project. Alternatively, you can change it by updating the new project pom.xml once it gets generated.

1.2. Exploring the Newly Created Project

Once the project generation from a Jersey maven archetype is successfully finished, you should see the new `simple-service` project directory created in your current location. The directory contains a standard Maven project structure:

Project build and management configuration is described in the `pom.xml` located in the project root directory.
Project sources are located under `src/main/java`.

Project test sources are located under `src/test/java`.
There are 2 classes in the project source directory in the `com.example` package. The `Main` class is responsible for bootstrapping the Grizzly container as well as configuring and deploying the project's JAX-RS application to the container. Another class in the same package is `MyResource` class, that contains implementation of a simple JAX-RS resource. It looks like this:

```
 1 package com.example;
 2
 3 import javax.ws.rs.GET;
 4 import javax.ws.rs.Path;
 5 import javax.ws.rs.Produces;
 6 import javax.ws.rs.core.MediaType;
 7
 8 /**
 9  * Root resource (exposed at "myresource" path)
10  */
11 @Path("myresource")
12 public class MyResource {
13
14     /**
15      * Method handling HTTP GET requests. The returned object will be sent
16      * to the client as "text/plain" media type.
17      *
18      * @return String that will be returned as a text/plain response.
19      */
20     @GET
21     @Produces(MediaType.TEXT_PLAIN)
22     public String getIt() {
23         return "Got it!";
24     }
25 }
```

A JAX-RS resource is an annotated POJO that provides so-called *resource methods* that are able to handle HTTP requests for URI paths that the resource is bound to. See Chapter 3, *JAX-RS Application, Resources and Sub-Resources* for a complete guide to JAX-RS resources. In our case, the resource exposes a single resource method that is able to handle HTTP `GET` requests, is bound to `/myresource` URI path and can produce responses with response message content represented in `"text/plain"` media type. In this version, the resource returns the same `"Got it!"` response to all client requests.

The last piece of code that has been generated in this skeleton project is a `MyResourceTest` unit test class that is located in the same `com.example` package as the `MyResource` class, however, this unit test class is placed into the maven project test source directory `src/test/java` (certain code comments and JUnit imports have been excluded for brevity):

```
 1 package com.example;
 2
 3 import javax.ws.rs.client.Client;
 4 import javax.ws.rs.client.ClientBuilder;
 5 import javax.ws.rs.client.WebTarget;
 6
 7 import org.glassfish.grizzly.http.server.HttpServer;
 8
 9 ...
10
11 public class MyResourceTest {
```

```
12
13      private HttpServer server;
14      private WebTarget target;
15
16      @Before
17      public void setUp() throws Exception {
18          server = Main.startServer();
19
20          Client c = ClientBuilder.newClient();
21          target = c.target(Main.BASE_URI);
22      }
23
24      @After
25      public void tearDown() throws Exception {
26          server.stop();
27      }
28
29      /**
30       * Test to see that the message "Got it!" is sent in the response.
31       */
32      @Test
33      public void testGetIt() {
34          String responseMsg = target.path("myresource").request().get(String.cl
35          assertEquals("Got it!", responseMsg);
36      }
37  }
```

In this unit test, a Grizzly container is first started and server application is deployed in the test `setUp()` method by a static call to `Main.startServer()`. Next, a JAX-RS client components are created in the same test set-up method. First a new JAX-RS client instance `c` is built and then a JAX-RS web target component pointing to the context root of our application deployed at `http://localhost:8080/myapp/` (a value of `Main.BASE_URI` constant) is stored into a `target` field of the unit test class. This field is then used in the actual unit test method (`testGetIt()`).

In the `testGetIt()` method a fluent JAX-RS Client API is used to connect to and send a HTTP `GET` request to the `MyResource` JAX-RS resource class listening on `/myresource` URI. As part of the same fluent JAX-RS API method invocation chain, a response is read as a Java `String` type. On the second line in the test method, the response content string returned from the server is compared with the expected phrase in the test assertion. To learn more about using JAX-RS Client API, please see the Chapter 5, *Client API* chapter.

1.3. Running the Project

Now that we have seen the content of the project, let's try to test-run it. To do this, we need to invoke following command on the command line:

```
mvn clean test
```

This will compile the project and run the project unit tests. We should see a similar output that informs about a successful build once the build is finished:

```
Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 34.527s
[INFO] Finished at: Sun May 26 19:26:24 CEST 2013
[INFO] Final Memory: 17M/490M
[INFO] ------------------------------------------------------------------------
```

Now that we have verified that the project compiles and that the unit test passes, we can execute the application in a standalone mode. To do this, run the following maven command:

```
mvn clean exec:java
```

The application starts and you should soon see the following notification in your console:

```
May 26, 2013 8:08:45 PM org.glassfish.grizzly.http.server.NetworkListener start
INFO: Started listener bound to [localhost:8080]
May 26, 2013 8:08:45 PM org.glassfish.grizzly.http.server.HttpServer start
INFO: [HttpServer] Started.
Jersey app started with WADL available at http://localhost:8080/myapp/application.
Hit enter to stop it...
```

This informs you that the application has been started and it's WADL descriptor is available at `http://localhost:8080/myapp/application.wadl` URL. You can retrieve the WADL content by executing a `curl http://localhost:8080/myapp/application.wadl` command in your console or by typing the WADL URL into your favorite browser. You should get back an XML document in describing your deployed RESTful application in a WADL format. To learn more about working with WADL, check the Chapter 15, *WADL Support* chapter.

The last thing we should try before concluding this section is to see if we can communicate with our resource deployed at `/myresource` path. We can again either type the resource URL in the browser or we can use `curl`:

```
$ curl http://localhost:8080/myapp/myresource
Got it!
```

As we can see, the `curl` command returned with the `Got it!` message that was sent by our resource. We can also ask `curl` to provide more information about the response, for example we can let it display all response headers by using the `-i` switch:

```
curl -i http://localhost:8080/myapp/myresource
HTTP/1.1 200 OK
Content-Type: text/plain
Date: Sun, 26 May 2013 18:27:19 GMT
Content-Length: 7

Got it!
```

Here we see the whole content of the response message that our Jersey/JAX-RS application returned, including all the HTTP headers. Notice the `Content-Type: text/plain` header that was derived from the value of @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] annotation attached to the `MyResource` class.

In case you want to see even more details about the communication between our `curl` client and our resource running on Jersey in a Grizzly I/O container, feel free to try other various options and switches

that `curl` provides. For example, this last command will make `curl` output a lot of additional information about the whole communication:

```
$ curl -v http://localhost:8080/myapp/myresource
* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* Connection refused
*   Trying 127.0.0.1...
* connected
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /myapp/myresource HTTP/1.1
> User-Agent: curl/7.25.0 (x86_64-apple-darwin11.3.0) libcurl/7.25.0 OpenSSL/1.0.1
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Date: Sun, 26 May 2013 18:29:18 GMT
< Content-Length: 7
<
* Connection #0 to host localhost left intact
Got it!* Closing connection #0
```

1.4. Creating a JavaEE Web Application

To create a Web Application that can be packaged as WAR and deployed in a Servlet container follow a similar process to the one described in Section 1.1, “Creating a New Project from Maven Archetype”. In addition to the Grizzly-based archetype, Jersey provides also a Maven arcehtype for creating web application skeletons. To create the new web application skeleton project, execute the following Maven command in the directory where the new project should reside:

```
mvn archetype:generate -DarchetypeArtifactId=jersey-quickstart-webapp \
                -DarchetypeGroupId=org.glassfish.jersey.archetypes -DinteractiveMo
                -DgroupId=com.example -DartifactId=simple-service-webapp -Dpackage
                -DarchetypeVersion=2.1
```

As with the Grizzly based project, feel free to adjust the `groupId`, `package` and/or `artifactId` of your new web application project. Alternatively, you can change it by updating the new project pom.xml once it gets generated.

Once the project generation from a Jersey maven archetype is successfully finished, you should see the new `simple-service-webapp` project directory created in your current location. The directory contains a standard Maven project structure, similar to the `simple-service` project content we have seen earlier, except it is extended with an additional web application specific content:

Project build and management configuration is described in the `pom.xml` located in the project root directory.
Project sources are located under `src/main/java`.
Project resources are located under `src/main/resources`.
Project web application files are located under `src/main/webapp`.
The project contains the same `MyResouce` JAX-RS resource class. It does not contain any unit tests as well as it does not contain a `Main` class that was used to setup Grizzly container in the previous project. Instead, it contains the standard Java EE web application `web.xml` deployment descriptor under `src/main/webapp/WEB-INF`. The last component in the project is an `index.jsp` page that serves as a client for the `MyResource` resource class that is packaged and deployed with the application.

To compile and package the application into a WAR, invoke the following maven command in your console:

```
mvn clean package
```

A successfull build output will produce an output similar to the one below:

```
Results :

Tests run: 0, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-war-plugin:2.1.1:war (default-war) @ simple-service-webapp ---
[INFO] Packaging webapp
[INFO] Assembling webapp [simple-service-webapp] in [.../simple-service-webapp/tar
[INFO] Processing war project
[INFO] Copying webapp resources [.../simple-service-webapp/src/main/webapp]
[INFO] Webapp assembled in [75 msecs]
[INFO] Building war: .../simple-service-webapp/target/simple-service-webapp.war
[INFO] WEB-INF/web.xml already added, skipping
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 9.067s
[INFO] Finished at: Sun May 26 21:07:44 CEST 2013
[INFO] Final Memory: 17M/490M
[INFO] ------------------------------------------------------------------------
```

Now you are ready to take the packaged WAR (located under `./target/simple-service-webapp.war`) and deploy it to a Servlet container of your choice.

Important

To deploy a Jersey application, you will need a Servlet container that supports Servlet 2.5 or later. For full set of advanced features (such as JAX-RS 2.0 Async Support) you will need a Servlet 3.0 or later compliant container.

1.5. Exploring Other Jersey Examples

In the sections above, we have covered an approach how to get dirty with Jersey quickly. Please consult the other sections of the Jersey User Guide to learn more about Jersey and JAX-RS. Even though we try our best to cover as much as possibly in the User Guide, there is always a chance that you would not be able to get a full answer to the problem you are solving. In that case, consider diving in our examples that provide additional tips and hints to the features you may want to use in your projects.

Jersey codebase contains a number of useful examples on how to use various JAX-RS and Jersey features. Feel free to browse through the code of individual Jersey Examples [https://github.com/jersey/jersey/tree/2.1/examples] in the Jersey source repository. For off-line browsing, you can also download a bundle with all the examples from here [https://maven.java.net/content/repositories/releases/org/glassfish/jersey/bundles/jersey-examples/2.1/].

Chapter 2. Modules and dependencies

2.1. Java SE Compatibility

All Jersey components are compiled with Java SE 6 target. It means, you will also need at least Java SE 6 to be able to compile and run your application.

2.2. Introduction to Jersey dependencies

Jersey is built, assembled and installed using Apache Maven [http://maven.apache.org/]. Non-snapshot Jersey releases are deployed to the Central Maven Repository [http://search.maven.org/]. Jersey is also being deployed to Java.Net Maven repositories [http://maven.java.net/], which contain also Jersey SNAPSHOT versions. In case you would want to test the latest development builds check out the Java.Net Snapshots Maven repository [https://maven.java.net/content/repositories/snapshots/org/glassfish/jersey].

An application that uses Jersey and depends on Jersey modules is in turn required to also include in the application dependencies the set of 3rd party modules that Jersey modules depend on. Jersey is designed as a pluggable component architecture and different applications can therefore require different sets of Jersey modules. This also means that the set of external Jersey dependencies required to be included in the application dependencies may vary in each application based on the Jersey modules that are being used by the application.

Developers using Maven or a Maven-aware build system in their projects are likely to find it easier to include and manage dependencies of their applications compared to developers using ant or other build systems that are not compatible with Maven. This document will explain to both maven and non-maven developers how to depend on Jersey modules in their application. Ant developers are likely to find the Ant Tasks for Maven [http://maven.apache.org/ant-tasks/index.html] very useful.

2.3. Common Jersey Use Cases

2.3.1. Servlet based application on Glassfish

If you are using Glassfish application server, you don't need to package anything with your application, everything is already included. You just need to declare (provided) dependency on JAX-RS API to be able to compile your application.

```
<dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>javax.ws.rs-api</artifactId>
    <version>2.0</version>
    <scope>provided</scope>
</dependency>
```

If you are using any Jersey specific feature, you will need to depend on Jersey directly.

```
<dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-servlet</artifactId>
    <version>2.1</version>
    <scope>provided</scope>
```

```
</dependency>
<!-- if you are using Jersey client specific features -->
<dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>2.1</version>
    <scope>provided</scope>
</dependency>
```

2.3.2. Servlet based server-side application

Following dependencies apply to application server (servlet containers) without any integrated JAX-RS implementation. Then application needs to include JAX-RS API and Jersey implementation in deployed application.

```
<dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <!-- if your container implements Servlet API older than 3.0, use "jersey-cont
    <artifactId>jersey-container-servlet</artifactId>
    <version>2.1</version>
</dependency>
<!-- Required only when you are using JAX-RS Client -->
<dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>2.1</version>
</dependency>
```

2.3.3. Client application on JDK

Applications running on plain JDK using only client part of JAX-RS specification need to depend only on client. There are various additional modules which can be added, like for example grizzly or apache connector (see dependencies snipped below). Jersey client runs by default with plain JDK (using HttpUrlConnection). See Chapter 5, *Client API*. for more details.

```
<dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>2.1</version>
</dependency>
```

Currently available connectors:

```
<dependency>
    <groupId>org.glassfish.jersey.connectors</groupId>
    <artifactId>jersey-grizzly-connector</artifactId>
    <version>2.1</version>
</dependency>

<dependency>
    <groupId>org.glassfish.jersey.connectors</groupId>
    <artifactId>jersey-apache-connector</artifactId>
```

```
    <version>2.1</version>
</dependency>
```

2.3.4. Server-side application on supported containers

Apart for a standard JAX-RS Servlet-based deployment that works with any Servlet container that supports Servlet 2.5 and higher, Jersey provides support for programmatic deployment to the following containers: Grizzly 2 (HTTP and Servlet), JDK Http server and Simple Http server. This chapter presents only required maven dependencies, more information can be found in Chapter 4, *Deploying a RESTful Web Service*.

```
<dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-grizzly2-http</artifactId>
    <version>2.1</version>
</dependency>
<dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-grizzly2-servlet</artifactId>
    <version>2.1</version>
</dependency>
<dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-jdk-http</artifactId>
    <version>2.1</version>
</dependency>
<dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-simple-http</artifactId>
    <version>2.1</version>
</dependency>
```

2.4. List of modules

The following chapters provide an overview of all Jersey modules and their dependencies with links to the respective binaries (follow a link on a module name to get complete set of downloadable dependencies).

Table 2.1. Jersey Core

Core	
jersey-client [https://jersey.java.net/project-info/2.1/jersey/jersey-client/dependencies.html]	Jersey core client implementation.
jersey-common [https://jersey.java.net/project-info/2.1/jersey/jersey-common/dependencies.html]	Jersey core common packages.

jersey-server [https://jersey.java.net/project-info/2.1/jersey/jersey-server/dependencies.html]	Jersey core server implementation.

Table 2.2. Jersey Containers

Containers	
jersey-container-grizzly2-http [https://jersey.java.net/project-info/2.1/jersey/project/jersey-container-grizzly2-http/dependencies.html]	Grizzly 2 Http Container.
jersey-container-grizzly2-servlet [https://jersey.java.net/project-info/2.1/jersey/project/jersey-container-grizzly2-servlet/dependencies.html]	Grizzly 2 Servlet Container.
jersey-container-jdk-http [https://jersey.java.net/project-info/2.1/jersey/project/jersey-container-jdk-http/dependencies.html]	JDK Http Container.
jersey-container-servlet [https://jersey.java.net/project-info/2.1/jersey/project/jersey-container-servlet/dependencies.html]	Jersey core Servlet 3.x implementation.
jersey-container-servlet-core [https://jersey.java.net/project-info/2.1/jersey/project/jersey-container-servlet-core/dependencies.html]	Jersey core Servlet 2.x implementation.
jersey-container-simple-http [https://	Simple Http Container.

jersey.java.net/ project-info/2.1/ jersey/project/ jersey-container-simple-http/ dependencies.html]	

Table 2.3. Jersey Connectors

Connectors	
jersey-grizzly-connector [https://jersey.java.net/project-info/2.1/jersey/project/jersey-grizzly-connector/dependencies.html]	Jersey Client Transport via Grizzly.
jersey-apache-connector [https://jersey.java.net/project-info/2.1/jersey/project/jersey-apache-connector/dependencies.html]	Jersey Client Transport via Apache.

Table 2.4. Jersey Media

Media	
jersey-media-json-jackson [https://jersey.java.net/project-info/2.1/jersey/project/jersey-media-json-jackson/dependencies.html]	Jersey JSON Jackson entity providers support module.
jersey-media-json-jettison [https://jersey.java.net/project-info/2.1/jersey/project/jersey-media-json-jettison/dependencies.html]	Jersey JSON Jettison entity providers support module.
jersey-media-json-processing [https://jersey.java.net/project-info/2.1/jersey/project/jersey-media-json-processing/dependencies.html]	Jersey JSON-P (JSR 353) entity providers support proxy module.
jersey-media-moxy [https://	Jersey JSON entity providers support module based on EclipseLink MOXy.

jersey.java.net/project-info/2.1/jersey/project/jersey-media-moxy/dependencies.html]	
jersey-media-multipart [https://jersey.java.net/project-info/2.1/jersey/project/jersey-media-multipart/dependencies.html]	Jersey Multipart entity providers support module.
jersey-media-sse [https://jersey.java.net/project-info/2.1/jersey/project/jersey-media-sse/dependencies.html]	Jersey Server Sent Events entity providers support module.

Table 2.5. Jersey Extensions

Extensions	
jersey-bean-validation [https://jersey.java.net/project-info/2.1/jersey/project/jersey-bean-validation/dependencies.html]	Jersey extension module providing support for Bean Validation (JSR-349) API.
jersey-mvc [https://jersey.java.net/project-info/2.1/jersey/project/jersey-mvc/dependencies.html]	Jersey extension module providing support for MVC.
jersey-mvc-freemarker [https://jersey.java.net/project-info/2.1/jersey/project/jersey-mvc-freemarker/dependencies.html]	Jersey extension module providing support for Freemarker templates.
jersey-mvc-jsp [https://jersey.java.net/project-info/2.1/jersey/project/jersey-mvc-jsp/dependencies.html]	Jersey extension module providing support for JSP templates.

jersey-proxy-client [https://jersey.java.net/project-info/2.1/jersey/project/jersey-proxy-client/dependencies.html]	Jersey extension module providing support for (proxy-based) high-level client API.
jersey-servlet-portability [https://jersey.java.net/project-info/2.1/jersey/project/jersey-servlet-portability/dependencies.html]	Library that enables writing web applications that run with both Jersey 1.x and Jersey 2.x servlet containers.
jersey-wadl-doclet [https://jersey.java.net/project-info/2.1/jersey/project/jersey-wadl-doclet/dependencies.html]	A doclet that generates a resourcedoc xml file: this file contains the javadoc documentation of resource classes, so that this can be used for extending generated wadl with useful documentation.

Table 2.6. Jersey Test Framework

Test Framework	
jersey-test-framework-core [https://jersey.java.net/project-info/2.1/jersey/project/jersey-test-framework-core/dependencies.html]	Jersey Test Framework Core.
jersey-test-framework-provider-bundle [https://jersey.java.net/project-info/2.1/jersey/project/project/jersey-test-framework-provider-bundle/dependencies.html]	Jersey Test Framework Providers Bundle.
jersey-test-framework-provider-default-client [https://jersey.java.net/project-info/2.1/jersey/project/project/jersey-test-framework-provider-	Jersey Test Framework Default Client Provider.

default-client/ dependencies.html]	
jersey-test-framework-provider-external [https://jersey.java.net/project-info/2.1/jersey/project/project/jersey-test-framework-provider-external/dependencies.html]	Jersey Test Framework - External container.
jersey-test-framework-provider-grizzly2 [https://jersey.java.net/project-info/2.1/jersey/project/project/jersey-test-framework-provider-grizzly2/dependencies.html]	Jersey Test Framework - Grizzly2 container.
jersey-test-framework-provider-inmemory [https://jersey.java.net/project-info/2.1/jersey/project/project/jersey-test-framework-provider-inmemory/dependencies.html]	Jersey Test Framework - InMemory container.
jersey-test-framework-provider-jdk-http [https://jersey.java.net/project-info/2.1/jersey/project/project/jersey-test-framework-provider-jdk-http/dependencies.html]	Jersey Test Framework - JDK HTTP container.
jersey-test-framework-provider-simple [https://jersey.java.net/project-info/2.1/jersey/project/project/jersey-test-framework-	Jersey Test Framework - Simple HTTP container.

provider-simple/ dependencies.html]	

Table 2.7. Jersey Glassfish Bundles

Glassfish Bundles	
jersey-gf-cdi [https://jersey.java.net/project-info/2.1/jersey/project/project/jersey-gf-cdi/dependencies.html]	Jersey CDI for GlassFish integration.
jersey-gf-ejb [https://jersey.java.net/project-info/2.1/jersey/project/project/jersey-gf-ejb/dependencies.html]	Jersey EJB for GlassFish integration.

Table 2.8. Jersey Examples

Examples	
clipboard [https://jersey.java.net/project-info/2.1/jersey/project/clipboard/dependencies.html]	Jersey clipboard example.
clipboard-programmatic [https://jersey.java.net/project-info/2.1/jersey/project/clipboard-programmatic/dependencies.html]	Jersey programmatic resource API clipboard example.
exception-mapping [https://jersey.java.net/project-info/2.1/jersey/project/exception-mapping/dependencies.html]	Jersey example showing exception mappers in action.
helloworld [https://jersey.java.net/project-info/2.1/jersey/project/helloworld/dependencies.html]	Jersey annotated resource class "Hello world" example.

helloworld-programmatic [https://jersey.java.net/project-info/2.1/jersey/project/helloworld-programmatic/dependencies.html]	Jersey programmatic resource API "Hello world" example.
helloworld-pure-jax-rs [https://jersey.java.net/project-info/2.1/jersey/project/helloworld-pure-jax-rs/dependencies.html]	Example using only the standard JAX-RS API's and the lightweight HTTP server bundled in JDK.
http-trace [https://jersey.java.net/project-info/2.1/jersey/project/http-trace/dependencies.html]	Jersey HTTP TRACE support example
https-clientserver-grizzly [https://jersey.java.net/project-info/2.1/jersey/project/https-clientserver-grizzly/dependencies.html]	Jersey HTTPS Client/Server example on Grizzly.
jaxb [https://jersey.java.net/project-info/2.1/jersey/project/jaxb/dependencies.html]	Jersey JAXB example.
jaxrs-types-injection [https://jersey.java.net/project-info/2.1/jersey/project/jaxrs-types-injection/dependencies.html]	Jersey JAX-RS types injection example.
json-jackson [https://jersey.java.net/project-info/2.1/jersey/project/json-jackson/dependencies.html]	Jersey JSON with Jackson example.
json-jettison [https://	Jersey JSON with Jettison JAXB example.

jersey.java.net/project-info/2.1/jersey/project/json-jettison/dependencies.html]	
json-moxy [https://jersey.java.net/project-info/2.1/jersey/project/json-moxy/dependencies.html]	Jersey JSON with MOXy example.
json-with-padding [https://jersey.java.net/project-info/2.1/jersey/project/json-with-padding/dependencies.html]	Jersey JSON with Padding example.
managed-client [https://jersey.java.net/project-info/2.1/jersey/project/managed-client/dependencies.html]	Jersey managed client example.
osgi-helloworld-webapp [https://jersey.java.net/project-info/2.1/jersey/project/osgi-helloworld-webapp/dependencies.html]	OSGi Helloworld.
osgi-helloworld-webapp [https://jersey.java.net/project-info/2.1/jersey/project/osgi-helloworld-webapp/lib-bundle/dependencies.html]	OSGi Helloworld - bundle.
osgi-helloworld-webapp [https://jersey.java.net/project-info/2.1/jersey/project/osgi-helloworld-webapp/war-bundle/dependencies.html]	OSGi Helloworld - war.
osgi-http-service [https://jersey.java.net/	OSGi Helloworld.

project-info/2.1/jersey/project/osgi-http-service/bundle/dependencies.html]	
osgi-http-service [https://jersey.java.net/project-info/2.1/jersey/project/osgi-http-service/dependencies.html]	OSGi Helloworld.
reload [https://jersey.java.net/project-info/2.1/jersey/project/reload/dependencies.html]	Jersey resource configuration reload example.
server-async [https://jersey.java.net/project-info/2.1/jersey/project/server-async/dependencies.html]	Jersey JAX-RS asynchronous server-side example.
server-async-managed [https://jersey.java.net/project-info/2.1/jersey/project/server-async-managed/dependencies.html]	Jersey JAX-RS asynchronous server-side example with custom Jersey executor providers.
server-async-standalone [https://jersey.java.net/project-info/2.1/jersey/project/server-async-standalone/server-async-standalone-client/dependencies.html]	Standalone Jersey JAX-RS asynchronous server-side processing example.
server-sent-events [https://jersey.java.net/project-info/2.1/jersey/project/server-sent-events/dependencies.html]	Jersey Server-Sent Events example.
simple-console [https://jersey.java.net/project-info/2.1/jersey/project/	Jersey Simple Console example.

simple-console/ dependencies.html]	
sse-twitter-aggregator [https://jersey.java.net/project-info/2.1/jersey/project/sse-twitter-aggregator/dependencies.html]	Jersey SSE Twitter Message Aggregator Example.
system-properties-example [https://jersey.java.net/project-info/2.1/jersey/project/system-properties-example/dependencies.html]	Jersey system properties example.
xml-moxy [https://jersey.java.net/project-info/2.1/jersey/project/xml-moxy/dependencies.html]	Jersey XML MOXy example.

Table 2.9. Jersey Examples - WebApps

Examples - WAR	
bean-validation-webapp [https://jersey.java.net/project-info/2.1/jersey/project/webapp-example-parent/bean-validation-webapp/dependencies.html]	Jersey Bean Validation (JSR-349) example.
bookmark [https://jersey.java.net/project-info/2.1/jersey/project/webapp-example-parent/bookmark/dependencies.html]	Jersey Bookmark example.
bookmark-em [https://jersey.java.net/project-info/2.1/jersey/project/webapp-example-parent/bookmark-em/dependencies.html]	Jersey Bookmark example using EntityManager.

bookstore-webapp [https://jersey.java.net/project-info/2.1/jersey/project/webapp-example-parent/bookstore-webapp/dependencies.html]	Jersey MVC Bookstore example.
cdi-webapp [https://jersey.java.net/project-info/2.1/jersey/project/webapp-example-parent/cdi-webapp/dependencies.html]	Jersey CDI example.
extended-wadl-webapp [https://jersey.java.net/project-info/2.1/jersey/project/webapp-example-parent/extended-wadl-webapp/dependencies.html]	Extended WADL example.
freemarker-webapp [https://jersey.java.net/project-info/2.1/jersey/project/webapp-example-parent/freemarker-webapp/dependencies.html]	Jersey Freemarker example.
helloworld-webapp [https://jersey.java.net/project-info/2.1/jersey/project/webapp-example-parent/helloworld-webapp/dependencies.html]	Jersey annotated resource class "Hello world" example.
https-server-glassfish [https://jersey.java.net/project-info/2.1/jersey/project/webapp-example-parent/https-	Jersey HTTPS server on GlassFish example.

server-glassfish/ dependencies.html]	
jersey-ejb [https:// jersey.java.net/ project-info/2.1/ jersey/project/ webapp-example- parent/jersey-ejb/ dependencies.html]	Jersey EJB example.
json-processing- webapp [https:// jersey.java.net/ project-info/2.1/ jersey/project/ webapp-example- parent/json- processing-webapp/ dependencies.html]	Jersey JSON-P (JSR 353) example.
managed-beans- webapp [https:// jersey.java.net/ project-info/2.1/ jersey/project/ webapp-example- parent/managed- beans-webapp/ dependencies.html]	Jersey Managed Beans Web Application example.
managed- client-simple- webapp [https:// jersey.java.net/ project-info/2.1/ jersey/project/ webapp- example-parent/ managed-client- simple-webapp/ dependencies.html]	Jersey Managed Client Simple Webapp example.
managed-client- webapp [https:// jersey.java.net/ project-info/2.1/ jersey/project/ webapp-example- parent/managed- client-webapp/ dependencies.html]	Jersey managed client web application example.
multipart- webapp [https:// jersey.java.net/ project-info/2.1/	Jersey Multipart example.

jersey/project/webapp-example-parent/multipart-webapp/dependencies.html]	
sse-item-store-webapp [https://jersey.java.net/project-info/2.1/jersey/project/webapp-example-parent/sse-item-store-webapp/dependencies.html]	Jersey SSE-based item store example.

Chapter 3. JAX-RS Application, Resources and Sub-Resources

This chapter presents an overview of the core JAX-RS concepts - resources and sub-resources.

The JAX-RS 2.0 JavaDoc can be found online here [http://jax-rs-spec.java.net/nonav/2.0/apidocs/index.html].

The JAX-RS 2.0 specification draft can be found online here [http://jcp.org/en/jsr/summary?id=339].

3.1. Root Resource Classes

Root resource classes are POJOs (Plain Old Java Objects) that are annotated with @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] have at least one method annotated with @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] or a resource method designator annotation such as @GET [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/GET.html], @PUT [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PUT.html], @POST [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/POST.html], @DELETE [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/DELETE.html]. Resource methods are methods of a resource class annotated with a resource method designator. This section shows how to use Jersey to annotate Java objects to create RESTful web services.

The following code example is a very simple example of a root resource class using JAX-RS annotations. The example code shown here is from one of the samples that ships with Jersey, the zip file of which can be found in the maven repository here [https://maven.java.net/content/repositories/releases/org/glassfish/jersey/examples/helloworld/2.1/].

Example 3.1. Simple hello world root resource class

```
 1 package org.glassfish.jersey.examples.helloworld;
 2
 3 import javax.ws.rs.GET;
 4 import javax.ws.rs.Path;
 5 import javax.ws.rs.Produces;
 6
 7 @Path("helloworld")
 8 public class HelloWorldResource {
 9     public static final String CLICHED_MESSAGE = "Hello World!";
10
11 @GET
12 @Produces("text/plain")
13     public String getHello() {
14         return CLICHED_MESSAGE;
15     }
16 }
```

Let's look at some of the JAX-RS annotations used in this example.

3.1.1. @Path

The @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] annotation's value is a relative URI path. In the example above, the Java class will be hosted at the URI path `/helloworld`.

This is an extremely simple use of the @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] annotation. What makes JAX-RS so useful is that you can embed variables in the URIs.

URI path templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by curly braces. For example, look at the following @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] annotation:

```
@Path("/users/{username}")
```

In this type of example, a user will be prompted to enter their name, and then a Jersey web service configured to respond to requests to this URI path template will respond. For example, if the user entered their username as "Galileo", the web service will respond to the following URL: `http://example.com/users/Galileo`

To obtain the value of the username variable the @PathParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PathParam.html] may be used on method parameter of a request method, for example:

Example 3.2. Specifying URI path parameter

```
1 @Path("/users/{username}")
2 public class UserResource {
3
4     @GET
5     @Produces("text/xml")
6     public String getUser(@PathParam("username") String userName) {
7         ...
8     }
9 }
```

If it is required that a user name must only consist of lower and upper case numeric characters then it is possible to declare a particular regular expression, which overrides the default regular expression, "[^/]+?", for example:

```
@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]*}")
```

In this type of example the username variable will only match user names that begin with one upper or lower case letter and zero or more alpha numeric characters and the underscore character. If a user name does not match that a 404 (Not Found) response will occur.

A @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] value may or may not begin with a '/', it makes no difference. Likewise, by default, a @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] value may or may not end in a '/', it makes no difference, and thus request URLs that end or do not end in a '/' will both be matched.

3.1.2. @GET, @PUT, @POST, @DELETE, ... (HTTP Methods)

@GET [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/GET.html], @PUT [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PUT.html], @POST [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/POST.html], @DELETE [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/DELETE.html] and @HEAD [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/HEAD.html] are *resource method designator* annotations defined by JAX-RS and which correspond to the similarly named HTTP methods. In the example above, the annotated Java method will process HTTP GET requests. The behavior of a resource is determined by which of the HTTP methods the resource is responding to.

The following example is an extract from the storage service sample that shows the use of the PUT method to create or update a storage container:

Example 3.3. PUT method

```
 1 @PUT
 2 public Response putContainer() {
 3     System.out.println("PUT CONTAINER " + container);
 4
 5     URI uri = uriInfo.getAbsolutePath();
 6     Container c = new Container(container, uri.toString());
 7
 8     Response r;
 9     if (!MemoryStore.MS.hasContainer(c)) {
10         r = Response.created(uri).build();
11     } else {
12         r = Response.noContent().build();
13     }
14
15     MemoryStore.MS.createContainer(c);
16     return r;
17 }
```

By default the JAX-RS runtime will automatically support the methods HEAD and OPTIONS, if not explicitly implemented. For HEAD the runtime will invoke the implemented GET method (if present) and ignore the response entity (if set). A response returned for the OPTIONS method depends on the requested media type defined in the 'Accept' header. The OPTIONS method can return a response with a set of supported resource methods in the 'Allow' header or return a WADL [http://wadl.java.net/] document. See wadl section for more information.

3.1.3. @Produces

The @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] annotation is used to specify the MIME media types of representations a resource can produce and send back to the client. In this example, the Java method will produce representations identified by the MIME media type "text/plain". @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] can be applied at both the class and method levels. Here's an example:

Example 3.4. Specifying output MIME type

```
 1 @Path("/myResource")
 2 @Produces("text/plain")
 3 public class SomeResource {
 4     @GET
 5     public String doGetAsPlainText() {
 6         ...
 7     }
 8
 9     @GET
10     @Produces("text/html")
11     public String doGetAsHtml() {
12         ...
13     }
14 }
```

The `doGetAsPlainText` method defaults to the MIME type of the @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] annotation at the class level. The `doGetAsHtml` method's @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] annotation overrides the class-level @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] setting, and specifies that the method can produce HTML rather than plain text.

If a resource class is capable of producing more that one MIME media type then the resource method chosen will correspond to the most acceptable media type as declared by the client. More specifically the Accept header of the HTTP request declares what is most acceptable. For example if the Accept header is "`Accept: text/plain`" then the `doGetAsPlainText` method will be invoked. Alternatively if the Accept header is " `Accept: text/plain;q=0.9, text/html`", which declares that the client can accept media types of "text/plain" and "text/html" but prefers the latter, then the `doGetAsHtml` method will be invoked.

More than one media type may be declared in the same @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] declaration, for example:

Example 3.5. Using multiple output MIME types

```
1 @GET
2 @Produces({"application/xml", "application/json"})
3 public String doGetAsXmlOrJson() {
4     ...
5 }
```

The `doGetAsXmlOrJson` method will get invoked if either of the media types "application/xml" and "application/json" are acceptable. If both are equally acceptable then the former will be chosen because it occurs first.

Optionally, server can also specify the quality factor for individual media types. These are considered if several are equally acceptable by the client. For example:

Example 3.6. Server-side content negotiation

```
1 @GET
2 @Produces({"application/xml; qs=0.9", "application/json"})
3 public String doGetAsXmlOrJson() {
4     ...
5 }
```

In the above sample, if client accepts both "application/xml" and "application/json" (equally), then a server always sends "application/json", since "application/xml" has a lower quality factor.

The examples above refers explicitly to MIME media types for clarity. It is possible to refer to constant values, which may reduce typographical errors, see the constant field values of MediaType [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/MediaType.html].

3.1.4. @Consumes

The @Consumes [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Consumes.html] annotation is used to specify the MIME media types of representations that can be consumed by a resource. The above example can be modified to set the cliched message as follows:

Example 3.7. Specifying input MIME type

```
1 @POST
2 @Consumes("text/plain")
3 public void postClichedMessage(String message) {
4     // Store the message
5 }
```

In this example, the Java method will consume representations identified by the MIME media type "text/plain". Notice that the resource method returns void. This means no representation is returned and response with a status code of 204 (No Content) will be returned to the client.

@Consumes [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Consumes.html] can be applied at both the class and the method levels and more than one media type may be declared in the same @Consumes [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Consumes.html] declaration.

3.2. Parameter Annotations (@*Param)

Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request. One of the previous examples presented the use of @PathParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PathParam.html] to extract a path parameter from the path component of the request URL that matched the path declared in @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html].

@QueryParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/QueryParam.html] is used to extract query parameters from the Query component of the request URL. The following example is an extract from the sparklines sample:

Example 3.8. Query parameters

```
 1 @Path("smooth")
 2 @GET
 3 public Response smooth(
 4     @DefaultValue("2") @QueryParam("step") int step,
 5     @DefaultValue("true") @QueryParam("min-m") boolean hasMin,
 6     @DefaultValue("true") @QueryParam("max-m") boolean hasMax,
 7     @DefaultValue("true") @QueryParam("last-m") boolean hasLast,
 8     @DefaultValue("blue") @QueryParam("min-color") ColorParam minColor,
 9     @DefaultValue("green") @QueryParam("max-color") ColorParam maxColor,
10     @DefaultValue("red") @QueryParam("last-color") ColorParam lastColor) {
11     ...
12 }
```

If a query parameter "step" exists in the query component of the request URI then the "step" value will be will extracted and parsed as a 32 bit signed integer and assigned to the step method parameter. If "step" does not exist then a default value of 2, as declared in the @DefaultValue [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/DefaultValue.html] annotation, will be assigned to the step method parameter. If the "step" value cannot be parsed as a 32 bit signed integer then a HTTP 404 (Not Found) response is returned. User defined Java types such as `ColorParam` may be used, which as implemented as follows:

Example 3.9. Custom Java type for consuming request parameters

```
 1 public class ColorParam extends Color {
 2
 3     public ColorParam(String s) {
 4         super(getRGB(s));
 5     }
 6
 7     private static int getRGB(String s) {
 8         if (s.charAt(0) == '#') {
 9             try {
10                 Color c = Color.decode("0x" + s.substring(1));
11                 return c.getRGB();
12             } catch (NumberFormatException e) {
13                 throw new WebApplicationException(400);
14             }
15         } else {
16             try {
17                 Field f = Color.class.getField(s);
18                 return ((Color)f.get(null)).getRGB();
19             } catch (Exception e) {
20                 throw new WebApplicationException(400);
21             }
22         }
23     }
24 }
```

In general the Java type of the method parameter may:

1. Be a primitive type;

2. Have a constructor that accepts a single `String` argument;

3. Have a static method named `valueOf` or `fromString` that accepts a single `String` argument (see, for example, `Integer.valueOf(String)` and `java.util.UUID.fromString(String)`);

4. Have a registered implementation of `javax.ws.rs.ext.ParamConverterProvider` JAX-RS extension SPI that returns a `javax.ws.rs.ext.ParamConverter` instance capable of a "from string" conversion for the type. or

5. Be `List<T>`, `Set<T>` or `SortedSet<T>`, where `T` satisfies 2 or 3 above. The resulting collection is read-only.

Sometimes parameters may contain more than one value for the same name. If this is the case then types in 5) may be used to obtain all values.

If the @DefaultValue [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/DefaultValue.html] is not used in conjunction with @QueryParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/QueryParam.html] and the query parameter is not present in the request then value will be an empty collection for`List`, `Set` or `SortedSet`, `null` for other object types, and the Java-defined default for primitive types.

The @PathParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PathParam.html] and the other parameter-based annotations, @MatrixParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/MatrixParam.html], @HeaderParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/

javax/ws/rs/HeaderParam.html], @CookieParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/CookieParam.html], @FormParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/FormParam.html] obey the same rules as @QueryParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/QueryParam.html]. @MatrixParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/MatrixParam.html] extracts information from URL path segments. @HeaderParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/HeaderParam.html] extracts information from the HTTP headers. @CookieParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/CookieParam.html] extracts information from the cookies declared in cookie related HTTP headers.

@FormParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/FormParam.html] is slightly special because it extracts information from a request representation that is of the MIME media type `"application/x-www-form-urlencoded"` and conforms to the encoding specified by HTML forms, as described here. This parameter is very useful for extracting information that is POSTed by HTML forms, for example the following extracts the form parameter named "name" from the POSTed form data:

Example 3.10. Processing POSTed HTML form

```
1 @POST
2 @Consumes("application/x-www-form-urlencoded")
3 public void post(@FormParam("name") String name) {
4     // Store the message
5 }
```

If it is necessary to obtain a general map of parameter name to values then, for query and path parameters it is possible to do the following:

Example 3.11. Obtaining general map of URI path and/or query parameters

```
1 @GET
2 public String get(@Context UriInfo ui) {
3     MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
4     MultivaluedMap<String, String> pathParams = ui.getPathParameters();
5 }
```

For header and cookie parameters the following:

Example 3.12. Obtaining general map of header parameters

```
1 @GET
2 public String get(@Context HttpHeaders hh) {
3     MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
4     Map<String, Cookie> pathParams = hh.getCookies();
5 }
```

In general @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html] can be used to obtain contextual Java types related to the request or response.

Because form parameters (unlike others) are part of the message entity, it is possible to do the following:

Example 3.13. Obtaining general map of form parameters

```
1 @POST
2 @Consumes("application/x-www-form-urlencoded")
3 public void post(MultivaluedMap<String, String> formParams) {
4     // Store the message
5 }
```

I.e. you don't need to use the @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html] annotation.

Another kind of injection is the @BeanParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/BeanParam.html] which allows to inject the parameters described above into a single bean. A bean annotated with @BeanParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/BeanParam.html] containing any fields and appropriate `*param` annotation(like @PathParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PathParam.html]) will be initialized with corresponding request values in expected way as if these fields were in the resource class. Then instead of injecting request values like path param into a constructor parameters or class fields the @BeanParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/BeanParam.html] can be used to inject such a bean into a resource or resource method. The @BeanParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/BeanParam.html] is used this way to aggregate more request parameters into a single bean.

Example 3.14. Example of the bean which will be used as @BeanParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/BeanParam.html]

```
 1 public class MyBeanParam {
 2     @PathParam("p")
 3     private String pathParam;
 4
 5     @MatrixParam("m")
 6     @Encoded
 7     @DefaultValue("default")
 8     private String matrixParam;
 9
10     @HeaderParam("header")
11     private String headerParam;
12
13     private String queryParam;
14
15     public MyBeanParam(@QueryParam("q") String queryParam) {
16         this.queryParam = queryParam;
17     }
18
19     public String getPathParam() {
20         return pathParam;
21     }
22     ...
23 }
```

Example 3.15. Injection of MyBeanParam as a method parameter:

```
1 @POST
2 public void post(@BeanParam MyBeanParam beanParam, String entity) {
3     final String pathParam = beanParam.getPathParam(); // contains injected pa
4     ...
5 }
```

The example shows aggregation of injections @PathParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PathParam.html], @QueryParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/QueryParam.html] @MatrixParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/MatrixParam.html] and @HeaderParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/

HeaderParam.html] into one single bean. The rules for injections inside the bean are the same as described above for these injections. The @DefaultValue [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/DefaultValue.html] is used to define the default value for matrix parameter matrixParam. Also the @Encoded [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Encoded.html] annotation has the same behaviour as if it were used for injection in the resource method directly. Injecting the bean parameter into @Singleton resource class fields is not allowed (injections into method parameter must be used instead).

@BeanParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/BeanParam.html] can contain all parameters injections injections (@PathParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PathParam.html], @QueryParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/QueryParam.html], @MatrixParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/MatrixParam.html], @HeaderParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/HeaderParam.html], @CookieParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/CookieParam.html], @FormParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/FormParam.html]). More beans can be injected into one resource or method parameters even if they inject the same request values. For example the following is possible:

Example 3.16. Injection of more beans into one resource methods:

```
1 @POST
2 public void post(@BeanParam MyBeanParam beanParam, @BeanParam AnotherBean anot
3 String entity) {
4     // beanParam.getPathParam() == pathParam
5     ...
6 }
```

3.3. Sub-resources

@Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] may be used on classes and such classes are referred to as root resource classes. @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] may also be used on methods of root resource classes. This enables common functionality for a number of resources to be grouped together and potentially reused.

The first way @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] may be used is on resource methods and such methods are referred to as *sub-resource methods*. The following example shows the method signatures for a root resource class from the jmaki-backend sample:

Example 3.17. Sub-resource methods

```
 1 @Singleton
 2 @Path("/printers")
 3 public class PrintersResource {
 4
 5     @GET
 6     @Produces({"application/json", "application/xml"})
 7     public WebResourceList getMyResources() { ... }
 8
 9     @GET @Path("/list")
10     @Produces({"application/json", "application/xml"})
11     public WebResourceList getListOfPrinters() { ... }
12
13     @GET @Path("/jMakiTable")
14     @Produces("application/json")
15     public PrinterTableModel getTable() { ... }
16
17     @GET @Path("/jMakiTree")
18     @Produces("application/json")
19     public TreeModel getTree() { ... }
20
21     @GET @Path("/ids/{printerid}")
22     @Produces({"application/json", "application/xml"})
23     public Printer getPrinter(@PathParam("printerid") String printerId) { ...
24
25     @PUT @Path("/ids/{printerid}")
26     @Consumes({"application/json", "application/xml"})
27     public void putPrinter(@PathParam("printerid") String printerId, Printer p
28
29     @DELETE @Path("/ids/{printerid}")
30     public void deletePrinter(@PathParam("printerid") String printerId) { ...
31 }
```

If the path of the request URL is "printers" then the resource methods not annotated with @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] will be selected. If the request path of the request URL is "printers/list" then first the root resource class will be matched and then the sub-resource methods that match "list" will be selected, which in this case is the sub-resource method`getListOfPrinters`. So, in this example hierarchical matching on the path of the request URL is performed.

The second way @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] may be used is on methods **not** annotated with resource method designators such as @GET [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/GET.html] or @POST [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/POST.html]. Such methods are referred to as *sub-resource locators*. The following example shows the method signatures for a root resource class and a resource class from the optimistic-concurrency sample:

Example 3.18. Sub-resource locators

```
 1 @Path("/item")
 2 public class ItemResource {
 3     @Context UriInfo uriInfo;
 4
 5     @Path("content")
 6     public ItemContentResource getItemContentResource() {
 7         return new ItemContentResource();
 8     }
 9
10     @GET
11     @Produces("application/xml")
12         public Item get() { ... }
13     }
14 }
15
16 public class ItemContentResource {
17
18     @GET
19     public Response get() { ... }
20
21     @PUT
22     @Path("{version}")
23     public void put(@PathParam("version") int version,
24                     @Context HttpHeaders headers,
25                     byte[] in) {
26         ...
27     }
28 }
```

The root resource class `ItemResource` contains the sub-resource locator method `getItemContentResource` that returns a new resource class. If the path of the request URL is "item/content" then first of all the root resource will be matched, then the sub-resource locator will be matched and invoked, which returns an instance of the `ItemContentResource` resource class. Sub-resource locators enable reuse of resource classes. A method can be annotated with the @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] annotation with empty path (`@Path("/")` or `@Path("")`) which means that the sub resource locator is matched for the path of the enclosing resource (without sub-resource path).

Example 3.19. Sub-resource locators with empty path

```
 1 @Path("/item")
 2 public class ItemResource {
 3
 4     @Path("/")
 5     public ItemContentResource getItemContentResource() {
 6         return new ItemContentResource();
 7     }
 8 }
```

In the example above the sub-resource locator method `getItemContentResource` is matched for example for request path "/item/locator" or even for only "/item".

In addition the processing of resource classes returned by sub-resource locators is performed at runtime thus it is possible to support polymorphism. A sub-resource locator may return different sub-types depending on the request (for example a sub-resource locator could return different sub-types dependent on the role of the principle that is authenticated). So for example the following sub resource locator is valid:

Example 3.20. Sub-resource locators returning sub-type

```
1 @Path("/item")
2 public class ItemResource {
3
4     @Path("/")
5     public Object getItemContentResource() {
6         return new AnyResource();
7     }
8 }
```

Note that the runtime will not manage the life-cycle or perform any field injection onto instances returned from sub-resource locator methods. This is because the runtime does not know what the life-cycle of the instance is. If it is required that the runtime manages the sub-resources as standard resources the `Class` should be returned as shown in the following example:

Example 3.21. Sub-resource locators created from classes

```
 1 import javax.inject.Singleton;
 2
 3 @Path("/item")
 4 public class ItemResource {
 5     @Path("content")
 6     public Class<ItemContentSingletonResource> getItemContentResource() {
 7         return ItemContentSingletonResource.class;
 8     }
 9 }
10
11 @Singleton
12 public class ItemContentSingletonResource {
13     // this class is managed in the singleton life cycle
14 }
```

JAX-RS resources are managed in per-request scope by default which means that new resource is created for each request. In this example the `javax.inject.Singleton` annotation says that the resource will be managed as singleton and not in request scope. The sub-resource locator method returns a class which means that the runtime will managed the resource instance and its life-cycle. If the method would return instance instead, the `Singleton` annotation would have no effect and the returned instance would be used.

The sub resource locator can also return a *programmatic resource model*. See resource builder section for information of how the programmatic resource model is constructed. The following example shows very simple resource returned from the sub-resource locator method.

Example 3.22. Sub-resource locators returning resource model

```
 1 import org.glassfish.jersey.server.model.Resource;
 2
 3 @Path("/item")
 4 public class ItemResource {
 5
 6     @Path("content")
 7     public Resource getItemContentResource() {
 8         return Resource.from(ItemContentSingletonResource.class);
 9     }
10 }
```

The code above has exactly the same effect as previous example. `Resource` is a resource simple resource constructed from `ItemContentSingletonResource`. More complex programmatic resource can be returned as long they are valid resources.

3.4. Life-cycle of Root Resource Classes

By default the life-cycle of root resource classes is per-request which, namely that a new instance of a root resource class is created every time the request URI path matches the root resource. This makes for a very natural programming model where constructors and fields can be utilized (as in the previous section showing the constructor of the `SparklinesResource` class) without concern for multiple concurrent requests to the same resource.

In general this is unlikely to be a cause of performance issues. Class construction and garbage collection of JVMs has vastly improved over the years and many objects will be created and discarded to serve and process the HTTP request and return the HTTP response.

Instances of singleton root resource classes can be declared by an instance of Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html].

Jersey supports two further life-cycles using Jersey specific annotations.

Table 3.1. Resource scopes

Scope	Annotation	Annotation full class name	Description
Request scope	@RequestScoped (or none)	org.glassfish.jersey.process.internal.RequestScoped	*Default lifecycle* (applied when no annotation is present). In this scope the resource instance is created for each new request and used for processing of this request. If the resource is used more than one time in the request processing, always the same instance will be used. This can happen when a resource is a sub resource is returned more times during the matching. In this situation only on instance will server the requests.
Per-lookup scope	@PerLookup	org.glassfish.jersey.process.internal.PerLookup	In this scope a new resource instance is created every time it is needed for the processing even it handles the same request.
Singleton	@Singleton	javax.inject.Singleton	In this scope there is only one instance per jax-rs application. Singleton resource can be either annotated with @Singleton and its

Scope	Annotation	Annotation full class name	Description
			class can be registered using the instance of Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html]. You can also create singletons by registering singleton instances into Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html].

3.5. Rules of Injection

Previous sections have presented examples of annotated types, mostly annotated method parameters but also annotated fields of a class, for the injection of values onto those types.

This section presents the rules of injection of values on annotated types. Injection can be performed on fields, constructor parameters, resource/sub-resource/sub-resource locator method parameters and bean setter methods. The following presents an example of all such injection cases:

Example 3.23. Injection

```
@Path("id: \d+")
public class InjectedResource {
    // Injection onto field
    @DefaultValue("q") @QueryParam("p")
    private String p;

    // Injection onto constructor parameter
    public InjectedResource(@PathParam("id") int id) { ... }

    // Injection onto resource method parameter
    @GET
    public String get(@Context UriInfo ui) { ... }

    // Injection onto sub-resource resource method parameter
    @Path("sub-id")
    @GET
    public String get(@PathParam("sub-id") String id) { ... }

    // Injection onto sub-resource locator method parameter
    @Path("sub-id")
    public SubResource getSubResource(@PathParam("sub-id") String id) { ... }

    // Injection using bean setter method
    @HeaderParam("X-header")
    public void setHeader(String header) { ... }
}
```

There are some restrictions when injecting on to resource classes with a life-cycle of singleton scope. In such cases the class fields or constructor parameters cannot be injected with request specific parameters. So, for example the following is not allowed.

Example 3.24. Wrong injection into a singleton scope

```
 1 @Path("resource")
 2 @Singleton
 3 public static class MySingletonResource {
 4
 5     @QueryParam("query")
 6     String param; // WRONG: initialization of application will fail as you can
 7                   // inject request specific parameters into a singleton resou
 8
 9     @GET
10     public String get() {
11         return "query param: " + param;
12     }
13 }
```

The example above will cause validation failure during application initialization as singleton resources cannot inject request specific parameters. The same example would fail if the query parameter would be injected into constructor parameter of such a singleton. In other words, if you wish one resource instance to server more requests (in the same time) it cannot be bound to a specific request parameter.

The exception exists for specific request objects which can injected even into constructor or class fields. For these objects the runtime will inject proxies which are able to simultaneously server more request. These request objects are `HttpHeaders`, `Request`, `UriInfo`, `SecurityContext`. These proxies can be injected using the @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html] annotation. The following example shows injection of proxies into the singleton resource class.

Example 3.25. Injection of proxies into singleton

```
 1 @Path("resource")
 2 @Singleton
 3 public static class MySingletonResource {
 4     @Context
 5     Request request; // this is ok: the proxy of Request will be injected into
 6
 7     public MySingletonResource(@Context SecurityContext securityContext) {
 8         // this is ok too: the proxy of SecurityContext will be injected
 9     }
10
11     @GET
12     public String get() {
13         return "query param: " + param;
14     }
15 }
```

To summarize the injection can be done into the following constructs:

Table 3.2. Overview of injection types

Java construct	Description
Class fields	Inject value directly into the field of the class. The field can be private and must not be final. Cannot be used in Singleton scope except proxiable types mentioned above.

Java construct	Description
Constructor parameters	The constructor will be invoked with injected values. If more constructors exists the one with the most injectable parameters will be invoked. Cannot be used in Singleton scope except proxiable types mentioned above.
Resource methods	The resource methods (these annotated with @GET, @POST, ...) can contain parameters that can be injected when the resource method is executed. Can be used in any scope.
Sub resource locators	The sub resource locators (methods annotated with @Path but not @GET, @POST, ...) can contain parameters that can be injected when the resource method is executed. Can be used in any scope.
Setter methods	Instead of injecting values directly into field the value can be injected into the setter method which will initialize the field. This injection can be used only with @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html] annotation. This means it cannot be used for example for injecting of query params but it can be used for injections of request. The setters will be called after the object creation and only once. The name of the method does not necessary have a setter pattern. Cannot be used in Singleton scope except proxiable types mentioned above.

The following example shows all possible java constructs into which the values can be injected.

Example 3.26. Example of possible injections

```
 1 @Path("resource")
 2 public static class SummaryOfInjectionsResource {
 3     @QueryParam("query")
 4     String param; // injection into a class field
 5
 6
 7     @GET
 8     public String get(@QueryParam("query") String methodQueryParam) {
 9         // injection into a resource method parameter
10         return "query param: " + param;
11     }
12
13     @Path("sub-resource-locator")
14     public Class<SubResource> subResourceLocator(@QueryParam("query") String s
15         // injection into a sub resource locator parameter
16         return SubResource.class;
17     }
18
19     public SummaryOfInjectionsResource(@QueryParam("query") String constructor
20         // injection into a constructor parameter
21     }
22
23
24     @Context
25     public void setRequest(Request request) {
26         // injection into a setter method
27         System.out.println(request != null);
28     }
29 }
30
31 public static class SubResource {
32     @GET
33     public String get() {
34         return "sub resource";
35     }
36 }
```

The @FormParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/FormParam.html] annotation is special and may only be utilized on resource and sub-resource methods. This is because it extracts information from a request entity.

3.6. Use of @Context

Previous sections have introduced the use of @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html]. Chapter 5 of the JAX-RS specification presents all the standard JAX-RS Java types that may be used with @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html].

When deploying a JAX-RS application using servlet then ServletConfig [http://docs.oracle.com/javaee/5/api/javax/servlet/ServletConfig.html], ServletContext [http://docs.oracle.com/javaee/5/api/javax/servlet/ServletContext.html], HttpServletRequest [http://docs.oracle.com/javaee/5/api/javax/servlet/http/HttpServletRequest.html] and HttpServletResponse [http://docs.oracle.com/javaee/5/

api/javax/servlet/http/HttpServletResponse.html] are available using @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html].

3.7. Programmatic resource model

Resources can be constructed from classes or instances but also can be constructed from a programmatic resource model. Every resource created from from resource classes can also be constructed using the programmatic resource builder api. See resource builder section for more information.

Chapter 4. Deploying a RESTful Web Service

JAX-RS provides a deployment agnostic abstract class Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] for declaring root resource and provider classes, and root resource and provider singleton instances. A Web service may extend this class to declare root resource and provider classes. For example,

Example 4.1. Deployment agnostic application model

```
1 public class MyApplication extends Application {
2     @Override
3     public Set<Class<?>> getClasses() {
4         Set<Class<?>> s = new HashSet<Class<?>>();
5         s.add(HelloWorldResource.class);
6         return s;
7     }
8 }
```

Alternatively it is possible to reuse one of Jersey's implementations that scans for root resource and provider classes given a classpath or a set of package names. Such classes are automatically added to the set of classes that are returned by`getClasses`. For example, the following scans for root resource and provider classes in packages "org.foo.rest", "org.bar.rest" and in any sub-packages of those two:

Example 4.2. Reusing Jersey implementation in your custom application model

```
1 public class MyApplication extends ResourceConfig {
2     public MyApplication() {
3         packages("org.foo.rest;org.bar.rest");
4     }
5 }
```

There are multiple deployment options for the class that implements Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] interface in the Servlet 3.0 container. For simple deployments, no `web.xml` is needed at all. Instead, an @ApplicationPath [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ApplicationPath.html] annotation can be used to annotate the user defined application class and specify the the base resource URI of all application resources:

Example 4.3. Deployment of a JAX-RS application using `@ApplicationPath` with Servlet 3.0

```
1 @ApplicationPath("resources")
2 public class MyApplication extends ResourceConfig {
3     public MyApplication() {
4         packages("org.foo.rest;org.bar.rest");
5     }
6     ...
7 }
```

Please note that there is more which can be set or called during execution of ResourceConfig descendants constructor, see ResourceConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ResourceConfig.html] javadoc for more details.

You also need to set maven-war-plugin attribute failOnMissingWebXml [http://maven.apache.org/plugins/maven-war-plugin/war-mojo.html#failOnMissingWebXml] to false in pom.xml when building .war without web.xml file using maven:

Example 4.4. Configuration of maven-war-plugin in `pom.xml` with Servlet 3.0

```
<plugins>
    ...
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.3</version>
        <configuration>
            <failOnMissingWebXml>false</failOnMissingWebXml>
        </configuration>
    </plugin>
    ...
</plugins>
```

Another deployment option is to declare JAX-RS application details in the`web.xml`. This is usually suitable in case of more complex deployments, e.g. when security model needs to be properly defined or when additional initialization parameters have to be passed to Jersey runtime. JAX-RS 1.1 specifies that a fully qualified name of the class that implements Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] may be declared in the `<servlet-name>` element of the JAX-RS application's `web.xml`. This is supported in a Web container implementing Servlet 3.0 as follows:

Example 4.5. Deployment of a JAX-RS application using `web.xml` with Servlet 3.0

```
<web-app>
    <servlet>
        <servlet-name>org.foo.rest.MyApplication</servlet-name>
    </servlet>
    ...
    <servlet-mapping>
        <servlet-name>org.foo.rest.MyApplication</servlet-name>
        <url-pattern>/resources</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

Note that the `<servlet-class>` element is omitted from the servlet declaration. This is a correct declaration utilizing the Servlet 3.0 extension mechanism. Also note that `<servlet-mapping>` is used to define the base resource URI.

When running in a Servlet 2.x then instead it is necessary to declare the Jersey specific servlet and pass the Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] implementation class name as one of the servlet's `init-param` entries:

Example 4.6. Deployment of your application using Jersey specific servlet

```
 1 <web-app>
 2     <servlet>
 3         <servlet-name>Jersey Web Application</servlet-name>
 4         <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-
 5         <init-param>
 6             <param-name>javax.ws.rs.Application</param-name>
 7             <param-value>org.foo.rest.MyApplication</param-value>
 8         </init-param>
 9         ...
10     </servlet>
11     ...
12 </web-app>
```

If there is no configuration to be set and deployed application consists only from resources and providers stored in particular packages, Jersey can scan them and register automatically:

Example 4.7. Using Jersey specific servlet without an application model instance

```
 1 <web-app>
 2     <servlet>
 3         <servlet-name>Jersey Web Application</servlet-name>
 4         <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-
 5         <init-param>
 6             <param-name>jersey.config.server.provider.packages</param-name>
 7             <param-value>org.foo.rest;org.bar.rest</param-value>
 8         </init-param>
 9         ...
10     </servlet>
11     ...
12 </web-app>
```

JAX-RS also provides the ability to obtain a container specific artifact from an Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] instance. For example, Jersey supports using Grizzly [http://grizzly.java.net/] as follows:

```
HttpHandler endpoint = RuntimeDelegate.getInstance().createEndpoint(new MyApplicat
```

Jersey also provides Grizzly [http://grizzly.java.net/] helper classes to deploy the HttpHandler instance at a base URL for in-process deployment.

The Jersey samples provide many examples of Servlet-based and Grizzly-in-process-based deployments.

4.1. Auto-Discoverable Features

For a few modules provided by Jersey there is no need to explicitly register their Feature [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Feature.html]s as these `Feature`s are discovered and registered in the Configuration [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Configuration.html] (on client/server) automatically by Jersey when the modules implementing these features are present on the classpath during the an application deployment. The modules that are automatically discovered include:

- `jersey-media-moxy` (JSON part)

- `jersey-media-json-processing`
- `jersey-bean-validation`

Besides these modules there are also few features/providers present in `jersey-server` module that are discovered by this mechanism and their availability is affected by Jersey auto-discovery support configuration (see Section 4.1.1, “Configuring the Feature Auto-discovery mechanism”):

- WadlFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/wadl/WadlFeature.html] - enables WADL processing.
- UriConnegFilter [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/filter/UriConnegFilter.html] - a URI-based content negotiation filter.

Note

Auto discovery functionality is in Jersey supported by implementing an internal SPI `AutoDiscoverable` interface. This interface is not public at the moment, so be careful when using it.

4.1.1. Configuring the Feature Auto-discovery mechanism

The auto-discovery of features in Jersey that is enabled by default can be disabled by using special (common/server/client) properties:

Common auto discovery properties

- CommonProperties.FEATURE_AUTO_DISCOVERY_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/CommonProperties.html#FEATURE_AUTO_DISCOVERY_DISABLE]

  Disables auto discovery globally on client/server.

- CommonProperties.JSON_PROCESSING_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/CommonProperties.html#JSON_PROCESSING_FEATURE_DISABLE]

  Disables configuration of Json Processing (JSR-353) feature.

- CommonProperties.MOXY_JSON_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/CommonProperties.html#MOXY_JSON_FEATURE_DISABLE]

  Disables configuration of MOXy Json feature.

For each of these properties there is a client/server counter-part that only disables the feature on the client or server respectively (see ClientProperties [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html]/ServerProperties [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html]). Each of these client/server specific auto-discovery related properties overrides the value of the related common property (if set).

Note

In case an auto-discoverable feature is disabled then all the featured, components and/or properties, registered with the feature by default using the auto-discovery mechanism have to be registered manually.

Chapter 5. Client API

This section introduces the JAX-RS Client API, which is a fluent Java based API for communication with RESTful Web services. This standard API that is also part of Java EE 7 is designed to make it very easy to consume a Web service exposed via HTTP protocol and enables developers to concisely and efficiently implement portable client-side solutions that leverage existing and well established client-side HTTP connector implementations.

The JAX-RS client API can be utilized to consume any Web service exposed on top of a HTTP protocol or it's extension (e.g. WebDAV), and is not restricted to services implemented using JAX-RS. Yet, developers familiar with JAX-RS should find the client API complementary to their services, especially if the client API is utilized by those services themselves, or to test those services. The JAX-RS client API finds inspiration in the proprietary Jersey 1.x Client API and developers familiar with the Jersey 1.x Client API should find it easy to understand all the concepts introduced in the new JAX-RS Client API.

The goals of the client API are threefold:

1. Encapsulate a key constraint of the REST architectural style, namely the Uniform Interface Constraint and associated data elements, as client-side Java artifacts;

2. Make it as easy to consume RESTful Web services exposed over HTTP, same as the JAX-RS server-side API makes it easy to develop RESTful Web services; and

3. Share common concepts and extensibility points of the JAX-RS API between the server and the client side programming models.

As an extension to the standard JAX-RS Client API, the Jersey Client API supports a pluggable architecture to enable the use of different underlying HTTP client Connector [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/spi/Connector.html] implementations. Several such implementations are currently provided with Jersey. We have a default client connector using `Http(s)URLConnection` supplied with the JDK as well as connector implementations based on Apache HTTP Client, and Grizzly Asynchronous Client.

5.1. Uniform Interface Constraint

The uniform interface constraint bounds the architecture of RESTful Web services so that a client, such as a browser, can utilize the same interface to communicate with any service. This is a very powerful concept in software engineering that makes Web-based search engines and service mash-ups possible. It induces properties such as:

1. simplicity, the architecture is easier to understand and maintain; and

2. evolvability or loose coupling, clients and services can evolve over time perhaps in new and unexpected ways, while retaining backwards compatibility.

Further constraints are required:

1. every resource is identified by a URI;

2. a client interacts with the resource via HTTP requests and responses using a fixed set of HTTP methods;

3. one or more representations can be returned and are identified by media types; and

4. the contents of which can link to further resources.

The above process repeated over and again should be familiar to anyone who has used a browser to fill in HTML forms and follow links. That same process is applicable to non-browser based clients.

Many existing Java-based client APIs, such as the Apache HTTP client API or `HttpUrlConnection` supplied with the JDK place too much focus on the Client-Server constraint for the exchanges of request and responses rather than a resource, identified by a URI, and the use of a fixed set of HTTP methods.

A resource in the JAX-RS client API is an instance of the Java class WebTarget [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/WebTarget.html]. and encapsulates an URI. The fixed set of HTTP methods can be invoked based on the `WebTarget`. The representations are Java types, instances of which, may contain links that new instances of `WebTarget` may be created from.

5.2. Ease of use and reusing JAX-RS artifacts

Since a JAX-RS component is represented as an annotated Java type, it makes it easy to configure, pass around and inject in ways that are not so intuitive or possible with other client-side APIs. The Jersey Client API reuses many aspects of the JAX-RS and the Jersey implementation such as:

1. URI building using UriBuilder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriBuilder.html] and UriTemplate [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/uri/UriTemplate.html] to safely build URIs;

2. Built-in support for Java types of representations such as `byte[]`, `String`, `Number`, `Boolean`, `Character`, `InputStream`, `java.io.Reader`, `File`, `DataSource`, JAXB beans as well as additional Jersey-specific JSON and Multi Part [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/multipart/package-info.html] support.

3. Using the fluent builder-style API pattern to make it easier to construct requests.

Some APIs, like the Apache HTTP Client or HttpURLConnection [http://docs.oracle.com/javase/6/docs/api/java/net/HttpURLConnection.html] can be rather hard to use and/or require too much code to do something relatively simple, especially when the client needs to understand different payload representations. This is why the Jersey implementation of JAX-RS Client API provides support for wrapping `HttpUrlConnection` and the Apache HTTP client. Thus it is possible to get the benefits of the established JAX-RS implementations and features while getting the ease of use benefit of the simple design of the JAX-RS client API. For example, with a low-level HTTP client library, sending a POST request with a bunch of typed HTML form parameters and receiving a response de-serialized into a JAXB bean is not straightforward at all. With the new JAX-RS Client API supported by Jersey this task is very easy:

Example 5.1. POST request with form parameters

```
 1 Client client = ClientBuilder.newClient();
 2 WebTarget target = client.target("http://localhost:9998").path("resource");
 3
 4 Form form = new Form();
 5 form.param("x", "foo");
 6 form.param("y", "bar");
 7
 8 MyJAXBBean bean =
 9 target.request(MediaType.APPLICATION_JSON_TYPE)
10     .post(Entity.entity(form,MediaType.APPLICATION_FORM_URLENCODED_TYPE),
11         MyJAXBBean.class);
```

In the Example 5.1, “POST request with form parameters” a new `WebTarget` instance is created using a new Client [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/Client.html] instance first, next a Form [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Form.html] instance is created with two form parameters. Once ready, the `Form` instance is `POST`ed to the target resource. First, the

acceptable media type is specified in the `request(...)` method. Then in the `post(...)` method, a call to a static method on JAX-RS Entity [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/Entity.html] is made to construct the request entity instance and attach the proper content media type to the form entity that is being sent. The second parameter in the `post(...)` method specifies the Java type of the response entity that should be returned from the method in case of a successful response. In this case an instance of JAXB bean is requested to be returned on success. The Jersey client API takes care of selecting the proper MessageBodyWriter<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyWriter.html] for the serialization of the `Form` instance, invoking the `POST` request and producing and de-serialization of the response message payload into an instance of a JAXB bean using a proper MessageBodyReader<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyReader.html].

If the code above had to be written using `HttpUrlConnection`, the developer would have to write custom code to serialize the form data that are sent within the POST request and de-serialize the response input stream into a JAXB bean. Additionally, more code would have to be written to make it easy to reuse the logic when communicating with the same resource `"http://localhost:8080/resource"` that is represented by the JAX-RS `WebTarget` instance in our example.

5.3. Overview of the Client API

5.3.1. Getting started with the client API

Refer to the dependencies for details on the dependencies when using the Jersey JAX-RS Client support.

You may also want to use a custom Connector [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/spi/Connector.html] implementation. In such case you would need to include additional dependencies on the module(s) containing the custom client connector that you want to use. See section "Configuring custom Connectors" about how to use and configure a custom Jersey client transport `Connector`.

5.3.2. Creating and configuring a Client instance

JAX-RS Client API is a designed to allow fluent programming model. This means, a construction of a `Client` instance, from which a `WebTarget` is created, from which a request Invocation [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/Invocation.html] is built and invoked can be chained in a single "flow" of invocations. The individual steps of the flow will be shown in the following sections. To utilize the client API it is first necessary to build an instance of a Client [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/Client.html] using one of the static ClientBuilder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientBuilder.html] factory methods. Here's the most simple example:

```
Client client = ClientBuilder.newClient();
```

The `ClientBuilder` is a JAX-RS API used to create new instances of `Client`. In a slightly more advanced scenarios, `ClientBuilder` can be used to configure additional client instance properties, such as a SSL transport settings, if needed (see ??? below).

A `Client` instance can be configured during creation by passing a ClientConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientConfig.html] to the `newClient(Configurable)` `ClientBuilder` factory method. ClientConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientConfig.html] implements Configurable [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Configurable.html] and therefore it offers methods to register providers (e.g. features or individual entity providers, filters or interceptors) and setup properties. The following code shows a registration of custom client filters:

```
1 ClientConfig clientConfig = new ClientConfig();
2 clientConfig.register(MyClientResponseFilter.class);
3 clientConfig.register(new AnotherClientFilter());
4 Client client = ClientBuilder.newClient(clientConfig);
```

In the example, filters are registered using the `ClientConfig.register(...)` method. There are multiple overloaded versions of the method that support registration of feature and provider classes or instances. Once a `ClientConfig` instance is configured, it can be passed to the `ClientBuilder` to create a pre-configured `Client` instance.

Note that the Jersey `ClientConfig` supports the fluent API model of Configurable [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Configurable.html]. With that the code that configures a new client instance can be also written using a more compact style as shown below.

```
1
2 Client client = ClientBuilder.newClient(new ClientConfig()
3         .register(MyClientResponseFilter.class)
4         .register(new AnotherClientFilter());
```

The ability to leverage this compact pattern is inherent to all JAX-RS and Jersey Client API components.

Since `Client` implements Configurable [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Configurable.html] interface too, it can be configured further even after it has been created. Important is to mention that any configuration change done on a `Client` instance will not influence the ClientConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientConfig.html] instance that was used to provide the initial `Client` instance configuration at the instance creation time. The next piece of code shows a configuration of an existing `Client` instance.

```
1 client.register(ThirdClientFilter.class);
```

Similarly to earlier examples, since `Client.register(...)` method supports the fluent API style, multiple client instance configuration calls can be chained:

```
1 client.register(FilterA.class)
2       .register(new FilterB())
3       .property("my-property", true);
```

To get the current configuration of the `Client` instance a `getConfiguration()` method can be used.

```
1 ClientConfig clientConfig = new ClientConfig();
2 clientConfig.register(MyClientResponseFilter.class);
3 clientConfig.register(new AnotherClientFilter());
4 Client client = ClientBuilder.newClient(clientConfig);
5 client.register(ThirdClientFilter.class);
6 Configuration newConfiguration = client.getConfiguration();
```

In the code, an additional `MyClientResponseFilter` class and `AnotherClientFilter` instance are registered in the `clientConfig`. The `clientConfig` is then used to construct a new `Client` instance. The `ThirdClientFilter` is added separately to the constructed `Client` instance. This does not influence the configuration represented by the original `clientConfig`. In the last step a `newConfiguration` is retrieved from the `client`. This configuration contains all three registered filters while the original `clientConfig` instance still contains only two filters. Unlike `clientConfig` created separately, the `newConfiguration` retrieved from the `client` instance represents a live client configuration view. Any additional configuration changes made to the `client` instance are also reflected in the `newConfiguration`. So, `newConfiguration` is really a view of the `client` configuration and not a configuration state copy. These principles are important in the client API and will be used in

the following sections too. For example, you can construct a common base configuration for all clients (in our case it would be `clientConfig`) and then reuse this common configuration instance to configure multiple `client` instances that can be further specialized. Similarly, you can use an existing `client` instance configuration to configure another client instance without having to worry about any side effects in the original `client` instance.

5.3.3. Targeting a web resource

Once you have a `Client` instance you can create a `WebTarget` from it.

```
1 WebTarget webTarget = client.target("http://example.com/rest");
```

A `Client` contains several `target(...)` methods that allow for creation of `WebTarget` instance. In this case we're using `target(String uri)` version. The `uri` passed to the method as a `String` is the URI of the targeted web resource. In more complex scenarios it could be the context root URI of the whole RESTful application, from which `WebTarget` instances representing individual resource targets can be derived and individually configured. This is possible, because JAX-RS `WebTarget` also implements `Configurable`:

```
1 WebTarget webTarget = client.target("http://example.com/rest");
2 webTarget.register(FilterForExampleCom.class);
```

The configuration principles used in JAX-RS client API apply to `WebTarget` as well. Each `WebTarget` instance inherits a configuration from it's parent (either a client or another web target) and can be further custom-configured without affecting the configuration of the parent component. In this case, the `FilterForExampleCom` will be registered only in the `webTarget` and not in `client`. So, the `client` can still be used to create new `WebTarget` instances pointing at other URIs using just the common client configuration, which `FilterForExampleCom` filter is not part of.

5.3.4. Identifying resource on WebTarget

Let's assume we have a `webTarget` pointing at `"http://example.com/rest"` URI that represents a context root of a RESTful application and there is a resource exposed on the URI `"http://example.com/rest/resource"`. As already mentioned, a `WebTarget` instance can be used to derive other web targets. Use the following code to define a path to the resource.

```
1 WebTarget resourceWebTarget = webTarget.path("resource");
```

The `resourceWebTarget` now points to the resource on URI `"http://example.com/rest/resource"`. Again if we configure the `resourceWebTarget` with a filter specific to the `resource`, it will not influence the original `webTarget` instance. However, the filter `FilterForExampleCom` registration will still be inherited by the `resourceWebTarget` as it has been created from `webTarget`. This mechanism allows you to share the common configuration of related resources (typically hosted under the same URI root, in our case represented by the `webTarget` instance), while allowing for further configuration specialization based on the specific requirements of each individual resource. The same configuration principles of inheritance (to allow common config propagation) and decoupling (to allow individual config customization) applies to all components in JAX-RS Client API discussed below.

Let's say there is a sub resource on the path `"http://example.com/rest/resource/helloworld"`. You can derive a `WebTarget` for this resource simply by:

```
1 WebTarget helloworldWebTarget = resourceWebTarget.path("helloworld");
```

Let's assume that the `helloworld` resource accepts a query param for `GET` requests which defines the greeting message. The next code snippet shows a code that creates a new `WebTarget` with the query param defined.

```
1 WebTarget helloworldWebTargetWithQueryParam =
2         helloworldWebTarget.queryParam("greeting", "Hi World!");
```

Please note that apart from methods that can derive new `WebTarget` instance based on a URI path or query parameters, the JAX-RS `WebTarget` API contains also methods for working with matrix parameters too.

5.3.5. Invoking a HTTP request

Let's now focus on invoking a `GET` HTTP request on the created web targets. To start building a new HTTP request invocation, we need to create a new Invocation.Builder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/Invocation.Builder.html].

```
1 Invocation.Builder invocationBuilder =
2         helloworldWebTargetWithQueryParam.request(MediaType.TEXT_PLAIN_TYPE);
3 invocationBuilder.header("some-header", "true");
```

A new invocation builder instance is created using one of the `request(...)` methods that are available on `WebTarget`. A couple of these methods accept parameters that let you define the media type of the representation requested to be returned from the resource. Here we are saying that we request a `"text/plain"` type. This tells Jersey to add a `Accept: text/plain` HTTP header to our request.

The `invocationBuilder` is used to setup request specific parameters. Here we can setup headers for the request or for example cookie parameters. In our example we set up a `"some-header"` header to value `true`.

Once finished with request customizations, it's time to invoke the request. We have two options now. We can use the `Invocation.Builder` to build a generic Invocation [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/Invocation.html] instance that will be invoked some time later. Using `Invocation` we will be able to e.g. set additional request properties which are properties in a batch of several requests and use the generic JAX-RS `Invocation` API to invoke the batch of requests without actually knowing all the details (such as request HTTP method, configuration etc.). Any properties set on an invocation instance can be read during the request processing. For example, in a custom ClientRequestFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/RequestFilter.html] you can call `getProperty()` method on the supplied ClientRequestContext [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientRequestContext.html] to read a request property. Note that these request properties are different from the configuration properties set on `Configurable`. As mentioned earlier, an `Invocation` instance provides generic invocation API to invoke the HTTP request it represents either synchronously or asynchronously. See the Chapter 10, *Asynchronous Services and Clients* for more information on asynchronous invocations.

In case you do not want to do any batch processing on your HTTP request invocations prior to invoking them, there is another, more convenient approach that you can use to invoke your requests directly from an `Invocation.Builder` instance. This approach is demonstrated in the next Java code listing.

```
1 Response response = invocationBuilder.get();
```

While short, the code in the example performs multiple actions. First, it will build the the request from the `invocationBuilder`. The URI of request will be `http://example.com/rest/resource/helloworld?greeting="Hi%20World!"` and the request will contain `some-header: true` and `Accept: text/plain` headers. The request will then pass trough all configured request filters (`AnotherClientFilter`, `ThirdClientFilter` and `FilterForExampleCom`). Once processed by the filters, the request will be sent to the remote resource. Let's say the resource then returns an HTTP 200 message with a plain text response content that contains the value sent in the request `greeting` query parameter. Now we can observe the returned response:

```
1 System.out.println(response.getStatus());
2 System.out.println(response.readEntity(String.class));
```

which will produce the following output to the console:

```
200
Hi World!
```

As we can see, the request was successfully processed (code 200) and returned an entity (representation) is `"Hi World!"`. Note that since ve have configured a `MyClientResponseFilter` in the resource target, when `response.readEntity(String.class)` gets called, the response returned from the remote endpoint is passed through the response filter chain (including the `MyClientResponseFilter`) and entity interceptor chain and at last a proper MessageBodyReader<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyReader.html] is located to read the response content bytes from the response stream into a Java `String` instance. Check Chapter 9, *Filters and Interceptors* to lear more about request and response filters and entity interceptors.

Imagine now that you would like to invoke a `POST` request but without any query parameters. You would just use the `helloworldWebTarget` instance created earlier and call the `post()` instead of `get()`.

```
1 Response postResponse =
2         helloworldWebTarget.request(MediaType.TEXT_PLAIN_TYPE)
3                 .post(Entity.entity("A string entity to be POSTed", MediaType.
```

5.3.6. Example summary

The following code puts together the pieces used in the earlier examples.

Example 5.2. Using JAX-RS Client API

```
 1 ClientConfig clientConfig = new ClientConfig();
 2 clientConfig.register(MyClientResponseFilter.class);
 3 clientConfig.register(new AnotherClientFilter());
 4
 5 Client client = ClientBuilder.newClient(clientConfig);
 6 client.register(ThirdClientFilter.class);
 7
 8 WebTarget webTarget = client.target("http://example.com/rest");
 9 webTarget.register(FilterForExampleCom.class);
10 WebTarget resourceWebTarget = webTarget.path("resource");
11 WebTarget helloworldWebTarget = resourceWebTarget.path("helloworld");
12 WebTarget helloworldWebTargetWithQueryParam =
13         helloworldWebTarget.queryParam("greeting", "Hi World!");
14
15 Invocation.Builder invocationBuilder =
16         helloworldWebTargetWithQueryParam.request(MediaType.TEXT_PLAIN_TYPE);
17 invocationBuilder.header("some-header", "true");
18
19 Response response = invocationBuilder.get();
20 System.out.println(response.getStatus());
21 System.out.println(response.readEntity(String.class));
```

Now we can try to leverage the fluent API style to write this code in a more compact way.

Example 5.3. Using JAX-RS Client API fluently

```
Client client = ClientBuilder.newClient(new ClientConfig()
            .register(MyClientResponseFilter.class)
            .register(new AnotherClientFilter()));

String entity = client.target("http://example.com/rest")
            .register(FilterForExampleCom.class)
            .path("resource/helloworld")
            .queryParam("greeting", "Hi World!")
            .request(MediaType.TEXT_PLAIN_TYPE)
            .header("some-header", "true")
            .get(String.class);
```

The code above does the same thing except it skips the generic `Response` processing and directly requests an entity in the last `get(String.class)` method call. This shortcut method let's you specify that (in case the response was returned successfully with a HTTP 2xx status code) the response entity should be returned as Java `String` type. This compact example demonstrates another advantage of the JAX-RS client API. The fluency of JAX-RS Client API is convenient especially with simple use cases. Here is another a very simple GET request returning a String representation (entity):

```
String responseEntity = ClientBuilder.newClient()
            .target("http://example.com").path("resource/rest")
                        .request().get(String.class);
```

5.4. Java instances and types for representations

All the Java types and representations supported by default on the Jersey server side for requests and responses are also supported on the client side. For example, to process a response entity (or representation) as a stream of bytes use InputStream as follows:

```
InputStream in = response.readEntity(InputStream.class);

... // Read from the stream

in.close();
```

Note that it is important to close the stream after processing so that resources are freed up.

To `POST` a file use a `File` instance as follows:

```
File f = ...

...

webTarget.request().post(Entity.entity(f, MediaType.TEXT_PLAIN_TYPE));
```

5.4.1. Adding support for new representations

The support for new application-defined representations as Java types requires the implementation of the same JAX-RS entity provider extension interfaces as for the server side JAX-RS

API, namely MessageBodyReader<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyReader.html] and MessageBodyWriter<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyWriter.html] respectively, for request and response entities (or inbound and outbound representations).

Classes or implementations of the provider-based interfaces need to be registered as providers within the JAX-RS or Jersey Client API components that implement `Configurable` contract (`ClientBuilder`, `Client`, `WebTarget` or `ClientConfig`), as was shown in the earlier sections. Some media types are provided in the form of JAX-RS Feature [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Feature.html] a concept that allows the extension providers to group together multiple different extension providers and/or configuration properties in order to simplify the registration and configuration of the provided feature by the end users. For example, MoxyJsonFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/moxy/json/MoxyJsonFeature.html] can be register to enable and configure JSON binding support via MOXy library.

5.5. Client Transport Connectors

By default, the transport layer in Jersey is provided by `HttpUrlConnection`. This transport is implemented in Jersey via HttpUrlConnector [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/HttpUrlConnector.html] that implements Jersey-specific Connector [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/spi/Connector.html] SPI. You can implement and/or register your own `Connector` instance to the Jersey `Client` implementation, that will replace the default `HttpUrlConnection`-based transport layer. Jersey provides several alternative client transport connector implementations that are ready-to-use. You can use ApacheConnector [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/apache/ApacheConnector.html] (add a maven dependency to `org.glassfish.jersey.connectors:jersey-apache-connector`) or GrizzlyConnector [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/grizzly/GrizzlyConnector.html] (add a maven dependency to `org.glassfish.jersey.connectors:jersey-grizzly-connector`) alternatively. Please, note again, that the Connector [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/spi/Connector.html] SPI is not a feature of JAX-RS but is a Jersey-specific extension API that will only work with Jersey. Following example shows how to setup the custom `Connector` to the Jersey client.

```
1 ClientConfig clientConfig = new ClientConfig();
2 Connector connector = new GrizzlyConnector(clientConfig);
3 clientConfig.connector(connector);
4 Client client = ClientBuilder.newClient(clientConfig);
```

`Client` accepts as as a constructor argument a `Configurable` instance. Jersey implementation of the `Configurable` provider for the client is `ClientConfig`. By using the Jersey `ClientConfig` you can configure the custom `Connector` into the `ClientConfig`. The `GrizzlyConnector` is used as a custom connector in the example above. Please note that the connector cannot be registered as a provider using `Configurable.register(...)` at the moment.

5.6. Using client request and response filters

Filtering requests and responses can provide useful lower-level concept focused on a certain independent aspect or domain that is decoupled from the application layer of building and sending requests, and processing responses. Filters can read/modify the request URI, headers and entity or read/modify the response status, headers and entity.

Jersey contains the following useful client-side filters that you may want to use in your applications:

CsrfProtectionFilter [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/filter/CsrfProtectionFilter.html]: Cross-site request forgery protection filter (adds `X-Requested-By` to each state changing request).
EncodingFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/filter/EncodingFeature.html]: Feature that registers encoding filter which use registered ContentEncoder [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/common/spi/ContentEncoder.html]s to encode and decode the communication. The encoding/decoding is performed in interceptor (you don't need to register this interceptor). Check the javadoc of the EncodingFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/filter/EncodingFeature.html] in order to use it.
HttpBasicAuthFilter [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/filter/HttpBasicAuthFilter.html]: HTTP Basic Authentication filter (see ??? below).
Note that these features are provided by Jersey, but since they use and implement JAX-RS API, the features should be portable and run in any JAX-RS implementation, not just Jersey. See Chapter 9, *Filters and Interceptors* chapter for more information on filters and interceptors.

5.7. Securing a Client

This section describes how to setup SSL configuration on Jersey client (using JAX-RS API). The SSL configuration is setup in ClientBuilder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientBuilder.html]. The client builder contains methods for definition of KeyStore [http://docs.oracle.com/javase/6/docs/api/java/security/KeyStore.html], TrustStore [http://docs.oracle.com/javase/6/docs/api/java/security/TrustStore.html] or entire SslContext [http://docs.oracle.com/javase/6/docs/api/javax/net/ssl/SslContext.html]. See the following example:

```
1 SSLContext ssl = ... your configured SSL context;
2 Client client = ClientBuilder.newBuilder().sslContext(ssl).build();
3 Response response = client.target("https://example.com/resource").request().ge
```

The example above shows how to setup a custom `SslContext` to the `ClientBuilder`. Creating a `SslContext` can be more difficult as you might need to init instance properly with the protocol, `KeyStore`, `TrustStore`, etc. Jersey offers a utility ClientConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/SslConfigurator.html] class that can be used to setup the `SslContext`. The `SslConfigurator` can be configured based on standardized system properties for SSL configuration, so for example you can configure the `KeyStore` file name using a environment variable `javax.net.ssl.keyStore` and `SslConfigurator` will use such a variable to setup the `SslContext`. See javadoc of ClientConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/SslConfigurator.html] for more details. The following code shows how a `SslConfigurator` can be used to create a custom SSL context.

```
1 SslConfigurator sslConfig = SslConfigurator.newInstance()
2         .trustStoreFile("./truststore_client")
3         .trustStorePassword("secret-password-for-truststore")
4         .keyStoreFile("./keystore_client")
5         .keyPassword("secret-password-for-keystore");
6
7 SSLContext sslContext = sslConfig.createSSLContext();
8 Client client = ClientBuilder.newBuilder().sslContext(sslContext).build();
```

Note that you can also setup `KeyStore` and `TrustStore` directly on a `ClientBuilder` instance without wrapping them into the `SslContext`. However, if you setup a `SslContext` it will override any previously defined `KeyStore` and `TrustStore` settings. `ClientBuilder` also offers a method for defining a custom HostnameVerifier [http://docs.oracle.com/javase/6/docs/api/javax/net/ssl/HostnameVerifier.html] implementation. `HostnameVerifier` implementations are invoked when default host URL verification fails.

Important

Note that to utilize HTTP with SSL it is necessary to utilize the `“https”` scheme.

Currently the default connector `HttpUrlConnector` based on `HttpUrlConnection` implements support for SSL defined by JAX-RS configuration discussed in this sample.

5.7.1. HTTP Basic Authentication Support

Jersey contains a filter that allows client to authenticate to servers which requires HTTP Basic Authentication. Use HttpBasicAuthFilter [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/filter/HttpBasicAuthFilter.html] to add authentication header to requests initiated from from the client. See the example of how to configure and register the filter:

```
1 client.register(new HttpBasicAuthFilter("Homer", "SecretPassword"));
```

Chapter 6. Representations and Responses

6.1. Representations and Java Types

Previous sections on @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] and @Consumes [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Consumes.html] annotations referred to media type of an entity representation. Examples above depicted resource methods that could consume and/or produce String Java type for a number of different media types. This approach is easy to understand and relatively straightforward when applied to simple use cases.

To cover also other cases, handling non-textual data for example or handling data stored in the file system, etc., JAX-RS implementations are required to support also other kinds of media type conversions where additional, non-String, Java types are being utilized. Following is a short listing of the Java types that are supported out of the box with respect to supported media type:

- All media types (`*/*`)
  - `byte[]`
  - `java.lang.String`
  - `java.io.Reader` (inbound only)
  - `java.io.File`
  - `javax.activation.DataSource`
  - `javax.ws.rs.core.StreamingOutput` (outbound only)
- XML media types (`text/xml`, `application/xml` and `application/...+xml`)
  - `javax.xml.transform.Source`
  - `javax.xml.bind.JAXBElement`
  - Application supplied JAXB classes (types annotated with @XmlRootElement [http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/annotation/XmlRootElement.html] or@XmlType [http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/annotation/XmlType.html])
- Form content (`application/x-www-form-urlencoded`)
  - `MultivaluedMap<String,String>`
- Plain text (`text/plain`)
  - `java.lang.Boolean`
  - `java.lang.Character`
  - `java.lang.Number`

Unlike method parameters that are associated with the extraction of request parameters, the method parameter associated with the representation being consumed does not require annotating. In other words

the representation (entity) parameter does not require a specific 'entity' annotation. A method parameter without a annotation is an entity. A maximum of one such unannotated method parameter may exist since there may only be a maximum of one such representation sent in a request.

The representation being produced corresponds to what is returned by the resource method. For example JAX-RS makes it simple to produce images that are instance of `File` as follows:

Example 6.1. Using `File` with a specific media type to produce a response

```
 1 @GET
 2 @Path("/images/{image}")
 3 @Produces("image/*")
 4 public Response getImage(@PathParam("image") String image) {
 5   File f = new File(image);
 6
 7   if (!f.exists()) {
 8     throw new WebApplicationException(404);
 9   }
10
11   String mt = new MimetypesFileTypeMap().getContentType(f);
12   return Response.ok(f, mt).build();
13 }
```

The `File` type can also be used when consuming a representation (request entity). In that case a temporary file will be created from the incoming request entity and passed as a parameter to the resource method.

The `Content-Type` response header (if not set programmatically as described in the next section) will be automatically set based on the media types declared by @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] annotation. Given the following method, the most acceptable media type is used when multiple output media types are allowed:

```
1 @GET
2 @Produces({"application/xml", "application/json"})
3 public String doGetAsXmlOrJson() {
4   ...
5 }
```

If `application/xml` is the most acceptable media type defined by the request (e.g. by header `Accept: application/xml`), then the `Content-Type` response header will be set to `application/xml`.

6.2. Building Responses

Sometimes it is necessary to return additional information in response to a HTTP request. Such information may be built and returned using Response [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Response.html] and Response.ResponseBuilder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Response.ResponseBuilder.html]. For example, a common RESTful pattern for the creation of a new resource is to support a POST request that returns a 201 (Created) status code and a `Location` header whose value is the URI to the newly created resource. This may be achieved as follows:

Example 6.2. Returning 201 status code and adding `Location` header in response to POST request

```
1 @POST
2 @Consumes("application/xml")
3 public Response post(String content) {
4   URI createdUri = ...
5   create(content);
6   return Response.created(createdUri).build();
7 }
```

In the above no representation produced is returned, this can be achieved by building an entity as part of the response as follows:

Example 6.3. Adding an entity body to a custom response

```
1 @POST
2 @Consumes("application/xml")
3 public Response post(String content) {
4   URI createdUri = ...
5   String createdContent = create(content);
6   return Response.created(createdUri).entity(Entity.text(createdContent)).buil
7 }
```

Response building provides other functionality such as setting the entity tag and last modified date of the representation.

6.3. WebApplicationException and Mapping Exceptions to Responses

Previous section shows how to return HTTP responses, that are built up programmatically. It is possible to use the very same mechanism to return HTTP errors directly, e.g. when handling exceptions in a try-catch block. However, to better align with the Java programming model, JAX-RS allows to define direct mapping of Java exceptions to HTTP error responses.

The following example shows throwing `CustomNotFoundException` from a resource method in order to return an error HTTP response to the client:

Example 6.4. Throwing exceptions to control response

```
1 @Path("items/{itemid}/")
2 public Item getItem(@PathParam("itemid") String itemid) {
3   Item i = getItems().get(itemid);
4   if (i == null) {
5     throw new CustomNotFoundException("Item, " + itemid + ", is not found");
6   }
7
8   return i;
9 }
```

This exception is an application specific exception that extends WebApplicationException [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/WebApplicationException.html] and builds a HTTP response with the 404 status code and an optional message as the body of the response:

Example 6.5. Application specific exception implementation

```
 1 public class CustomNotFoundException extends WebApplicationException {
 2
 3   /**
 4   * Create a HTTP 404 (Not Found) exception.
 5   */
 6   public CustomNotFoundException() {
 7     super(Responses.notFound().build());
 8   }
 9
10   /**
11   * Create a HTTP 404 (Not Found) exception.
12   * @param message the String that is the entity of the 404 response.
13   */
14   public CustomNotFoundException(String message) {
15     super(Response.status(Responses.NOT_FOUND).
16     entity(message).type("text/plain").build());
17   }
18 }
```

In other cases it may not be appropriate to throw instances of WebApplicationException [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/WebApplicationException.html], or classes that extend WebApplicationException [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/WebApplicationException.html], and instead it may be preferable to map an existing exception to a response. For such cases it is possible to use a custom exception mapping provider. The provider must implement the ExceptionMapper<E extends Throwable> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/ExceptionMapper.html] interface. For example, the following maps the EntityNotFoundException [http://docs.oracle.com/javaee/5/api/javax/persistence/EntityNotFoundException.html] to a HTTP 404 (Not Found) response:

Example 6.6. Mapping generic exceptions to responses

```
1 @Provider
2 public class EntityNotFoundMapper implements ExceptionMapper<javax.persistence
3   public Response toResponse(javax.persistence.EntityNotFoundException ex) {
4     return Response.status(404).
5       entity(ex.getMessage()).
6       type("text/plain").
7       build();
8   }
9 }
```

The above class is annotated with @Provider [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/Provider.html], this declares that the class is of interest to the JAX-RS runtime. Such a class may be added to the set of classes of the Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] instance that is configured. When an application throws an EntityNotFoundException [http://docs.oracle.com/javaee/6/api/javax/persistence/EntityNotFoundException.html] the `toResponse` method of the `EntityNotFoundMapper` instance will be invoked.

Jersey supports extension of the exception mappers. These extended mappers must implement the `org.glassfish.jersey.spi.ExtendedExceptionMapper` interface. This interface additionally defines method `isMappable(Throwable)` which will be invoked by the Jersey runtime

when exception is thrown and this provider is considered as mappable based on the exception type. Using this method the provider can reject mapping of the exception before the method `toResponse` is invoked. The provider can for example check the exception parameters and based on them return false and let other provider to be chosen for the exception mapping.

6.4. Conditional GETs and Returning 304 (Not Modified) Responses

Conditional GETs are a great way to reduce bandwidth, and potentially improve on the server-side performance, depending on how the information used to determine conditions is calculated. A well-designed web site may for example return 304 (Not Modified) responses for many of static images it serves.

JAX-RS provides support for conditional GETs using the contextual interface Request [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Request.html].

The following example shows conditional GET support:

Example 6.7. Conditional GET support

```
 1 public SparklinesResource(
 2   @QueryParam("d") IntegerList data,
 3   @DefaultValue("0,100") @QueryParam("limits") Interval limits,
 4   @Context Request request,
 5   @Context UriInfo ui) {
 6   if (data == null) {
 7     throw new WebApplicationException(400);
 8   }
 9
10   this.data = data;
11   this.limits = limits;
12
13   if (!limits.contains(data)) {
14     throw new WebApplicationException(400);
15   }
16
17   this.tag = computeEntityTag(ui.getRequestUri());
18
19   if (request.getMethod().equals("GET")) {
20     Response.ResponseBuilder rb = request.evaluatePreconditions(tag);
21     if (rb != null) {
22       throw new WebApplicationException(rb.build());
23     }
24   }
25 }
```

The constructor of the `SparklinesResouce` root resource class computes an entity tag from the request URI and then calls the request.evaluatePreconditions [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Request.html#evaluatePreconditions(javax.ws.rs.core.EntityTag)] with that entity tag. If a client request contains an `If-None-Match` header with a value that contains the same entity tag that was calculated then the evaluatePreconditions [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Request.html#evaluatePreconditions(javax.ws.rs.core.EntityTag)] returns a pre-filled out response, with the 304 status code and entity tag set, that may be built and returned. Otherwise, evaluatePreconditions [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/

core/Request.html#evaluatePreconditions(javax.ws.rs.core.EntityTag)] returns `null` and the normal response can be returned.

Notice that in this example the constructor of a resource class is used to perform actions that may otherwise have to be duplicated to invoked for each resource method. The life cycle of resource classes is per-request which means that the resource instance is created for each request and therefore can work with request parameters and for example make changes to the request processing by throwing an exception as it is shown in this example.

Chapter 7. JAX-RS Entity Providers

7.1. Introduction

Entity payload, if present in an received HTTP message, is passed to Jersey from an I/O container as an input stream. The stream may, for example, contain data represented as a plain text, XML or JSON document. However, in many JAX-RS components that process these inbound data, such as resource methods or client responses, the JAX-RS API user can access the inbound entity as an arbitrary Java object that is created from the content of the input stream based on the representation type information. For example, an entity created from an input stream that contains data represented as a XML document, can be converted to a custom JAXB bean. Similar concept is supported for the outbound entities. An entity returned from the resource method in the form of an arbitrary Java object can be serialized by Jersey into a container output stream as a specified representation. Of course, while JAX-RS implementations do provide default support for most common combinations of Java type and it's respective on-the-wire representation formats, JAX-RS implementations do not support the conversion described above for any arbitrary Java type and any arbitrary representation format by default. Instead, a generic extension concept is exposed in JAX-RS API to allow application-level customizations of this JAX-RS runtime to support for entity conversions. The JAX-RS extension API components that provide the user-level extensibility are typically referred to by several terms with the same meaning, such as *entity providers*, *message body providers*, *message body workers* or *message body readers and writers*. You may find all these terms used interchangeably throughout the user guide and they all refer to the same concept.

In JAX-RS extension API (or SPI - service provider interface, if you like) the concept is captured in 2 interfaces. One for handling inbound entity representation-to-Java de-serialization - MessageBodyReader<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyReader.html] and the other one for handling the outbound entity Java-to-representation serialization - MessageBodyWriter<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyWriter.html]. A `MessageBodyReader<T>`, as the name suggests, is an extension that supports reading the message body representation from an input stream and converting the data into an instance of a specific Java type. A `MessageBodyWriter<T>` is then responsible for converting a message payload from an instance of a specific Java type into a specific representation format that is sent over the wire to the other party as part of an HTTP message exchange. Both of these providers can be used to provide message payload serialization and de-serialization support on the server as well as the client side. A message body reader or writer is always used whenever a HTTP request or response contains an entity and the entity is either requested by the application code (e.g. injected as a parameter of JAX-RS resource method or a response entity read on the client from a Response [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Response.html]) or has to be serialized and sent to the other party (e.g. an instance returned from a JAX-RS resource method or a request entity sent by a JAX-RS client).

7.2. How to Write Custom Entity Providers

A best way how to learn about entity providers is to walk through an example of writing one. Therefore we will describe here the process of implementing a custom MessageBodyWriter<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyWriter.html] and MessageBodyReader<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyReader.html] using a practical example. Let's first setup the stage by defining a JAX-RS resource class for the server side story of our application.

Example 7.1. Example resource class

```
 1 @Path("resource")
 2 public class MyResource {
 3     @GET
 4     @Produces("application/xml")
 5     public MyBean getMyBean() {
 6         return new MyBean("Hello World!", 42);
 7     }
 8
 9     @POST
10     @Consumes("application/xml")
11     public String postMyBean(MyBean myBean) {
12         return myBean.anyString;
13     }
14 }
```

The resource class defines `GET` and `POST` resource methods. Both methods work with an entity that is an instance of `MyBean`.

The `MyBean` class is defined in the next example:

Example 7.2. MyBean entity class

```
 1 @XmlRootElement
 2 public class MyBean {
 3     @XmlElement
 4     public String anyString;
 5     @XmlElement
 6     public int anyNumber;
 7
 8     public MyBean(String anyString, int anyNumber) {
 9         this.anyString = anyString;
10         this.anyNumber = anyNumber;
11     }
12
13     // empty constructor needed for deserialization by JAXB
14     public MyBean() {
15     }
16
17     @Override
18     public String toString() {
19         return "MyBean{" +
20             "anyString='" + anyString + '\'' +
21             ", anyNumber=" + anyNumber +
22             '}';
23     }
24 }
```

7.2.1. MessageBodyWriter

The `MyBean` is a JAXB-annotated POJO. In `GET` resource method we return the instance of MyBean and we would like Jersey runtime to serialize it into XML and write it as an entity body to the response output stream. We design a custom `MessageBodyWriter<T>` that can serialize this POJO into XML. See the following code sample:

Note

Please note, that this is only a demonstration of how to write a custom entity provider. Jersey already contains default support for entity providers that can serialize JAXB beans into XML.

Example 7.3. MessageBodyWriter example

```
@Produces("application/xml")
public class MyBeanMessageBodyWriter implements MessageBodyWriter<MyBean> {

    @Override
    public boolean isWriteable(Class<?> type, Type genericType,
                               Annotation[] annotations, MediaType mediaType)
        return type == MyBean.class;
    }

    @Override
    public long getSize(MyBean myBean, Class<?> type, Type genericType,
                        Annotation[] annotations, MediaType mediaType) {
        // deprecated by JAX-RS 2.0 and ignored by Jersey runtime
        return 0;
    }

    @Override
    public void writeTo(MyBean myBean,
                        Class<?> type,
                        Type genericType,
                        Annotation[] annotations,
                        MediaType mediaType,
                        MultivaluedMap<String, Object> httpHeaders,
                        OutputStream entityStream)
                        throws IOException, WebApplicationException {

        try {
            JAXBContext jaxbContext = JAXBContext.newInstance(MyBean.class);

            // serialize the entity myBean to the entity output stream
            jaxbContext.createMarshaller().marshal(myBean, entityStream);
        } catch (JAXBException jaxbException) {
            throw new ProcessingException(
                "Error serializing a MyBean to the output stream", jaxbExcepti
        }
    }
}
```

The `MyBeanMessageBodyWriter` implements the `MessageBodyWriter<T>` interface that contains three methods. In the next sections we'll explore these methods more closely.

7.2.1.1. `MessageBodyWriter.isWriteable`

A method `isWriteable` should return true if the `MessageBodyWriter<T>` is able to write the given type. Method does not decide only based on the Java type of the entity but also on annotations attached to the entity and the requested representation media type.

Parameters `type` and `genericType` both define the entity, where `type` is a raw Java type (for example, a `java.util.List` class> and `genericType` is a ParameterizedType [http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/ParameterizedType.html] including generic information (for example `List<String>`).

Parameter `annotations` contains annotations that are either attached to the resource method and/or annotations that are attached to the entity by building response like in the following piece of code:

Example 7.4. Example of assignment of annotations to a response entity

```
 1 @Path("resource")
 2 public static class AnnotatedResource {
 3
 4     @GET
 5     public Response get() {
 6         Annotation annotation = AnnotatedResource.class
 7                                 .getAnnotation(Path.class);
 8         return Response.ok()
 9                 .entity("Entity", new Annotation[] {annotation}).build();
10     }
11 }
```

In the example above, the `MessageBodyWriter<T>` would get `annotations` parameter containing a JAX-RS `@GET` annotation as it annotates the resource method and also a `@Path` annotation as it is passed in the response (but not because it annotates the resource; only resource method annotations are included). In the case of `MyResource` and method `getMyBean` the annotations would contain the @GET [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/GET.html] and the @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] annotation.

The last parameter of the `isWriteable` method is the `mediaType` which is the media type attached to the response entity by annotating the resource method with a `@Produces` annotation or the request media type specified in the JAX-RS Client API. In our example, the media type passed to providers for the resource `MyResource` and method `getMyBean` would be `"application/xml"`.

In our implementation of the `isWriteable` method, we just check that the type is `MyBean`. Please note, that this method might be executed multiple times by Jersey runtime as Jersey needs to check whether this provider can be used for a particular combination of entity Java type, media type, and attached annotations, which may be potentially a performance hog. You can limit the number of execution by properly defining the `@Produces` annotation on the `MessageBodyWriter<T>`. In our case thanks to `@Produces` annotation, the provider will be considered as writeable (and the method `isWriteable` might be executed) only if the media type of the outbound message is `"application/xml"`. Additionally, the provider will only be considered as possible candidate and its `isWriteable` method will be executed, if the generic type of the provider is either a sub class or super class of `type` parameter.

7.2.1.2. `MessageBodyWriter.writeTo`

Once a message body writer is selected as the most appropriate (see the Section 7.3, “Entity Provider Selection” for more details on entity provider selection), its `writeTo` method is invoked. This method receives parameters with the same meaning as in `isWriteable` as well as a few additional ones.

In addition to the parameters already introduced, the `writeTo` method defies also `httpHeaders` parameter, that contains HTTP headers associated with the outbound message.

Note

When a `MessageBodyWriter<T>` is invoked, the headers still can be modified in this point and any modification will be reflected in the outbound HTTP message being sent. The modification of headers must however happen before a first byte is written to the supplied output stream.

Another new parameter, `myBean`, contains the entity instance to be serialized (the type of entity corresponds to generic type of `MessageBodyWriter<T>`). Related parameter `entityStream` contains the entity output stream to which the method should serialize the entity. In our case we use JAXB to marshall the entity into the `entityStream`. Note, that the `entityStream` is not closed at the end of method; the stream will be closed by Jersey.

Important

Do not close the entity output stream in the `writeTo` method of your `MessageBodyWriter<T>` implementation.

7.2.1.3. `MessageBodyWriter.getSize`

The method is deprecated since JAX-RS 2.0 and Jersey 2 ignores the return value. In JAX-RS 1.0 the method could return the size of the entity that would be then used for "Content-Length" response header. In Jersey 2.0 the "Content-Length" parameter is computed automatically using an internal outbound entity buffering. For details about configuration options of outbound entity buffering see the javadoc of MessageProperties [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/message/MessageProperties.html], property `OUTBOUND_CONTENT_LENGTH_BUFFER` which configures the size of the buffer.

Note

You can disable the Jersey outbound entity buffering by setting the buffer size to 0.

7.2.1.4. Testing a `MessageBodyWriter<T>`

Before testing the `MyBeanMessageBodyWriter`, the writer must be registered as a custom JAX-RS extension provider. It should either be added to your application ResourceConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ResourceConfig.html], or returned from your custom Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] sub-class, or annotated with @Provider [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/Provider.html] annotation to leverage JAX-RS provider auto-discovery feature.

After registering the `MyBeanMessageBodyWriter` and `MyResource` class in our application, the request can be initiated (in this example from Client API).

Example 7.5. Client code testing MyBeanMessageBodyWriter

```
1 WebTarget webTarget = // initialize web target to the context root
2             // of example application
3 Response response = webTarget.path("resource")
4                         .request(MediaType.APPLICATION_XML).get();
5 System.out.println(response.getStatus());
6 String myBeanXml = response.readEntity(String.class);
7 System.out.println(myBeanXml);
```

The client code initiates the `GET` which will be matched to the resource method `MyResource.getMyBean()`. The response entity is de-serialized as a `String`.

The result of console output is:

Example 7.6. Result of MyBeanMessageBodyWriter test

```
200
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><myBean>
<anyString>Hello World!</anyString><anyNumber>42</anyNumber></myBean>
```

The returned status is 200 and the entity is stored in the response in a `XML` format. Next, we will look at how the Jersey de-serializes this XML document into a `MyBean` consumed by our `POST` resource method.

7.2.2. MessageBodyReader

In order to de-serialize the entity of `MyBean` on the server or the client, we need to implement a custom MessageBodyReader<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyReader.html].

> **Note**
>
> Please note, that this is only a demonstration of how to write a custom entity provider. Jersey already contains default support for entity providers that can serialize JAXB beans into XML.

Our `MessageBodyReader<T>` implementation is listed in Example 7.7, “MessageBodyReader example”.

Example 7.7. MessageBodyReader example

```
 1 public static class MyBeanMessageBodyReader
 2         implements MessageBodyReader<MyBean> {
 3
 4 @Override
 5 public boolean isReadable(Class<?> type, Type genericType,
 6     Annotation[] annotations, MediaType mediaType) {
 7     return type == MyBean.class;
 8 }
 9
10 @Override
11 public MyBean readFrom(Class<MyBean> type,
12     Type genericType,
13     Annotation[] annotations, MediaType mediaType,
14     MultivaluedMap<String, String> httpHeaders,
15     InputStream entityStream)
16         throws IOException, WebApplicationException {
17
18     try {
19         JAXBContext jaxbContext = JAXBContext.newInstance(MyBean.class);
20         MyBean myBean = (MyBean) jaxbContext.createUnmarshaller()
21             .unmarshal(entityStream);
22         return myBean;
23     } catch (JAXBException jaxbException) {
24         throw new ProcessingException("Error deserializing a MyBean.",
25             jaxbException);
26     }
27 }
28 }
```

It is obvious that the `MessageBodyReader<T>` interface is similar to `MessageBodyWriter<T>`. In the next couple of sections we will explore it's API methods.

7.2.2.1. `MessageBodyReader.isReadable`

It defines the method `isReadable()` which has a very simliar meaning as method `isWriteable()` in `MessageBodyWriter<T>`. The method returns `true` if it is able to de-serialize the given type. The `annotations` parameter contains annotations that are attached to the entity parameter in the resource method. In our `POST` resource method `postMyBean` the entity parameter `myBean` is not annotated, therefore no annotation will be passed to the isReadable. The `mediaType` parameter contains the entity media type. The media type, in our case, must be consumable by the `POST` resource method, which is specified by placing a JAX-RS @Consumes [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Consumes.html] annotation to the method. The resource method `postMyBean()` is annotated with `@Consumes("application/xml")`, therefore for purpose of de-serialization of entity for the `postMyBean()` method, only requests with entities represented as `"application/xml"` media type will match the method. However, this method might be executed for for entity types that are sub classes or super classes of the declared generic type on the `MessageBodyReader<T>` will be also considered. It is a responsibility of the `isReadable` method to decide whether it is able to de-serialize the entity and type comparison is one of the basic decision steps.

Tip

In order to reduce number of `isReadable` executions, always define correctly the consumable media type(s) with the `@Consumes` annotation on your custom `MessageBodyReader<T>`.

7.2.2.2. `MessageBodyReader.readFrom`

The `readForm()` method gets the parameters with the same meaning as in `isReadable()`. The additional `entityStream` parameter provides a handle to the entity input stream from which the entity bytes should be read and de-serialized into a Java entity which is then returned from the method. Our `MyBeanMessageBodyReader` de-serializes the incoming XML data into an instance of `MyBean` using JAXB.

Important

Do not close the entity input stream in your `MessageBodyReader<T>` implementation. The stream will be automatically closed by Jersey runtime.

7.2.2.3. Testing a `MessageBodyWriter<T>`

Now let's send a test request using the JAX-RS Client API.

Example 7.8. Testing MyBeanMessageBodyReader

```
1 final MyBean myBean = new MyBean("posted MyBean", 11);
2 Response response = webTarget.path("resource").request("application/xml")
3         .post(Entity.entity(myBean, "application/xml"));
4
5 System.out.println(response.getStatus());
6 final String responseEntity = response.readEntity(String.class);
7 System.out.println(responseEntity);
8
```

The console output is:

Example 7.9. Result of testing MyBeanMessageBodyReader

```
200
posted MyBean
```

7.2.2.4. Using Entity Providers with JAX-RS Client API

Both, `MessageBodyReader<T>` and `MessageBodyWriter<T>` can be registered in a configuration of JAX-RS Client API components typically without any need to change their code. The example Example 7.10, “MessageBodyReader registered on a JAX-RS client” is a variation on the Example 7.5, “Client code testing MyBeanMessageBodyWriter” listed in one of the previous sections.

Example 7.10. MessageBodyReader registered on a JAX-RS client

```
1 Client client = ClientBuilder.newBuilder()
2     .register(MyBeanMessageBodyReader.class).build();
3
4 Response response = client.target("http://example/comm/resource")
5     .request(MediaType.APPLICATION_XML).get();
6 System.out.println(response.getStatus());
7 MyBean myBean = response.readEntity(MyBean.class);
8 System.out.println(myBean);
```

The code above registers `MyBeanMessageBodyReader` to the Client [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/Client.html] configuration using a ClientBuilder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientBuilder.html] which means that the provider will be used for any WebTarget [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/WebTarget.html] produced by the `client` instance.

> **Note**
>
> You could also register the JAX-RS entity (and any other) providers to individual `WebTarget` instances produced by the client.

Then, using the fluent chain of method invocations, a resource target pointing to our `MyResource` is defined, a HTTP `GET` request is invoked. The response entity is then read as an instance of a `MyBean` type by invoking the `response.readEntity` method, that internally locates the registered `MyBeanMessageBodyReader` and uses it for entity de-serialization.

The console output for the example is:

Example 7.11. Result of client code execution

```
200
MyBean{anyString='Hello World!', anyNumber=42}
```

7.3. Entity Provider Selection

Usually there are many entity providers registered on the server or client side (be default there must be at least providers mandated by the JAX-RS specification, such as providers for primitive types, byte array, JAXB beans, etc.). JAX-RS defines an algorithm for selecting the most suitable provider for entity

processing. This algorithm works with information such as entity Java type and on-the-wire media type representation of entity, and searches for the most suitable entity provider from the list of available providers based on the supported media type declared on each provider (defined by `@Produces` or `@Consumes` on the provider class) as well as based on the generic type declaration of the available providers. When a list of suitable candidate entity providers is selected and sorted based on the rules defined in JAX-RS specification, a JAX-RS runtime then it invokes `isReadable` or `isWriteable` method respectively on each provider in the list until a first provider is found that returns `true`. This provider is then used to process the entity.

The following steps describe the algorithm for selecting a `MessageBodyWriter<T>` (extracted from JAX-RS with little modifications). The steps refer to the previously discussed example application. The `MessageBodyWriter<T>` is searched for purpose of deserialization of `MyBean` entity returned from the method `getMyBean`. So, *type is `MyBean` and media type `"application/xml"`*. Let's assume the runtime contains also registered providers, namely:

`A`: `@Produces("application/*")` with generic type `<Object>`
`B`: `@Produces("*/*")` with generic type `<MyBean>`
`C`: `@Produces("text/plain")` with generic type `<MyBean>`
`D`: `@Produces("application/xml")` with generic type `<Object>`
`MyBeanMessageBodyWriter`: `@Produces("application/xml")` with generic type `<MyBean>`

The algorithm executed by a JAX-RS runtime to select a proper `MessageBodyWriter<T>` implementation is illustrated in Procedure 7.1, “`MessageBodyWriter<T>` Selection Algorithm”.

Procedure 7.1. `MessageBodyWriter<T>` Selection Algorithm

1. Obtain the object that will be mapped to the message entity body. For a return type of Response or subclasses, the object is the value of the entity property, for other return types it is the returned object.

   So in our case, for the resource method `getMyBean` the type will be `MyBean`.

2. Determine the media type of the response.

   In our case. for resource method `getMyBean` annotated with `@Produces("application/xml")`, the media type will be `"application/xml"`.

3. Select the set of MessageBodyWriter providers that support the object and media type of the message entity body.

   In our case, for entity media type `"application/xml"` and type `MyBean`, the appropriate `MessageBodyWriter<T>` will be the `A`, `B`, `D` and `MyBeanMessageBodyWriter`. The provider `C` does not define the appropriate media type. `A` and `B` are fine as their type is more generic and compatible with `"application/xml"`.

4. Sort the selected MessageBodyWriter providers with a primary key of generic type where providers whose generic type is the nearest superclass of the object class are sorted first and a secondary key of media type. Additionally, JAX-RS specification mandates that custom, user registered providers have to be sorted ahead of default providers provided by JAX-RS implementation. This is used as a tertiary comparison key. User providers are places prior to Jersey internal providers in to the final ordered list.

   The sorted providers will be: `MyBeanMessageBodyWriter`, `B`. `D`, `A`.

5. Iterate through the sorted `MessageBodyWriter<T>` providers and, utilizing the `isWriteable` method of each until you find a `MessageBodyWriter<T>` that returns `true`.

The first provider in the list - our `MyBeanMessageBodyWriter` returns `true` as it compares types and the types matches. If it would return `false`, the next provider `B` would by check by invoking its `isWriteable` method.

6. If step 5 locates a suitable `MessageBodyWriter<T>` then use its writeTo method to map the object to the entity body.

   `MyBeanMessageBodyWriter.writeTo` will be executed and it will serialize the entity.

   - Otherwise, the server runtime MUST generate a generate an `InternalServerErrorException`, a subclass of `WebApplicationException` with its status set to 500, and no entity and the client runtime MUST generate a `ProcessingException`.

     We have successfully found a provider, thus no exception is generated.

Note

JAX-RS 2.0 is incompatible with JAX-RS 1.x in one step of the entity provider selection algorithm. JAX-RS 1.x defines sorting keys priorities in the Step 4 in exactly opposite order. So, in JAX-RS 1.x the keys are defined in the order: primary media type, secondary type declaration distance where custom providers have always precedence to internal providers. If you want to force Jersey to use the algorithm compatible with JAX-RS 1.x, setup the property (to ResourceConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ResourceConfig.html] or return from Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] from its `getProperties` method):

```
jersey.config.workers.legacyOrdering=true
```

Documentation of this property can be found in the javadoc of MessageProperties [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/message/MessageProperties.html].

The algorithm for selection of `MessageBodyReader<T>` is similar, including the incompatibility between JAX-RS 2.0 and JAX-RS 1.x and the property to workaround it. The algorithm is defined as follows:

Procedure 7.2. `MessageBodyReader<T>` Selection Algorithm

1. Obtain the media type of the request. If the request does not contain a `Content-Type` header then use `application/octet-stream` media type.

2. Identify the Java type of the parameter whose value will be mapped from the entity body. The Java type on the server is the type of the entity parameter of the resource method. On the client it is the `Class` passed to `readFrom` method.

3. Select the set of available `MessageBodyReader<T>` providers that support the media type of the request.

4. Iterate through the selected `MessageBodyReader<T>` classes and, utilizing their `isReadable` method, choose the first `MessageBodyReader<T>` provider that supports the desired combination of Java type/media type/annotations parameters.

5. If Step 4 locates a suitable `MessageBodyReader<T>`, then use its `readFrom` method to map the entity body to the desired Java type.

- Otherwise, the server runtime MUST generate a `NotSupportedException` (HTTP 415 status code) and no entity and the client runtime MUST generate an instance of `ProcessingException`.

7.4. Jersey `MessageBodyWorkers` API

In case you need to directly work with JAX-RS entity providers, for example to serialize an entity in your resource method, filter or in a composite entity provider, you would need to perform quite a lot of steps. You would need to choose the appropriate `MessageBodyWriter<T>` based on the type, media type and other parameters. Then you would need to instantiate it, check it by `isWriteable` method and basically perform all the steps that are normally performed by Jersey (see Procedure 7.2, "`MessageBodyReader<T>` Selection Algorithm").

To remove this burden from developers, Jersey exposes a proprietary public API that simplifies the manipulation of entity providers. The API is defined by MessageBodyWorkers [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/message/MessageBodyWorkers.html] interface and Jersey provides an implementation that can be injected using the @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html] injection annotation. The interface declares methods for selection of most appropriate `MessageBodyReader<T>` and `MessageBodyWriter<T>` based on the rules defined in JAX-RS spec, methods for writing and reading entity that ensure proper and timely invocation of interceptors and other useful methods.

See the following example of usage of `MessageBodyWorkers`.

Example 7.12. Usage of MessageBodyWorkers interface

```
@Path("workers")
public static class WorkersResource {

    @Context
    private MessageBodyWorkers workers;

    @GET
    @Produces("application/xml")
    public String getMyBeanAsString() {

        final MyBean myBean = new MyBean("Hello World!", 42);

        // buffer into which myBean will be serialized
        ByteArrayOutputStream baos = new ByteArrayOutputStream();

        // get most appropriate MBW
        final MessageBodyWriter<MyBean> messageBodyWriter =
                workers.getMessageBodyWriter(MyBean.class, MyBean.class,
                        new Annotation[]{}, MediaType.APPLICATION_XML_TYPE);

        try {
            // use the MBW to serialize myBean into baos
            messageBodyWriter.writeTo(myBean,
                MyBean.class, MyBean.class, new Annotation[] {},
                MediaType.APPLICATION_XML_TYPE, new MultivaluedHashMap<String,
                baos);
        } catch (IOException e) {
            throw new RuntimeException(
                "Error while serializing MyBean.", e);
        }

        final String stringXmlOutput = baos.toString();
        // stringXmlOutput now contains XML representation:
        // "<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
        // <myBean><anyString>Hello World!</anyString>
        // <anyNumber>42</anyNumber></myBean>"

        return stringXmlOutput;
    }
}
```

In the example a resource injects `MessageBodyWorkers` and uses it for selection of the most appropriate `MessageBodyWriter<T>`. Then the writer is utilized to serialize the entity into the buffer as XML document. The `String` content of the buffer is then returned. This will cause that Jersey will not use `MyBeanMessageBodyWriter` to serialize the entity as it is already in the `String` type (`MyBeanMessageBodyWriter` does not support `String`). Instead, a simple `String`-based `MessageBodyWriter<T>` will be chosen and it will only serialize the `String` with XML to the output entity stream by writing out the bytes of the `String`.

Of course, the code in the example does not bring any benefit as the entity could have been serialized by `MyBeanMessageBodyWriter` by Jersey as in previous examples; the purpose of the example was to show how to use `MessageBodyWorkers` in a resource method.

7.5. Default Jersey Entity Providers

Jersey internally contains entity providers for these types with combination of media types (in brackets):

`byte[]` (`*/*`)
String [http://docs.oracle.com/javase/6/docs/api/java/io/String.html] (`*/*`)
InputStream [http://docs.oracle.com/javase/6/docs/api/java/io/InputStream.html] (`*/*`)
Reader [http://docs.oracle.com/javase/6/docs/api/java/io/Reader.html] (`*/*`)
File [http://docs.oracle.com/javase/6/docs/api/java/io/File.html] (`*/*`)
DataSource [http://docs.oracle.com/javase/6/docs/api/javax/activation/DataSource.html] (`*/*`)
Source [http://docs.oracle.com/javase/6/docs/api/javax/xml/transform/Source.html] (`text/xml`, `application/xml` and media types of the form `application/*+xml`)
JAXBElement [http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/JAXBElement.html] (`text/xml`, `application/xml` and media types of the form `application/*+xml`)
MultivaluedMap<K,V> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/MultivaluedMap.html] (`application/x-www-form-urlencoded`)
Form [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Form.html] (`application/x-www-form-urlencoded`)
StreamingOutput [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/StreamingOutput.html] ((`*/*`)) - this class can be used as an lightweight `MessageBodyWriter<T>` that can be returned from a resource method
Boolean [http://docs.oracle.com/javase/6/docs/api/java/lang/Boolean.html], Character [http://docs.oracle.com/javase/6/docs/api/java/lang/Character.html] and Number [http://docs.oracle.com/javase/6/docs/api/java/lang/Number.html] (`text/plain`) - corresponding primitive types supported via boxing/unboxing conversion

For other media type supported in jersey please see the Chapter 8, *Support for Common Media Type Representations* which describes additional Jersey entity provider extensions for serialization to JSON, XML, serialization of collections, Multi Part [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/multipart/package-info.html] and others.

Chapter 8. Support for Common Media Type Representations

8.1. JSON

Jersey JSON support comes as a set of extension modules where each of these modules contains an implementation of a Feature [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Feature.html] that needs to be registered into your Configurable [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Configurable.html] instance (client/server). There are multiple frameworks that provide support for JSON processing and/or JSON-to-Java binding. The modules listed bellow provide support for JSON representations by integrating the individual JSON frameworks into Jersey. At present, Jersey integrates with the following modules to provide JSON support:

- MOXy - JSON binding support via MOXy is a default and preferred way of supporting JSON binding in your Jersey applications since Jersey 2.0. When JSON MOXy module is on the class-path, Jersey will automatically discover the module and seamlessly enable JSON binding support via MOXy in your applications. (See Section 4.1, “Auto-Discoverable Features”.)
- Java API for JSON Processing (JSON-P)
- Jackson
- Jettison

8.1.1. Approaches to JSON Support

Each of the aforementioned extension modules uses one or more of the three basic approaches available when working with JSON representations:

- POJO based JSON binding support
- JAXB based JSON binding support
- Low-level JSON parsing & processing support

The first method is pretty generic and allows you to map any Java Object to JSON and vice versa. The other two approaches limit you in Java types your resource methods could produce and/or consume. JAXB based approach is useful if you plan to utilize certain JAXB features and support both XML and JSON representations. The last, low-level, approach gives you the best fine-grained control over the out-coming JSON data format.

8.1.1.1. POJO support

POJO support represents the easiest way to convert your Java Objects to JSON and back.

Media modules that support this approach are MOXy and Jackson

8.1.1.2. JAXB based JSON support

Taking this approach will save you a lot of time, if you want to easily produce/consume both JSON and XML data format. With JAXB beans you will be able to use a the same Java model to generate JSON as well

as XML representations. Another advantage is simplicity of working with such a model and availability of the API in Java SE Platform. JAXB leverages annotated POJOs and these could be handled as simple Java beans.

A disadvantage of JAXB based approach could be if you need to work with a very specific JSON format. Then it might be difficult to find a proper way to get such a format produced and consumed. This is a reason why a lot of configuration options are provided, so that you can control how JAXB beans get serialized and de-serialized. The extra configuration options however requires you to learn more details about the framework you are using.

Following is a very simple example of how a JAXB bean could look like.

Example 8.1. Simple JAXB bean implementation

```
 1 @XmlRootElement
 2 public class MyJaxbBean {
 3     public String name;
 4     public int age;
 5
 6     public MyJaxbBean() {} // JAXB needs this
 7
 8     public MyJaxbBean(String name, int age) {
 9         this.name = name;
10         this.age = age;
11     }
12 }
```

Using the above JAXB bean for producing JSON data format from you resource method, is then as simple as:

Example 8.2. JAXB bean used to generate JSON representation

```
1 @GET
2 @Produces("application/json")
3 public MyJaxbBean getMyBean() {
4     return new MyJaxbBean("Agamemnon", 32);
5 }
```

Notice, that JSON specific mime type is specified in `@Produces` annotation, and the method returns an instance of `MyJaxbBean`, which JAXB is able to process. Resulting JSON in this case would look like:

```
{"name":"Agamemnon", "age":"32"}
```

A proper use of JAXB annotations itself enables you to control output JSON format to certain extent. Specifically, renaming and omitting items is easy to do directly just by using JAXB annotations. For example, the following example depicts changes in the above mentioned MyJaxbBean that will result in `{"king":"Agamemnon"}` JSON output.

Example 8.3. Tweaking JSON format using JAXB

```
 1 @XmlRootElement
 2 public class MyJaxbBean {
 3
 4     @XmlElement(name="king")
 5     public String name;
 6
 7     @XmlTransient
 8     public int age;
 9
10     // several lines removed
11 }
```

Media modules that support this approach are MOXy, Jackson, Jettison

8.1.1.3. Low-level based JSON support

JSON Processing API is a new standard API for parsing and processing JSON structures in similar way to what SAX and StAX parsers provide for XML. The API is part of Java EE 7 and later. Another such JSON parsing/processing API is provided by Jettison framework. Both APIs provide a low-level access to producing and consuming JSON data structures. By adopting this low-level approach you would be working with `JsonObject` (or `JSONObject` respectively) and/or `JsonArray` (or `JSONArray` respectively) classes when processing your JSON data representations.

The biggest advantage of these low-level APIs is that you will gain full control over the JSON format produced and consumed. You will also be able to produce and consume very large JSON structures using streaming JSON parser/generator APIs. On the other hand, dealing with your data model objects will probably be a lot more complex, compared to the POJO or JAXB based binding approach. Differences are depicted at the following code snippets.

Let's start with JAXB-based approach.

Example 8.4. JAXB bean creation

```
1 MyJaxbBean myBean = new MyJaxbBean("Agamemnon", 32);
```

Above you construct a simple JAXB bean, which could be written in JSON as `{"name":"Agamemnon", "age":32}`

Now to build an equivalent `JsonObject`/`JSONObject` (in terms of resulting JSON expression), you would need several more lines of code. The following example illustrates how to construct the same JSON data using the standard Java EE 7 JSON-Processing API.

Example 8.5. Constructing a `JsonObject` (JSON-Processing)

```
1 JsonObject myObject = Json.createObjectBuilder()
2         .add("name", "Agamemnon")
3         .add("age", 32)
4         .build();
```

And at last, here's how the same work can be done with Jettison API.

Example 8.6. Constructing a `JSONObject` (Jettison)

```
1 JSONObject myObject = new JSONObject();
2 try {
3     myObject.put("name", "Agamemnon");
4     myObject.put("age", 32);
5 } catch (JSONException ex) {
6     LOGGER.log(Level.SEVERE, "Error ...", ex);
7 }
```

Media modules that support the low-level JSON parsing and generating approach are Java API for JSON Processing (JSON-P) and Jettison. Unless you have a strong reason for using the non-standard Jettison API, we recommend you to use the new standard Java API for JSON Processing (JSON-P) API instead.

8.1.2. MOXy

8.1.2.1. Dependency

To use MOXy as your JSON provider you need to add `jersey-media-moxy` module to your `pom.xml` file:

```
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-moxy</artifactId>
    <version>2.1</version>
</dependency>
```

If you're not using Maven make sure to have all needed dependencies (see jersey-media-moxy [https://jersey.java.net/project-info/2.1/jersey/project/jersey-media-moxy/dependencies.html]) on the classpath.

8.1.2.2. Configure and register

As stated in the Section 4.1, “Auto-Discoverable Features” as well as earlier in this chapter, MOXy media module is one of the modules where you don't need to explicitly register it's `Feature`s (`MoxyJsonFeature`) in your client/server Configurable [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Configurable.html] as this feature is automatically discovered and registered when you add `jersey-media-moxy` module to your class-path.

The auto-discoverable `jersey-media-moxy` module defines a few properties that can be used to control the automatic registration of `MoxyJsonFeature` (besides the generic CommonProperties.FEATURE_AUTO_DISCOVERY_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/CommonProperties.html#FEATURE_AUTO_DISCOVERY_DISABLE] an the its client/server variants):

- CommonProperties.MOXY_JSON_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/CommonProperties.html#MOXY_JSON_FEATURE_DISABLE]
- ServerProperties.MOXY_JSON_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#MOXY_JSON_FEATURE_DISABLE]
- ClientProperties.MOXY_JSON_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html#MOXY_JSON_FEATURE_DISABLE]

Note

A manual registration of any other Jersey JSON provider feature (except for Java API for JSON Processing (JSON-P)) disables the automated enabling and configuration of `MoxyJsonFeature`.

To configure MessageBodyReader<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyReader.html]s / MessageBodyWriter<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyWriter.html]s provided by MOXy you can simply create an instance of MoxyJsonConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/moxy/json/MoxyJsonConfig.html] and set values of needed properties. For most common properties you can use a particular method to set the value of the property or you can use more generic methods to set the property:

- MoxyJsonConfig#property(java.lang.String, java.lang.Object) [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/moxy/json/MoxyJsonConfig.html#property(java.lang.String, java.lang.Object)] - sets a property value for both Marshaller and Unmarshaller.

- MoxyJsonConfig#marshallerProperty(java.lang.String, java.lang.Object) [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/moxy/json/MoxyJsonConfig.html#marshallerProperty(java.lang.String, java.lang.Object)] - sets a property value for Marshaller.

- MoxyJsonConfig#unmarshallerProperty(java.lang.String, java.lang.Object) [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/moxy/json/MoxyJsonConfig.html#unmarshallerProperty(java.lang.String, java.lang.Object)] - sets a property value for Unmarshaller.

Example 8.7. MoxyJsonConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/moxy/json/MoxyJsonConfig.html] - Setting properties.

```
final Map<String, String> namespacePrefixMapper = new HashMap<String, String>();
namespacePrefixMapper.put("http://www.w3.org/2001/XMLSchema-instance", "xsi");

final MoxyJsonConfig configuration = new MoxyJsonConfig()
        .setNamespacePrefixMapper(namespacePrefixMapper)
        .setNamespaceSeparator(':');
```

In order to make `MoxyJsonConfig` visible for MOXy you need to create and register `ContextResolver<T>` in your client/server code.

Example 8.8. `ContextResolver<MoxyJsonConfig>`

```
@Provider
public class JsonMoxyConfigurationContextResolver implements ContextResolver<MoxyJ

    private final MoxyJsonConfig config;

    public JsonMoxyConfigurationContextResolver() {
        final Map<String, String> namespacePrefixMapper = new HashMap<String, Stri
        namespacePrefixMapper.put("http://www.w3.org/2001/XMLSchema-instance", "xs

        config = MoxyJsonConfig()
            .setNamespacePrefixMapper(namespacePrefixMapper)
            .setNamespaceSeparator(':');
    }

    @Override
    public MoxyJsonConfig getContext(Class<?> objectType) {
        return config;
    }
}
```

Another way to pass configuration properties to the underlying `MOXyJsonProvider` is to set them directly into your Configurable [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Configurable.html] instance (see an example below). These are overwritten by properties set into the MoxyJsonConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/moxy/json/MoxyJsonConfig.html].

Example 8.9. Setting properties for MOXy providers into Configurable [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Configurable.html]

```
new ResourceConfig()
                            .property(MarshallerProperties.JSON_NAMESPACE_SEPARATO
                            // further configuration
```

There are some properties for which Jersey sets the default value when MessageBodyReader<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyReader.html] / MessageBodyWriter<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyWriter.html] from MOXy is used and they are:

Table 8.1. Default property values for MOXy MessageBodyReader<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyReader.html] / MessageBodyWriter<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyWriter.html]

javax.xml.bind.Marshaller#JAXB_FORMATTED_OUTPUT	false
org.eclipse.persistence.jaxb.JAXBContextProperties#JSON_INCLUDE_ROOT	false
org.eclipse.persistence.jaxb.MarshallerProperties#JSON_MARSHAL_EMPTY_COLLECTIONS	true
org.eclipse.persistence.jaxb.JAXBContextProperties#JSON_NAMESPACE_SEPARATOR	#

Example 8.10. Building client with MOXy JSON feature enabled.

```
final Client client = ClientBuilder.newBuilder()
        // The line bellow that registers MOXy feature can be
        // omitted if FEATURE_AUTO_DISCOVERY_DISABLE is
        // not disabled.
        .register(MoxyJsonFeature.class)
        .register(JsonMoxyConfigurationContextResolver.class)
        .build();
```

Example 8.11. Creating JAX-RS application with MOXy JSON feature enabled.

```
// Create JAX-RS application.
final Application application = new ResourceConfig()
        .packages("org.glassfish.jersey.examples.jsonmoxy")
        // The line bellow that registers MOXy feature can be
        // omitted if FEATURE_AUTO_DISCOVERY_DISABLE is
        // not disabled.
        .register(MoxyJsonFeature.class)
        .register(JsonMoxyConfigurationContextResolver.class);
```

8.1.2.3. Examples

Jersey provides an JSON MOXy example [https://github.com/jersey/jersey/tree/2.1/examples/json-moxy] on how to use MOXy to consume/produce JSON.

8.1.3. Java API for JSON Processing (JSON-P)

8.1.3.1. Dependency

To use JSON-P as your JSON provider you need to add `jersey-media-json-processing` module to your `pom.xml` file:

```
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-processing</artifactId>
    <version>2.1</version>
</dependency>
```

If you're not using Maven make sure to have all needed dependencies (see jersey-media-json-processing [https://jersey.java.net/project-info/2.1/jersey/project/jersey-media-json-processing/dependencies.html]) on the class-path.

8.1.3.2. Configure and register

As stated in Section 4.1, “Auto-Discoverable Features” JSON-Processing media module is one of the modules where you don't need to explicitly register it's `Feature`s (`JsonProcessingFeature`) in your client/server Configurable [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Configurable.html] as this feature is automatically discovered and registered when you add `jersey-media-json-processing` module to your classpath.

As for the other modules, `jersey-media-json-processing` has also few properties that can affect the registration of `JsonProcessingFeature` (besides CommonProperties.FEATURE_AUTO_DISCOVERY_DISABLE [https://jersey.java.net/apidocs/2.1/

jersey/org/glassfish/jersey/CommonProperties.html#FEATURE_AUTO_DISCOVERY_DISABLE] and the like):

- CommonProperties.JSON_PROCESSING_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/CommonProperties.html#JSON_PROCESSING_FEATURE_DISABLE]
- ServerProperties.JSON_PROCESSING_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#JSON_PROCESSING_FEATURE_DISABLE]
- ClientProperties.JSON_PROCESSING_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html#JSON_PROCESSING_FEATURE_DISABLE]

To configure MessageBodyReader<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyReader.html]s / MessageBodyWriter<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyWriter.html]s provided by JSON-P you can simply add values for supported properties into the Configuration [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Configuration.html] instance (client/server). Currently supported are these properties:

- `JsonGenerator.PRETTY_PRINTING` (`"javax.json.stream.JsonGenerator.prettyPrinting"`)

Example 8.12. Building client with JSON-Processing JSON feature enabled.

```
ClientBuilder.newClient(new ClientConfig()
        // The line bellow that registers JSON-Processing feature can be
        // omitted if FEATURE_AUTO_DISCOVERY_DISABLE is not disabled.
        .register(JsonProcessingFeature.class)
        .property(JsonGenerator.PRETTY_PRINTING, true)
);
```

Example 8.13. Creating JAX-RS application with JSON-Processing JSON feature enabled.

```
// Create JAX-RS application.
final Application application = new ResourceConfig()
        // The line bellow that registers JSON-Processing feature can be
        // omitted if FEATURE_AUTO_DISCOVERY_DISABLE is not disabled.
        .register(JsonProcessingFeature.class)
        .packages("org.glassfish.jersey.examples.jsonp")
        .property(JsonGenerator.PRETTY_PRINTING, true);
```

8.1.3.3. Examples

Jersey provides an JSON Processing example [https://github.com/jersey/jersey/tree/2.1/examples/json-processing-webapp] on how to use JSON-Processing to consume/produce JSON.

8.1.4. Jackson

8.1.4.1. Dependency

To use Jackson as your JSON provider you need to add `jersey-media-json-jackson` module to your `pom.xml` file:

```
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
    <version>2.1</version>
</dependency>
```

If you're not using Maven make sure to have all needed dependencies (see jersey-media-json-jackson [https://jersey.java.net/project-info/2.1/jersey/project/jersey-media-json-jackson/dependencies.html]) on the classpath.

8.1.4.2. Configure and register

Jackson JSON processor could be controlled via providing a custom Jackson ObjectMapper [http://jackson.codehaus.org/1.9.11/javadoc/org/codehaus/jackson/map/ObjectMapper.html] instance. This could be handy if you need to redefine the default Jackson behaviour and to fine-tune how your JSON data structures look like. Detailed description of all Jackson features is out of scope of this guide. The example bellow gives you a hint on how to wire your `ObjectMapper` instance into your Jersey application.

In order to use Jackson as your JSON (JAXB/POJO) provider you need to register JacksonFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/jackson/JacksonFeature.html] and a `ContextResolver<T>` for `ObjectMapper` (if needed) in your Configurable [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Configurable.html] (client/server).

Example 8.14. `ContextResolver<ObjectMapper>`

```
@Provider
public class MyObjectMapperProvider implements ContextResolver<ObjectMapper> {

    final ObjectMapper defaultObjectMapper;
    final ObjectMapper combinedObjectMapper;

    public MyObjectMapperProvider() {
        defaultObjectMapper = createDefaultMapper();
        combinedObjectMapper = createCombinedObjectMapper();
    }

    @Override
    public ObjectMapper getContext(Class<?> type) {
        if (type == CombinedAnnotationBean.class) {
            return combinedObjectMapper;
        } else {
            return defaultObjectMapper;
        }
    }

    private static ObjectMapper createDefaultMapper() {
        final ObjectMapper result = new ObjectMapper();
        result.configure(Feature.INDENT_OUTPUT, true);

        return result;
    }

    // ...
}
```

Example 8.15. Building client with Jackson JSON feature enabled.

```
1 final Client client = ClientBuilder.newBuilder()
2         .register(MyObjectMapperProvider.class)  // No need to register this p
3         .register(JacksonFeature.class)
4         .build();
```

Example 8.16. Creating JAX-RS application with Jackson JSON feature enabled.

```
1 // Create JAX-RS application.
2 final Application application = new ResourceConfig()
3         .packages("org.glassfish.jersey.examples.jackson")
4         .register(MyObjectMapperProvider.class)  // No need to register this p
5         .register(JacksonFeature.class);
```

8.1.4.3. Examples

Jersey provides an JSON Jackson example [https://github.com/jersey/jersey/tree/2.1/examples/json-jackson] on how to use Jackson to consume/produce JSON.

8.1.5. Jettison

JAXB approach for (de)serializing JSON in Jettison module provides, in addition to using pure JAXB, configuration options that could be set on an JettisonConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/jettison/JettisonConfig.html] instance. The instance could be then further used to create a JettisonJaxbContext [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/jettison/JettisonJaxbContext.html], which serves as a main configuration point in this area. To pass your specialized `JettisonJaxbContext` to Jersey, you will finally need to implement a JAXBContext ContextResolver<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/ContextResolver.html] (see below).

8.1.5.1. Dependency

To use Jettison as your JSON provider you need to add `jersey-media-json-jettison` module to your `pom.xml` file:

```
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jettison</artifactId>
    <version>2.1</version>
</dependency>
```

If you're not using Maven make sure to have all needed dependencies (see jersey-media-json-jettison [https://jersey.java.net/project-info/2.1/jersey/project/jersey-media-json-jettison/dependencies.html]) on the classpath.

8.1.5.2. JSON Notations

`JettisonConfig` allows you to use two JSON notations. Each of these notations serializes JSON in a different way. Following is a list of supported notations:

- JETTISON_MAPPED (default notation)

- BADGERFISH

You might want to use one of these notations, when working with more complex XML documents. Namely when you deal with multiple XML namespaces in your JAXB beans.

Individual notations and their further configuration options are described bellow. Rather then explaining rules for mapping XML constructs into JSON, the notations will be described using a simple example. Following are JAXB beans, which will be used.

Example 8.17. JAXB beans for JSON supported notations description, simple address bean

```
 1 @XmlRootElement
 2 public class Address {
 3     public String street;
 4     public String town;
 5
 6     public Address(){}
 7
 8     public Address(String street, String town) {
 9         this.street = street;
10         this.town = town;
11     }
12 }
```

Example 8.18. JAXB beans for JSON supported notations description, contact bean

```
 1 @XmlRootElement
 2 public class Contact {
 3
 4     public int id;
 5     public String name;
 6     public List<Address> addresses;
 7
 8     public Contact() {};
 9
10     public Contact(int id, String name, List<Address> addresses) {
11         this.name = name;
12         this.id = id;
13         this.addresses =
14             (addresses != null) ? new LinkedList<Address>(addresses) : null;
15     }
16 }
```

Following text will be mainly working with a contact bean initialized with:

Example 8.19. JAXB beans for JSON supported notations description, initialization

```
1 Address[] addresses = {new Address("Long Street 1", "Short Village")};
2 Contact contact = new Contact(2, "Bob", Arrays.asList(addresses));
```

I.e. contact bean with `id=2`, `name="Bob"` containing a single address (`street="Long Street 1"`, `town="Short Village"`).

All bellow described configuration options are documented also in api-docs at JettisonConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/jettison/JettisonConfig.html].

8.1.5.2.1. Jettison mapped notation

If you need to deal with various XML namespaces, you will find Jettison `mapped` notation pretty useful. Lets define a particular namespace for `id` item:

```
1 ...
2 @XmlElement(namespace="http://example.com")
3 public int id;
4 ...
```

Then you simply configure a mapping from XML namespace into JSON prefix as follows:

Example 8.20. XML namespace to JSON mapping configuration for Jettison based `mapped` notation

```
1 Map<String,String> ns2json = new HashMap<String, String>();
2 ns2json.put("http://example.com", "example");
3 context = new JettisonJaxbContext(
4     JettisonConfig.mappedJettison().xml2JsonNs(ns2json).build(),
5     types);
```

Resulting JSON will look like in the example bellow.

Example 8.21. JSON expression with XML namespaces mapped into JSON

```
1 {
2    "contact":{
3       "example.id":2,
4       "name":"Bob",
5       "addresses":{
6          "street":"Long Street 1",
7          "town":"Short Village"
8       }
9    }
10 }
```

Please note, that `id` item became `example.id` based on the XML namespace mapping. If you have more XML namespaces in your XML, you will need to configure appropriate mapping for all of them.

Another configurable option introduced in Jersey version 2.2 is related to serialization of JSON arrays with Jettison's mapped notation. When serializing elements representing single item lists/arrays, you might want to utilise the following Jersey configuration method to explicitly name which elements to treat as arrays no matter what the actual content is.

Example 8.22. JSON Array configuration for Jettison based `mapped` notation

```
1 context = new JettisonJaxbContext(
2     JettisonConfig.mappedJettison().serializeAsArray("name").build(),
3     types);
```

Resulting JSON will look like in the example bellow, unimportant lines removed for sanity.

Example 8.23. JSON expression with JSON arrays explicitly configured via Jersey

```
{
   "contact":{
      ...
      "name":["Bob"],
      ...
   }
}
```

8.1.5.2.2. Badgerfish notation

From JSON and JavaScript perspective, this notation is definitely the worst readable one. You will probably not want to use it, unless you need to make sure your JAXB beans could be flawlessly written and read back to and from JSON, without bothering with any formatting configuration, namespaces, etc.

`JettisonConfig` instance using `badgerfish` notation could be built with

```
JettisonConfig.badgerFish().build()
```

and the JSON output JSON will be as follows.

Example 8.24. JSON expression produced using `badgerfish` notation

```
{
   "contact":{
      "id":{
         "$":"2"
      },
      "name":{
         "$":"Bob"
      },
      "addresses":{
         "street":{
            "$":"Long Street 1"
         },
         "town":{
            "$":"Short Village"
         }
      }
   }
}
```

8.1.5.3. Configure and register

In order to use Jettison as your JSON (JAXB/POJO) provider you need to register JettisonFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/jettison/JettisonFeature.html] and a `ContextResolver<T>` for `JAXBContext` (if needed) in your Configurable [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Configurable.html] (client/server).

Example 8.25. `ContextResolver<ObjectMapper>`

```
@Provider
public class JaxbContextResolver implements ContextResolver<JAXBContext> {

    private final JAXBContext context;
    private final Set<Class<?>> types;
    private final Class<?>[] cTypes = {Flights.class, FlightType.class, AircraftTy

    public JaxbContextResolver() throws Exception {
        this.types = new HashSet<Class<?>>(Arrays.asList(cTypes));
        this.context = new JettisonJaxbContext(JettisonConfig.DEFAULT, cTypes);
    }

    @Override
    public JAXBContext getContext(Class<?> objectType) {
        return (types.contains(objectType)) ? context : null;
    }
}
```

Example 8.26. Building client with Jettison JSON feature enabled.

```
final Client client = ClientBuilder.newBuilder()
        .register(JaxbContextResolver.class)  // No need to register this provider
        .register(JettisonFeature.class)
        .build();
```

Example 8.27. Creating JAX-RS application with Jettison JSON feature enabled.

```
// Create JAX-RS application.
final Application application = new ResourceConfig()
        .packages("org.glassfish.jersey.examples.jettison")
        .register(JaxbContextResolver.class)  // No need to register this provider
        .register(JettisonFeature.class);
```

8.1.5.4. Examples

Jersey provides an JSON Jettison example [https://github.com/jersey/jersey/tree/2.1/examples/json-jettison] on how to use Jettison to consume/produce JSON.

8.1.6. `@JSONP` - JSON with Padding Support

Jersey provides out-of-the-box support for JSONP [http://en.wikipedia.org/wiki/JSONP] - JSON with padding. The following conditions has to be met to take advantage of this capability:

- Resource method, which should return wrapped JSON, needs to be annotated with @JSONP [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/JSONP.html] annotation.
- MessageBodyWriter<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyWriter.html] for `application/json` media type, which also accepts the return type of the resource method, needs to be registered (see JSON section of this chapter).
- User's request has to contain `Accept` header with one of the JavaScript media types defined (see below).

Acceptable media types compatible with `@JSONP` are: `application/javascript`, `application/x-javascript`, `application/ecmascript`, `text/javascript`, `text/x-javascript`, `text/ecmascript`, `text/jscript`.

Example 8.28. Simplest case of using `@JSONP`

```
@GET
@JSONP
@Produces({"application/json", "application/javascript"})
public JaxbBean getSimpleJSONP() {
    return new JaxbBean("jsonp");
}
```

Assume that we have registered a JSON providers and that the `JaxbBean` looks like:

Example 8.29. JaxbBean for @JSONP example

```
@XmlRootElement
public class JaxbBean {

    private String value;

    public JaxbBean() {}

    public JaxbBean(final String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }

    public void setValue(final String value) {
        this.value = value;
    }
}
```

When you send a `GET` request with `Accept` header set to `application/javascript` you'll get a result entity that look like:

```
callback({
    "value" : "jsonp",
})
```

There are, of course, ways to configure wrapping method of the returned entity which defaults to `callback` as you can see in the previous example. `@JSONP` has two parameters that can be configured: `callback` and `queryParam`. `callback` stands for the name of the JavaScript callback function defined by the application. The second parameter, `queryParam`, defines the name of the query parameter holding the name of the callback function to be used (if present in the request). Value of `queryParam` defaults to `__callback` so even if you do not set the name of the query parameter yourself, client can always affect the result name of the wrapping JavaScript callback method.

Note

`queryParam` value (if set) always takes precedence over `callback` value.

Lets modify our example a little bit:

Example 8.30. Example of `@JSONP` with configured parameters.

```
@GET
@Produces({"application/json", "application/javascript"})
@JSONP(callback = "eval", queryParam = "jsonpCallback")
public JaxbBean getSimpleJSONP() {
    return new JaxbBean("jsonp");
}
```

And make two requests:

```
curl -X GET http://localhost:8080/jsonp
```

will return

```
eval({
    "value" : "jsonp",
})
```

and the

```
curl -X GET http://localhost:8080/jsonp?jsonpCallback=alert
```

will return

```
alert({
    "value" : "jsonp",
})
```

Example. You can take a look at a provided example available at JSON with Padding example [https://github.com/jersey/jersey/tree/2.1/examples/json-with-padding].

8.2. XML

As you probably already know, Jersey uses `MessageBodyWriter<T>`s and `MessageBodyReader<T>`s to parse incoming requests and create outgoing responses. Every user can create its own representation but... this is not recommended way how to do things. XML is proven standard for interchanging information, especially in web services. Jerseys supports low level data types used for direct manipulation and JAXB XML entities.

8.2.1. Low level XML support

Jersey currently support several low level data types: StreamSource [http://docs.oracle.com/javase/7/docs/api/javax/xml/transform/stream/StreamSource.html], SAXSource [http://docs.oracle.com/javase/7/docs/api/javax/xml/transform/sax/SAXSource.html], DOMSource [http://docs.oracle.com/javase/7/docs/api/javax/xml/transform/dom/DOMSource.html] and Document [http://docs.oracle.com/javase/7/docs/api/org/w3c/dom/Document.html]. You can use these types as the return type or as a method (resource) parameter. Lets say we want to test this feature and we have helloworld example [https://github.com/jersey/jersey/tree/2.1/examples/helloworld] as a starting point. All we need to do is add methods (resources) which consumes and produces XML and types mentioned above will be used.

Example 8.31. Low level XML test - methods added to `HelloWorldResource.java`

```
 1 @POST
 2 @Path("StreamSource")
 3 public StreamSource getStreamSource(StreamSource streamSource) {
 4     return streamSource;
 5 }
 6
 7 @POST
 8 @Path("SAXSource")
 9 public SAXSource getSAXSource(SAXSource saxSource) {
10     return saxSource;
11 }
12
13 @POST
14 @Path("DOMSource")
15 public DOMSource getDOMSource(DOMSource domSource) {
16     return domSource;
17 }
18
19 @POST
20 @Path("Document")
21 public Document getDocument(Document document) {
22     return document;
23 }
```

Both `MessageBodyWriter<T>` and `MessageBodyReader<T>` are used in this case, all we need is a `POST` request with some XML document as a request entity. To keep this as simple as possible only root element with no content will be sent: `"<test />"`. You can create JAX-RS client to do that or use some other tool, for example `curl`:

```
curl -v http://localhost:8080/base/helloworld/StreamSource -d "<test/>"
```

You should get exactly the same XML from our service as is present in the request; in this case, XML headers are added to response but content stays. Feel free to iterate through all resources.

8.2.2. Getting started with JAXB

Good start for people which already have some experience with JAXB annotations is JAXB example [https://github.com/jersey/jersey/tree/2.1/examples/jaxb]. You can see various use-cases there. This text is mainly meant for those who don't have prior experience with JAXB. Don't expect that all possible annotations and their combinations will be covered in this chapter, JAXB (JSR 222 implementation) [http://jaxb.java.net] is pretty complex and comprehensive. But if you just want to know how you can interchange XML messages with your REST service, you are looking at the right chapter.

Lets start with simple example. Lets say we have class `Planet` and service which produces "Planets".

Example 8.32. Planet class

```
 1 @XmlRootElement
 2 public class Planet {
 3     public int id;
 4     public String name;
 5     public double radius;
 6 }
```

Example 8.33. Resource class

```
 1 @Path("planet")
 2 public class Resource {
 3
 4     @GET
 5     @Produces(MediaType.APPLICATION_XML)
 6     public Planet getPlanet() {
 7         final Planet planet = new Planet();
 8
 9         planet.id = 1;
10         planet.name = "Earth";
11         planet.radius = 1.0;
12
13         return planet;
14     }
15 }
```

You can see there is some extra annotation declared on `Planet` class, particularly @XmlRootElement [http://jaxb.java.net/nonav/2.2.7/docs/api/javax/xml/bind/annotation/XmlRootElement.html]. This is an JAXB annotation which maps java classes to XML elements. We don't need to specify anything else, because `Planet` is very simple class and all fields are public. In this case, XML element name will be derived from the class name or you can set the name property: `@XmlRootElement(name="yourName")`.

Our resource class will respond to `GET /planet` with

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<planet>
    <id>1</id>
    <name>Earth</name>
    <radius>1.0</radius>
</planet>
```

which might be exactly what we want... or not. Or we might not really care, because we can use JAX-RS client for making requests to this resource and this is easy as:

```
Planet planet = webTarget.path("planet").request(MediaType.APP
```

There is pre-created `WebTarget` object which points to our applications context root and we simply add path (in our case its `planet`), accept header (not mandatory, but service could provide different content based on this header; for example `text/html` can be served for web browsers) and at the end we specify that we are expecting `Planet` class via `GET` request.

There may be need for not just producing XML, we might want to consume it as well.

Example 8.34. Method for consuming Planet

```
1 @POST
2 @Consumes(MediaType.APPLICATION_XML)
3 public void setPlanet(Planet planet) {
4     System.out.println("setPlanet " + planet);
5 }
```

After valid request is made, service will print out string representation of `Planet`, which can look like `Planet{id=2, name='Mars', radius=1.51}`. With JAX-RS client you can do:

```
webTarget.path("planet").post(planet);
```

If there is a need for some other (non default) XML representation, other JAXB annotations would need to be used. This process is usually simplified by generating java source from XML Schema which is done by `xjc` which is XML to java compiler and it is part of JAXB.

8.2.3. POJOs

Sometimes you can't / don't want to add JAXB annotations to source code and you still want to have resources consuming and producing XML representation of your classes. In this case, JAXBElement [http://jaxb.java.net/nonav/2.2.7/docs/api/javax/xml/bind/JAXBElement.html] class should help you. Let's redo planet resource but this time we won't have an @XmlRootElement [http://jaxb.java.net/nonav/2.2.7/docs/api/javax/xml/bind/annotation/XmlRootElement.html] annotation on `Planet` class.

Example 8.35. Resource class - JAXBElement

```
 1 @Path("planet")
 2 public class Resource {
 3
 4     @GET
 5     @Produces(MediaType.APPLICATION_XML)
 6     public JAXBElement<Planet> getPlanet() {
 7         Planet planet = new Planet();
 8
 9         planet.id = 1;
10         planet.name = "Earth";
11         planet.radius = 1.0;
12
13         return new JAXBElement<Planet>(new QName("planet"), Planet.class, plan
14     }
15
16     @POST
17     @Consumes(MediaType.APPLICATION_XML)
18     public void setPlanet(JAXBElement<Planet> planet) {
19         System.out.println("setPlanet " + planet.getValue());
20     }
21 }
```

As you can see, everything is little more complicated with `JAXBElement`. This is because now you need to explicitly set element name for `Planet` class XML representation. Client side is even more complicated

than server side because you can't do `JAXBElement<Planet>` so JAX-RS client API provides way how to workaround it by declaring subclass of `GenericType<T>`.

Example 8.36. Client side - JAXBElement

```
// GET
GenericType<JAXBElement<Planet>> planetType = new GenericType<JAXBElement<Plan

Planet planet = (Planet) webTarget.path("planet").request(MediaType.APPLICATIO
System.out.println("### " + planet);

// POST
planet = new Planet();

// ...

webTarget.path("planet").post(new JAXBElement<Planet>(new QName("planet"), Pla
```

8.2.4. Using custom JAXBContext

In some scenarios you can take advantage of using custom JAXBContext [http://jaxb.java.net/nonav/2.2.7/docs/api/javax/xml/bind/JAXBContext.html]. Creating `JAXBContext` is an expensive operation and if you already have one created, same instance can be used by Jersey. Other possible use-case for this is when you need to set some specific things to `JAXBContext`, for example to set a different class loader.

Example 8.37. PlanetJAXBContextProvider

```
@Provider
public class PlanetJAXBContextProvider implements ContextResolver<JAXBContext>
    private JAXBContext context = null;

    public JAXBContext getContext(Class<?> type) {
        if (type != Planet.class) {
            return null; // we don't support nothing else than Planet
        }

        if (context == null) {
            try {
                context = JAXBContext.newInstance(Planet.class);
            } catch (JAXBException e) {
                // log warning/error; null will be returned which indicates th
                // provider won't/can't be used.
            }
        }

        return context;
    }
}
```

Sample above shows simple `JAXBContext` creation, all you need to do is put this `@Provider` annotated class somewhere where Jersey can find it. Users sometimes have problems with using provider classes on client side, so just to reminder - you have to declare them in the client config (client does not do anything like package scanning done by server).

Example 8.38. Using Provider with JAX-RS client

```
1
2 ClientConfig config = new ClientConfig();
3 config.register(PlanetJAXBContextProvider.class);
4
5 Client client = ClientBuilder.newClient(config);
6
```

8.2.5. MOXy

If you want to use MOXy [http://www.eclipse.org/eclipselink/moxy.php] as your JAXB implementation instead of JAXB RI you have two options. You can either use the standard JAXB mechanisms to define the `JAXBContextFactory` from which a `JAXBContext` instance would be obtained (for more on this topic, read JavaDoc on JAXBContext [http://jaxb.java.net/nonav/2.2.7/docs/api/javax/xml/bind/JAXBContext.html]) or you can add `jersey-media-moxy` module to your project and register/configure MoxyXmlFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/moxy/xml/MoxyXmlFeature.html] class/instance in the Configurable [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Configurable.html].

Example 8.39. Add `jersey-media-moxy` dependency.

```
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-moxy</artifactId>
    <version>2.1</version>
</dependency>
```

Example 8.40. Register the `MoxyXmlFeature` class.

```
1 final ResourceConfig config = new ResourceConfig()
2     .packages("org.glassfish.jersey.examples.xmlmoxy")
3     .register(MoxyXmlFeature.class);
```

Example 8.41. Configure and register an `MoxyXmlFeature` instance.

```
1 // Configure Properties.
2 final Map<String, Object> properties = new HashMap<String, Object>();
3 // ...
4
5 // Obtain a ClassLoader you want to use.
6 final ClassLoader classLoader = Thread.currentThread().getContextClassLoader()
7
8 final ResourceConfig config = new ResourceConfig()
9     .packages("org.glassfish.jersey.examples.xmlmoxy")
10    .register(new MoxyXmlFeature(
11        properties,
12        classLoader,
13        true, // Flag to determine whether eclipselink-oxm.xml file should be
14        CustomClassA.class, CustomClassB.class  // Classes to be bound.
15    ));
```

8.3. Multipart

8.3.1. Overview

The classes in this module provide an integration of `multipart/*` request and response bodies in a JAX-RS runtime environment. The set of registered providers is leveraged, in that the content type for a body part of such a message reuses the same `MessageBodyReader<T>`/`MessageBodyWriter<T>` implementations as would be used for that content type as a standalone entity.

The following list of general MIME MultiPart features is currently supported:

- The `MIME-Version: 1.0` HTTP header is included on generated responses. It is accepted, but not required, on processed requests.
- A MessageBodyReader<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyReader.html] implementation for consuming MIME MultiPart entities.
- A `MessageBodyWriter<T>` implementation for producing MIME MultiPart entities. The appropriate `@Provider` is used to serialize each body part, based on its media type.
- Optional creation of an appropriate `boundary` parameter on a generated `Content-Type` header, if not already present.

For more information refer to Multi Part [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/multipart/package-info.html].

8.3.1.1. Dependency

To use multipart features you need to add `jersey-media-multipart` module to your `pom.xml` file:

```
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-multipart</artifactId>
    <version>2.1</version>
</dependency>
```

If you're not using Maven make sure to have all needed dependencies (see jersey-media-multipart [https://jersey.java.net/project-info/2.1/jersey/project/jersey-media-multipart/dependencies.html]) on the class-path.

8.3.1.2. Registration

Before you can use capabilities of the `jersey-media-multipart` module in your client/server code, you need to register MultiPartFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/multipart/MultiPartFeature.html].

Example 8.42. Building client with MultiPart feature enabled.

```
final Client client = ClientBuilder.newBuilder()
    .register(MultiPartFeature.class)
    .build();
```

Example 8.43. Creating JAX-RS application with MultiPart feature enabled.

```
// Create JAX-RS application.
final Application application = new ResourceConfig()
    .packages("org.glassfish.jersey.examples.multipart")
    .register(MultiPartFeature.class)
```

8.3.1.3. Examples

Jersey provides an multipart-webapp example [https://github.com/jersey/jersey/tree/2.1/examples/multipart-webapp] on how to use multipart features.

8.3.2. Client

MultiPart [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/multipart/MultiPart.html] class (or it's subclasses) can be used as an entry point to using `jersey-media-multipart` module on the client side. This class represents a MIME multipart message [http://en.wikipedia.org/wiki/MIME#Multipart_messages] and is able to hold an arbitrary number of BodyPart [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/multipart/BodyPart.html]s. Default media type is multipart/mixed [http://en.wikipedia.org/wiki/MIME#Mixed] for `MultiPart` entity and `text/plain` for `BodyPart`.

Example 8.44. `MultiPart` entity

```
final MultiPart multiPartEntity = new MultiPart()
        .bodyPart(new BodyPart().entity("hello"))
        .bodyPart(new BodyPart(new JaxbBean("xml"), MediaType.APPLICATION_XML_TYPE
        .bodyPart(new BodyPart(new JaxbBean("json"), MediaType.APPLICATION_JSON_TY

final WebTarget target = // Create WebTarget.
final Response response = target
        .request()
        .post(Entity.entity(multiPartEntity, multiPartEntity.getMediaType()));
```

If you send a `multiPartEntity` to the server the entity with `Content-Type` header in HTTP message would look like (don't forget to register a JSON provider):

Example 8.45. `MultiPart` entity in HTTP message.

```
Content-Type: multipart/mixed; boundary=Boundary_1_829077776_1369128119878

--Boundary_1_829077776_1369128119878
Content-Type: text/plain

hello
--Boundary_1_829077776_1369128119878
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8" standalone="yes"?><jaxbBean><value>xml</value
--Boundary_1_829077776_1369128119878
Content-Type: application/json

{"value":"json"}
--Boundary_1_829077776_1369128119878--
```

When working with forms (e.g. media type `multipart/form-data`) and various fields in them, there is a more convenient class to be used - FormDataMultiPart [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/multipart/FormDataMultiPart.html]. It automatically sets the media type for the `FormDataMultiPart` entity to `multipart/form-data` and `Content-Disposition` header to `FormDataBodyPart` body parts.

Example 8.46. `FormDataMultiPart` entity

```
final FormDataMultiPart multipart = new FormDataMultiPart()
    .field("hello", "hello")
    .field("xml", new JaxbBean("xml"))
    .field("json", new JaxbBean("json"), MediaType.APPLICATION_JSON_TYPE);

final WebTarget target = // Create WebTarget.
final Response response = target.request().post(Entity.entity(multipart, multipart
```

To illustrate the difference when using `FormDataMultiPart` instead of `FormDataBodyPart` you can take a look at the `FormDataMultiPart` entity from HTML message:

Example 8.47. `FormDataMultiPart` entity in HTTP message.

```
Content-Type: multipart/form-data; boundary=Boundary_1_511262261_1369143433608

--Boundary_1_511262261_1369143433608
Content-Type: text/plain
Content-Disposition: form-data; name="hello"

hello
--Boundary_1_511262261_1369143433608
Content-Type: application/xml
Content-Disposition: form-data; name="xml"

<?xml version="1.0" encoding="UTF-8" standalone="yes"?><jaxbBean><value>xml</value
--Boundary_1_511262261_1369143433608
Content-Type: application/json
Content-Disposition: form-data; name="json"

{"value":"json"}
--Boundary_1_511262261_1369143433608--
```

A common use-case for many users is sending files from client to server. For this purpose you can use classes from `org.glassfish.jersey.jersey.media.multipart` package, such as FileDataBodyPart [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/multipart/file/FileDataBodyPart.html] or StreamDataBodyPart [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/multipart/file/StreamDataBodyPart.html].

Example 8.48. Multipart - sending files.

```
// MediaType of the body part will be derived from the file.
final FileDataBodyPart filePart = new FileDataBodyPart("my_pom", new File("pom.xml

final FormDataMultiPart multipart = new FormDataMultiPart()
    .field("foo", "bar")
    .bodyPart(filePart);

final WebTarget target = // Create WebTarget.
final Response response = target.request()
    .post(Entity.entity(multipart, multipart.getMediaType()));
```

8.3.3. Server

Returning a multipart response from server to client is not much different from the parts described in the client section above. To obtain a multipart entity, sent by a client, in the application you can use two approaches:

- Injecting the whole MultiPart [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/multipart/MultiPart.html] entity.
- Injecting particular parts of a `form-data` multipart request via @FormDataParam [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/multipart/FormDataParam.html] annotation.

8.3.3.1. Injecting and returning the `MultiPart` entity

Working with `MultiPart` types is no different from injecting/returning other entity types. Jersey provides `MessageBodyReader<T>` for reading the request entity and injecting this entity into a method parameter of a resource method and `MessageBodyWriter<T>` for writing output entities. You can expect that either `MultiPart` or `FormDataMultiPart` (`multipart/form-data` media type) object to be injected into a resource method.

Example 8.49. Resource method using `MultiPart` as input parameter / return value.

```
@POST
@Produces("multipart/mixed")
public MultiPart post(final FormDataMultiPart multiPart) {
    return multiPart;
}
```

8.3.3.2. Injecting with `@FormDataParam`

If you just need to bin the named body part(s) of a `multipart/form-data` request entity body to a resource method parameter you can use @FormDataParam [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/multipart/FormDataParam.html] annotation.

This annotation in conjunction with the media type `multipart/form-data` should be used for submitting and consuming forms that contain files, non-ASCII data, and binary data.

The type of the annotated parameter can be one of the following (for more detailed description see javadoc to @FormDataParam [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/multipart/FormDataParam.html]):

- `FormDataBodyPart` - The value of the parameter will be the first named body part or `null` if such a named body part is not present.

- A `List` or `Collection` of `FormDataBodyPart`. The value of the parameter will one or more named body parts with the same name or `null` if such a named body part is not present.

- `FormDataContentDisposition` - The value of the parameter will be the content disposition of the first named body part part or `null` if such a named body part is not present.

- A `List` or `Collection` of `FormDataContentDisposition`. The value of the parameter will one or more content dispositions of the named body parts with the same name or `null` if such a named body part is not present.

- A type for which a message body reader is available given the media type of the first named body part. The value of the parameter will be the result of reading using the message body reader given the type `T`, the media type of the named part, and the bytes of the named body part as input.

  If there is no named part present and there is a default value present as declared by @DefaultValue [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/DefaultValue.html] then the media type will be set to `text/plain`. The value of the parameter will be the result of reading using the message body reader given the type `T`, the media type `text/plain`, and the UTF-8 encoded bytes of the default value as input.

  If there is no message body reader available and the type `T` conforms to a type specified by @FormParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/FormParam.html] then processing is performed as specified by `@FormParam`, where the values of the form parameter are `String` instances produced by reading the bytes of the named body parts utilizing a message body reader for the `String` type and the media type `text/plain`.

  If there is no named part present then processing is performed as specified by `@FormParam`.

Example 8.50. Use of `@FormDataParam` annotation

```
@POST
@Consumes(MediaType.MULTIPART_FORM_DATA_TYPE)
public String postForm(
    @DefaultValue("true") @FormDataParam("enabled") boolean enabled,
    @FormDataParam("data") FileData bean,
    @FormDataParam("file") InputStream file,
    @FormDataParam("file") FormDataContentDisposition fileDisposition) {

    // ...
}
```

In the example above the server consumes a `multipart/form-data` request entity body that contains one optional named body part `enabled` and two required named body parts `data` and `file`.

The optional part `enabled` is processed as a `boolean` value, if the part is absent then the value will be `true`.

The part `data` is processed as a JAXB bean and contains some meta-data about the following part.

The part `file` is a file that is uploaded, this is processed as an `InputStream`. Additional information about the file from the `Content-Disposition` header can be accessed by the parameter `fileDisposition`.

Tip

`@FormDataParam` annotation can be also used on fields.

Chapter 9. Filters and Interceptors

9.1. Introduction

This chapter describes filters, interceptors and their configuration. Filters and interceptors can be used on both sides, on the client and the server side. Filters can modify inbound and outbound requests and responses including modification of headers, entity and other request/response parameters. Interceptors are used primarily for modification of entity input and output streams. You can use interceptors for example to zip and unzip output and input entity streams.

9.2. Filters

Filters can be used when you want to modify any request or response parameters like headers. For example you would like to add a response header "X-Powered-By" to each generated response. Instead of adding this header in each resource method you would use a response filter to add this header.

There are filters on the server side and the client side.

Server filters:

ContainerRequestFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/ContainerRequestFilter.html]
ContainerResponseFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/ContainerResponseFilter.html]
Client filters:

ClientResponseFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientResponseFilter.html]
ClientResponseFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientResponseFilter.html]

9.2.1. Server filters

The following example shows a simple container response filter adding a header to each response.

Example 9.1. Container response filter

```
 1 import java.io.IOException;
 2 import javax.ws.rs.container.ContainerRequestContext;
 3 import javax.ws.rs.container.ContainerResponseContext;
 4 import javax.ws.rs.container.ContainerResponseFilter;
 5 import javax.ws.rs.core.Response;
 6
 7 public class PoweredByResponseFilter implements ContainerResponseFilter {
 8
 9     @Override
10     public void filter(ContainerRequestContext requestContext, ContainerRespon
11         throws IOException {
12
13             responseContext.getHeaders().add("X-Powered-By", "Jersey :-)");
14     }
15 }
```

In the example above the `PoweredByResponseFilter` always adds a header "X-Powered-By" to the response. The filter must inherit from the ContainerResponseFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/ContainerResponseFilter.html] and must be registered as a provider. The filter will be executed for every response which is in most cases after the resource method is executed. Response filters are executed even if the resource method is not run, for example when the resource method is not found and 404 "Not found" response code is returned by the Jersey runtime. In this case the filter will be executed and will process the 404 response.

The `filter()` method has two arguments, the container request and container response. The ContainerRequestContext [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/ContainerRequestContext.html] is accessible only for read only purposes as the filter is executed already in response phase. The modifications can be done in the ContainerResponseContext [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/ContainerResponseContext.html].

The following example shows the usage of a request filter.

Example 9.2. Container request filter

```
 1 import java.io.IOException;
 2 import javax.ws.rs.container.ContainerRequestContext;
 3 import javax.ws.rs.container.ContainerRequestFilter;
 4 import javax.ws.rs.core.Response;
 5 import javax.ws.rs.core.SecurityContext;
 6
 7 public class AuthorizationRequestFilter implements ContainerRequestFilter {
 8
 9     @Override
10     public void filter(ContainerRequestContext requestContext)
11                     throws IOException {
12
13         final SecurityContext securityContext =
14                     requestContext.getSecurityContext();
15         if (securityContext == null ||
16                     !securityContext.isUserInRole("privileged")) {
17
18                 requestContext.abortWith(Response
19                     .status(Response.Status.UNAUTHORIZED)
20                     .entity("User cannot access the resource.")
21                     .build());
22         }
23     }
24 }
```

The request filter is similar to the response filter but does not have access to the ContainerResponseContext as no response is accessible yet. Response filter inherits from ClientResponseFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientResponseFilter.html]. Request filter is executed before the resource method is run and before the response is created. The filter has possibility to manipulate the request parameters including request headers or entity.

The `AuthorizationRequestFilter` in the example checks whether the authenticated user is in the privileged role. If it is not then the request is *aborted* by calling `ContainerRequestContext.abortWith(Response response)` method. The method is intended to be called from the request filter in situation when the request should not be processed further in the standard processing chain. When the `filter` method is finished the response passed as a parameter

to the `abortWith` method is used to respond to the request. Response filters, if any are registered, will be executed and will have possibility to process the aborted response.

9.2.1.1. Pre-matching and post-matching filters

All the request filters shown above was implemented as post-matching filters. It means that the filters would be applied only after a suitable resource method has been selected to process the actual request i.e. after request matching happens. Request matching is the process of finding a resource method that should be executed based on the request path and other request parameters. Since post-matching request filters are invoked when a particular resource method has already been selected, such filters can not influence the resoure method matching process.

To overcome the above described limitation, there is a possibility to mark a server request filter as a *pre-matching* filter, i.e. to annotate the filter class with the @PreMatching [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/PreMatching.html] annotation. Pre-matching filters are request filters that are executed before the request matching is started. Thanks to this, pre-matching request filters have the possibility to influence which method will be matched. Such a pre-matching request filter example is shown here:

Example 9.3. Pre-matching request filter

```
 1 ...
 2 import javax.ws.rs.container.ContainerRequestContext;
 3 import javax.ws.rs.container.ContainerRequestFilter;
 4 import javax.ws.rs.container.PreMatching;
 5 ...
 6
 7 @PreMatching
 8 public class PreMatchingFilter implements ContainerRequestFilter {
 9
10     @Override
11     public void filter(ContainerRequestContext requestContext)
12                         throws IOException {
13         // change all PUT methods to POST
14         if (requestContext.getMethod().equals("PUT")) {
15             requestContext.setMethod("POST");
16         }
17     }
18 }
```

The `PreMatchingFilter` is a simple pre-matching filter which changes all PUT HTTP methods to POST. This might be useful when you want to always handle these PUT and POST HTTP methods with the same Java code. After the `PreMatchingFilter` has been invoked, the rest of the request processing will behave as if the POST HTTP method was originally used. You cannot do this in post-matching filters (standard filters without `@PreMatching` annotation) as the resource method is already matched (selected). An attempt to tweak the original HTTP method in a post-matching filter would cause an `IllegalArgumentException`.

As written above, pre-matching filters can fully influence the request matching process, which means you can even modify request URI in a pre-matching filter by invoking the `setRequestUri(URI)` method of `ContainerRequestFilter` so that a different resource would be matched.

Like in post-matching filters you can abort a response in pre-matching filters too.

9.2.2. Client fillers

Client filters are similar to container filters. The response can also be aborted in the ClientRequestFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientRequestFilter.html] which would cause that no request will actually be sent to the server at all. A new response is passed to the `abort` method. This response will be used and delivered as a result of the request invocation. Such a response goes through the client response filters. This is similar to what happens on the server side. The process is shown in the following example:

Example 9.4. Client request filter

```
 1 public class CheckRequestFilter implements ClientRequestFilter {
 2
 3     @Override
 4     public void filter(ClientRequestContext requestContext)
 5                         throws IOException {
 6         if (requestContext.getHeaders(
 7                         ).get("Client-Name") == null) {
 8             requestContext.abortWith(
 9                         Response.status(Response.Status.BAD_REQUEST)
10                 .entity("Client-Name header must be defined.")
11                         .build());
12          }
13     }
14 }
```

The `CheckRequestFilter` validates the outgoing request. It is checked for presence of a `Client-Name` header. If the header is not present the request will be aborted with a made up response with an appropriate code and message in the entity body. This will cause that the original request will not be effectivelly sent to the server but the actual invocation will still end up with a response as if it would be generated by the server side. If there would be any client response filter it would be executed on this response.

To summarize the workflow, for any client request invoked from the client API the client request filters (ClientRequestFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientRequestFilter.html]) are executed that could manipulate the request. If not aborted, the outcoming request is then physically sent over to the server side and once a response is received back from the server the client response filters (ClientResponseFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientResponseFilter.html]) are executed that might again manipulate the returned response. Finally the response is passed back to the code that invoked the request. If the request was aborted in any client request filter then the client/server communication is skipped and the aborted response is used in the response filters.

9.3. Interceptors

Interceptors share a common API for the server and the client side. Whereas filters are primarily intended to manipulate request and response parameters like HTTP headers, URIs and/or HTTP methods, interceptors are intended to manipulate entities, via manipulating entity input/output streams. If you for example need to encode entity body of a client request then you could implement an interceptor to do the work for you.

There are two kinds of interceptors, ReaderInterceptor [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/ReaderInterceptor.html] and WriterInterceptor [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/WriterInterceptor.html]. Reader interceptors are used to manipulate inbound

entity streams. These are the streams coming from the "wire". So, using a reader interceptor you can manipulate request entity stream on the server side (where an entity is read from the client request) and response entity stream on the client side (where an entity is read from the server response). Writer interceptors are used for cases where entity is written to the "wire" which on the server means when writing out a response entity and on the client side when writing request entity for a request to be sent out to the server. Writer and reader interceptors are executed before message body readers or writers are executed and their primary intention is to wrap the entity streams that will be used in message body reader and writers.

The following example shows a writer interceptor that enables GZIP compression of the whole entity body.

Example 9.5. GZIP writer interceptor

```
 1 public class GZIPWriterInterceptor implements WriterInterceptor {
 2
 3     @Override
 4     public void aroundWriteTo(WriterInterceptorContext context)
 5                     throws IOException, WebApplicationException {
 6         final OutputStream outputStream = context.getOutputStream();
 7         context.setOutputStream(new GZIPOutputStream(outputStream));
 8         context.proceed();
 9     }
10 }
```

The interceptor gets a output stream from the WriterInterceptorContext [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/WriterInterceptorContext.html] and sets a new one which is a GZIP wrapper of the original output stream. After all interceptors are executed the output stream lastly set to the `WriterInterceptorContext` will be used for serialization of the entity. In the example above the entity bytes will be written to the GZIPOutputStream which will compress the stream data and write them to the original output stream. The original stream is always the stream which writes the data to the "wire". When the interceptor is used on the server, the original output stream is the stream into which writes data to the underlying server container stream that sends the response to the client.

The interceptors wrap the streams and they itself work as wrappers. This means that each interceptor is a wrapper of another interceptor and it is responsibility of each interceptor implementation to call the wrapped interceptor. This is achieved by calling the `proceed()` method on the `WriterInterceptorContext`. This method will call the next registered interceptor in the chain, so effectivelly this will call all remaining registered interceptors. Calling `proceed()` from the last interceptor in the chain will call the appropriate message body reader. Therefore every interceptor must call the `proceed()` method otherwise the entity would not be written. The wrapping principle is reflected also in the method name, aroundWriteTo, which says that the method is wrapping the writing of the entity.

The method aroundWriteTo() gets `WriterInterceptorContext` as a parameter. This context contains getters and setters for header parameters, request properties, entity, entity stream and other properties. These are the properties which will be passed to the final `MessageBodyWriter<T>`. Interceptors are allowed to modify all these properties. This could influence writing of an entity by `MessageBodyWriter<T>` and even selection of such a writer. By changing media type (`WriterInterceptorContext`.setMediaType()) the interceptor can cause that different message body writer will be chosen. The interceptor can also completely replace the entity if it is needed. However, for modification of headers, request properties and such, the filters are usually more preferable choice. Interceptors are executed only when there is any entity and when the entity is to be written. So, when you always want to add a new header to a response no matter what, use filters as interceptors might not be executed when no entity is present. Interceptors should modify properties only for entity serialization and deserialization purposes.

Let's now look at an example of a `WriterInterceptor`

Example 9.6. GZIP reader interceptor

```
 1 public class GZIPReaderInterceptor implements ReaderInterceptor {
 2
 3     @Override
 4     public Object aroundReadFrom(ReaderInterceptorContext context)
 5                     throws IOException, WebApplicationException {
 6         final InputStream originalInputStream = context.getInputStream();
 7         context.setInputStream(new GZIPInputStream(originalInputStream));
 8         return context.proceed();
 9     }
10 }
```

The `GZIPReaderInterceptor` wraps the original input stream with the `GZIPInputStream`. All further reads from the entity stream will cause that data will be decompressed by this stream. The interceptor method `aroundReadFrom()` must return an entity. The entity is returned from the `proceed` method of the ReaderInterceptorContext [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/ReaderInterceptorContext.html]. The `proceed` method internally calls the wrapped interceptor which must also return an entity. The `proceed` method invoked from the last interceptor in the chain calls message body reader which deserializes the entity end returns it. Every interceptor can change this entity if there is a need but in the most cases interceptors will just return the entity as returned from the `proceed` method.

As already mentioned above, interceptors should be primarily used to manipulate entity body. Similar to methods exposed by `WriterInterceptorContext` the `ReaderInterceptorContext` introduces a set of methods for modification of request/response properties like HTTP headers, URIs and/or HTTP methods (excluding getters and setters for entity as entity has not been read yet). Again the same rules as for `WriterInterceptor` applies for changing these properties (change only properties in order to influence reading of an entity).

9.4. Filter and interceptor execution order

Let's look closer at the context of execution of filters and interceptors. The following steps describes scenario where a JAX-RS client makes a POST request to the server. The server receives an entity and sends a response back with the same entity. GZIP reader and writer interceptors are registered on the client and the server. Also filters are registered on client and server which change the headers of request and response.

1. Client request invoked: The POST request with attached entity is built on the client and invoked.

2. ClientRequestFilters: The ClientResponseFilters are executed on the client and they manipulate the request headers.

3. Client `WriterInterceptor`: As the request contains an entity, writer interceptor registered on the client is executed before a MessageBodyWriter is executed. It wraps the entity output stream with the GZipOutputStream.

4. Client MessageBodyWriter: message body writer is executed on the client which serializes the entity into the new GZipOutput stream. This stream zips the data and sends it to the "wire".

5. Server: server receives a request. Data of entity is compressed which means that pure read from the entity input stream would return compressed data.

6. Server pre-matching ContainerRequestFilters: ContainerRequestFilters are executed that can manipulate resource method matching process.

7. Server: matching: resource method matching is done.

8. Server: post-matching ContainerRequestFilters: ContainerRequestFilters post matching filters are executed. This include execution of all global filters (without name binding) and filters name-bound to the matched method.

9. Server `ReaderInterceptor`: reader interceptors are executed on the server. The GZIPReaderInterceptor wraps the input stream (the stream from the "wire") into the GZipInputStream and set it to context.

10. Server MessageBodyReader: server message body reader is executed and it deserializes the entity from new GZipInputStream (get from the context). This means the reader will read unzipped data and not the compressed data from the "wire".

11. Server resource method is executed: the deserialized entity object is passed to the matched resource method as a parameter. The method returns this entity as a response entity.

12. Server ContainerResponseFilters are executed: response filters are executed on the server and they manipulate the response headers. This include all global bound filters (without name binding) and all filters name-bound to the resource method.

13. Server `WriterInterceptor`: is executed on the server. It wraps the original output stream with a new GZIPOuptutStream. The original stream is the stream that "goes to the wire" (output stream for response from the underlying server container).

14. Server MessageBodyWriter: message body writer is executed on the server which serializes the entity into the GZIPOutputStream. This stream compresses the data and writes it to the original stream which sends this compressed data back to the client.

15. Client receives the response: the response contains compressed entity data.

16. Client ClientResponseFilters: client response filters are executed and they manipulate the response headers.

17. Client response is returned: the javax.ws.rs.core.Response is returned from the request invocation.

18. Client code calls response.readEntity(): read entity is executed on the client to extract the entity from the response.

19. Client `ReaderInterceptor`: the client reader interceptor is executed when readEntity(Class) is called. The interceptor wraps the entity input stream with GZIPInputStream. This will decompress the data from the original input stream.

20. Client MessageBodyReaders: client message body reader is invoked which reads decompressed data from GZIPInputStream and deserializes the entity.

21. Client: The entity is returned from the readEntity().

It is worth to mention that in the scenario above the reader and writer interceptors are invoked only if the entity is present (it does not make sense to wrap entity stream when no entity will be written). The same behaviour is there for message body readers and writers. As mentioned above, interceptors are executed before the message body reader/writer as a part of their execution and they can wrap the input/output stream before the entity is read/written. There are exceptions when interceptors are not run before message body reader/writers but this is not the case of simple scenario above. This happens for example when the entity is read many times from client response using internal buffering. Then the data are intercepted only once and kept 'decoded' in the buffer.

9.5. Name binding

Filters and interceptors can be *name-bound*. Name binding is a concept that allows to say to a JAX-RS runtime that a specific filter or interceptor will be executed only for a specific resource method. When a filter or an interceptor is limited only to a specific resource method we say that it is *name-bound*. Filters and interceptors that do not have such a limitation are called *global*.

Filter or interceptor can be assigned to a resource method using the @NameBinding [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/NameBinding.html] annotation. The annotation is used as meta annotation for other user implemented annotations that are applied to a providers and resource methods. See the following example:

Example 9.7. `@NameBinding` example

```
 1 ...
 2 import java.lang.annotation.Retention;
 3 import java.lang.annotation.RetentionPolicy;
 4 import java.util.zip.GZIPInputStream;
 5
 6 import javax.ws.rs.GET;
 7 import javax.ws.rs.NameBinding;
 8 import javax.ws.rs.Path;
 9 import javax.ws.rs.Produces;
10 ...
11
12
13 // @Compress annotation is the name binding annotation
14 @NameBinding
15 @Retention(RetentionPolicy.RUNTIME)
16 public @interface Compress {}
17
18
19 @Path("helloworld")
20 public class HelloWorldResource {
21
22     @GET
23     @Produces("text/plain")
24     public String getHello() {
25         return "Hello World!";
26     }
27
28     @GET
29     @Path("too-much-data")
30     @Compress
31     public String getVeryLongString() {
32         String str = ... // very long string
33         return str;
34     }
35 }
36
37 // interceptor will be executed only when resource methods
38 // annotated with @Compress annotation will be executed
39 @Compress
40 public class GZIPWriterInterceptor implements WriterInterceptor {
41     @Override
42     public void aroundWriteTo(WriterInterceptorContext context)
43                       throws IOException, WebApplicationException {
44         final OutputStream outputStream = context.getOutputStream();
45         context.setOutputStream(new GZIPOutputStream(outputStream));
46         context.proceed();
47     }
48 }
```

The example above defines a new `@Compress` annotation which is a name binding annotation as it is annotated with `@NameBinding`. The `@Compress` is applied on the resource method `getVeryLongString()` and on the interceptor `GZIPWriterInterceptor`. The interceptor will

be executed only if any resource method with such a annotation will be executed. In our example case the interceptor will be executed only for the `getVeryLongString()` method. The interceptor will not be executed for method `getHello()`. In this example the reason is probably clear. We would like to compress only long data and we do not need to compress the short response of "Hello World!".

Name binding can be applied on a resource class. In the example `HelloWorldResource` would be annotated with `@Compress`. This would mean that all resource methods will use compression in this case.

There might be many name binding annotations defined in an application. When any provider (filter or interceptor) is annotated with more than one name binding annotation, then it will be executed for resource methods which contain ALL these annotations. So, for example if our interceptor would be annotated with another name binding annotation @GZIP then the resource method would need to have both annotations attached, @Compress and @GZIP, otherwise the interceptor would not be executed. Based on the previous paragraph we can even use the combination when the resource method `getVeryLongString()` would be annotated with @Compress and resource class `HelloWorldResource` would be annotated from with @GZIP. This would also trigger the interceptor as annotations of resource methods are aggregated from resource method and from resource class. But this is probably just an edge case which will not be used so often.

Note that *global filters are executed always*, so even for resource methods which have any name binding annotations.

9.6. Dynamic binding

Dynamic binding is a way how to assign filters and interceptors to the resource methods in a dynamic manner. Name binding from the previous chapter uses a static approach and changes to binding require source code change and recompilation. With dynamic binding you can implement code which defines bindings during the application initialization time. The following example shows how to implement dynamic binding.

Example 9.8. Dynamic binding example

```
 1 ...
 2 import javax.ws.rs.core.FeatureContext;
 3 import javax.ws.rs.container.DynamicFeature;
 4 ...
 5
 6 @Path("helloworld")
 7 public class HelloWorldResource {
 8
 9     @GET
10     @Produces("text/plain")
11     public String getHello() {
12         return "Hello World!";
13     }
14
15     @GET
16     @Path("too-much-data")
17     public String getVeryLongString() {
18         String str = ... // very long string
19         return str;
20     }
21 }
22
23 // This dynamic binding provider registers GZIPWriterInterceptor
24 // only for HelloWorldResource and methods that contain
25 // "VeryLongString" in their name. It will be executed during
26 // application initialization phase.
27 public class CompressionDynamicBinding implements DynamicFeature {
28
29     @Override
30     public void configure(ResourceInfo resourceInfo, FeatureContext context) {
31         if (HelloWorldResource.class.equals(resourceInfo.getResourceClass())
32                 && resourceInfo.getResourceMethod()
33                     .getName().contains("VeryLongString")) {
34             context.register(GZIPWriterInterceptor.class);
35         }
36     }
37 }
```

The example contains one `HelloWorldResource` which is known from the previous name binding example. The difference is in the `getVeryLongString` method, which now does not define the `@Compress` name binding annotations. The binding is done using the provider which implements DynamicFeature [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/DynamicFeature.html] interface. The interface defines one `configure` method with two arguments, `ResourceInfo` and `FeatureContext`. `ResourceInfo` contains information about the resource and method to which the binding can be done. The `configure` method will be executed once for each resource method that is defined in the application. In the example above the provider will be executed twice, once for the `getHello()` method and once for `getVeryLongString()` (once the resourceInfo will contain information about getHello() method and once it will point to getVeryLongString()). If a dynamic binding provider wants to register any provider for the actual resource method it will do that using provided `FeatureContext` which extends JAX-RS `Configurable` API. All methods for registration of filter or interceptor classes or instances can be used. Such dynamically registered filters or interceptors will be bound only to the actual resource method. In the example above the `GZIPWriterInterceptor` will

be bound only to the method `getVeryLongString()` which will cause that data will be compressed only for this method and not for the method `getHello()`. The code of `GZIPWriterInterceptor` is in the examples above.

Note that filters and interceptors registered using dynamic binding are only additional filters run for the resource method. If there are any name bound providers or global providers they will still be executed.

9.7. Priorities

In case you register more filters and interceptors you might want to define an exact order in which they should be invoked. The order can be controlled by the `@Priority` annotation defined by the `javax.annotation.Priority` class. The annotation accepts an integer parameter of priority. Providers used in request processing (ContainerRequestFilter, ClientRequestFilter, ReaderInterceptors) are sorted based on the priority in an ascending manner. So, a request filter with priority defined with `@Priority(1000)` will be executed before another request filter with priority defined as `@Priority(2000)`. Providers used during response processing (ContainerResponseFilter, ClientResponseFilter, WriterIntercepors) are executed in the reverse order (using descending manner), so a provider with the priority defined with `@Priority(2000)` will be executed before another provider with priority defined with `@Priority(1000)`.

It's a good practice to assign a priority to filters and interceptors. Use Priorities [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Priorities.html] class which defines standardized priorities in JAX-RS for different usages, rather than inventing your own priorities. So, when you for example write an authentication filter you would assign a priority 1000 which is the value of `Priorities.AUTHENTICATION`. The following example shows the filter from the beginning of this chapter with a priority assigned.

Example 9.9. Priorities example

```
 1 ...
 2 import javax.annotation.Priority;
 3 import javax.ws.rs.Priorities;
 4 ...
 5
 6 @Priority(Priorities.HEADER_DECORATOR)
 7 public class ResponseFilter implements ContainerResponseFilter {
 8
 9     @Override
10     public void filter(ContainerRequestContext requestContext,
11                        ContainerResponseContext responseContext)
12                        throws IOException {
13
14         responseContext.getHeaders().add("X-Powered-By", "Jersey :-)");
15     }
16 }
```

As this is a response filter and response filters are executed in the reverse order, any other filter with priority lower than 3000 (`Priorities.HEADER_DECORATOR` is 3000) will be executed after this filter. So, for example `AUTHENTICATION` filter (priority 1000) would be run after this filter.

Chapter 10. Asynchronous Services and Clients

This chapter describes the usage of asynchronous API on the client and server side. The term *async* will be sometimes used interchangeably with the term *asynchronous* in this chapter.

10.1. Asynchronous Server API

Request processing on the server works by default in a synchronous processing mode, which means that a client connection of a request is processed in a single I/O container thread. Once the thread processing the request returns to the I/O container, the container can safely assume that the request processing is finished and that the client connection can be safely released including all the resources associated with the connection. This model is typically sufficient for processing of requests for which the processing resource method execution takes a relatively short time. However, in cases where a resource method execution is known to take a long time to compute the result, server-side asynchronous processing model should be used. In this model, the association between a request processing thread and client connection is broken. I/O container that handles incoming request may no longer assume that a client connection can be safely closed when a request processing thread returns. Instead a facility for explicitly suspending, resuming and closing client connections needs to be exposed. Note that the use of server-side asynchronous processing model will not improve the request processing time perceived by the client. It will however increase the throughput of the server, by releasing the initial request processing thread back to the I/O container while the request may still be waiting in a queue for processing or the processing may still be running on another dedicated thread. The released I/O container thread can be used to accept and process new incoming request connections.

The following example shows a simple asynchronous resource method defined using the new JAX-RS async API:

Example 10.1. Simple async resource

```
 1 @Path("/resource")
 2 public class AsyncResource {
 3     @GET
 4     public void asyncGet(@Suspended final AsyncResponse asyncResponse) {
 5
 6         new Thread(new Runnable() {
 7             @Override
 8             public void run() {
 9                 String result = veryExpensiveOperation();
10                 asyncResponse.resume(result);
11             }
12
13             private String veryExpensiveOperation() {
14                 // ... very expensive operation
15             }
16         }).start();
17     }
18 }
```

In the example above, a resource `AsyncResource` with one `GET` method `asyncGet` is defined. The `asyncGet` method injects a JAX-RS AsyncResponse [http://jax-rs-spec.java.net/nonav/2.0/

apidocs/javax/ws/rs/container/AsyncResponse.html] instance using a JAX-RS @Suspended [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/Suspended.html] annotation. Please note that `AsyncResponse` must be injected by the `@Suspended` annotation and not by @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html] as `@Suspended` does not only inject response but also says that the method is executed in the asynchronous mode. By the `AsyncResponse` parameter into a resource method we tell the Jersey runtime that the method is supposed to be invoked using the asynchronous processing mode, that is the client connection should not be automatically closed by the underlying I/O container when the method returns. Instead, the injected `AsyncResponse` instance (that represents the suspended client request connection) will be used to explicitly send the response back to the client using some other thread. In other words, Jersey runtime knows that when the `asyncGet` method completes, the response to the client may not be ready yet and the processing must be suspended and wait to be explictly resumed with a response once it becomes available. Note that the method `asyncGet` returns `void` in our example. This is perfectly valid in case of an asynchronous JAX-RS resource method, even for a @GET [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/GET.html] method, as the response is never returned directly from the resource method as its return value. Instead, the response is later returned using `AsyncResponse` instance as it is demonstrated in the example. The `asyncGet` resource method starts a new thread and exits from the method. In that state the request processing is suspended and the container thread (the one which entered the resource method) is returned back to the container's thread pool and it can process other requests. New thread started in the resource method may execute an expensive operation which might take a long time to finish. Once a result is ready it is resumed using the `resume()` method on the `AsyncResponse` instance. The resumed response is then processed in the new thread by Jersey in a same way as any other synchronous response, including execution of filters and interceptors, use of exception mappers as necessary and sending the response back to the client.

It is important to note that the asynchronous response (`asyncResponse` in the example) does not need to be resumed from the thread started from the resource method. The asynchronous response can be resumed even from different request processing thread as it is shown in the the example of the AsyncResponse [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/AsyncResponse.html] javadoc. In the javadoc example the async response suspended from the `GET` method is resumed later on from the `POST` method. The suspended async response is passed between requests using a static field and is resumed from the other resource method running on a different request processing thread.

Imagine now a situation when there is a long delay between two requests and you would not like to let the client wait for the response "forever" or at least for an unacceptable long time. In asynchronous processing model, occurrences of such situations should be carefully considered with client connections not being automatically closed when the processing method returns and the response needs to be resumed explicitly based on an event that may actually even never happen. To tackle these situations asynchronous *timeouts* can be used.

The following example shows the usage of timeouts:

Example 10.2. Simple async method with timeout

```
@GET
public void asyncGetWithTimeout(@Suspended final AsyncResponse asyncResponse)
    asyncResponse.setTimeoutHandler(new TimeoutHandler() {

        @Override
        public void handleTimeout(AsyncResponse asyncResponse) {
            asyncResponse.resume(Response.status(Response.Status.SERVICE_UNAVA
                    .entity("Operation time out.").build());
        }
    });
    asyncResponse.setTimeout(20, TimeUnit.SECONDS);

    new Thread(new Runnable() {

        @Override
        public void run() {
            String result = veryExpensiveOperation();
            asyncResponse.resume(result);
        }

        private String veryExpensiveOperation() {
            // ... very expensive operation that typically finishes within 20
        }
    }).start();
}
```

By default, there is no timeout defined on the suspended `AsyncResponse` instance. A custom timeout and timeout event handler may be defined using `setTimeoutHandler(TimeoutHandler)` and `setTimeout(long, TimeUnit)` methods. The `setTimeoutHandler(TimeoutHandler)` method defines the handler that will be invoked when timeout is reached. The handler resumes the response with the response code 503 (from Response.Status.`SERVICE_UNAVAILABLE`). A timeout interval can be also defined without specifying a custom timeout handler (using just the `setTimeout(long, TimeUnit)` method). In such case the default behaviour of Jersey runtime is to throw a `ServiceUnavailableException` that gets mapped into 503, "Service Unavailable" HTTP error response, as defined by the JAX-RS specification.

10.1.1. Asynchronous Server-side Callbacks

As operations in asynchronous cases might take long time and they are not always finished within a single resource method invocation, JAX-RS offers facility to register callbacks to be invoked based on suspended async response state changes. In Jersey you can register two JAX-RS callbacks:

- CompletionCallback [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/CompletionCallback.html] that is executed when request finishes or fails, and
- ConnectionCallback [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/ConnectionCallback.html] executed when a connection to a client is closed or lost.

Example 10.3. CompletionCallback example

```
 1 @Path("/resource")
 2 public class AsyncResource {
 3     private static int numberOfSuccessResponses = 0;
 4     private static int numberOfFailures = 0;
 5     private static Throwable lastException = null;
 6
 7     @GET
 8     public void asyncGetWithTimeout(@Suspended final AsyncResponse asyncRespon
 9         asyncResponse.register(new CompletionCallback() {
10             @Override
11             public void onComplete(Throwable throwable) {
12                 if (throwable == null) {
13                     // no throwable - the processing ended successfully
14                     // (response already written to the client)
15                     numberOfSuccessResponses++;
16                 } else {
17                     numberOfFailures++;
18                     lastException = throwable;
19                 }
20             }
21         });
22
23         new Thread(new Runnable() {
24             @Override
25             public void run() {
26                 String result = veryExpensiveOperation();
27                 asyncResponse.resume(result);
28             }
29
30             private String veryExpensiveOperation() {
31                 // ... very expensive operation
32             }
33         }).start();
34     }
35 }
```

A completion callback is registered using `register(...)` method on the `AsyncResponse` instance. A registered completion callback is bound only to the response(s) to which it has been registered. In the example the `CompletionCallback` is used to calculate successfully processed responses, failures and to store last exception. This is only a simple case demonstrating the usage of the callback. You can use completion callback to release the resources, change state of internal resources or representations or handle failures. The method has an argument `Throwable` which is set only in case of an error. Otherwise the parameter will be `null`, which means that the response was successfully written. The callback is executed only after the response is written to the client (not immediately after the response is resumed).

The `AsyncResponse` `register(...)` method is overloaded and offers options to register a single callback as an `Object` (in the example), as a `Class` or multiple callbacks using varags.

As some async requests may take long time to process the client may decide to terminate it's connection to the server before the response has been resumed or before it has been fully written to the client. To deal with these use cases a `ConnectionCallback` can be used. This callback will be executed only if the connection was prematurely terminated or lost while the response is being written to the back client. Note

that this callback will not be invoked when a response is written successfully and the client connection is closed as expected. See javadoc of ConnectionCallback [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/ConnectionCallback.html] for more information.

10.1.2. Chunked Output

Jersey offers a facility for sending response to the client in multiple more-or-less independent chunks using a *chunked output*. Each response chunk usually takes some (longer) time to prepare before sending it to the client. The most important fact about response chunks is that you want to send them to the client immediately as they become available without waiting for the remaining chunks to become available too. The first bytes of each chunked response consists of the HTTP headers that are sent to the client. The size -1 is set in the response `Content-Length` header to indicate that the response is chunked. As noted above, the entity of the response is then sent in chunks as they become available. Client knows that the response is going to be chunked, so it reads each chunk of the response separately, processes it, and waits for more chunks to arrive on the same connection. After some time, the server generates another response chunk and send it again to the client. Server keeps on sending response chunks until it closes the connection after sending the last chunk when the response processing is finished.

In Jersey you can use ChunkedOutput [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ChunkedOuptut.html] to send response to a client in chunks. Chunks are strictly defined pieces of a response body can be marshalled as a separate entities using Jersey/JAX-RS MessageBodyWriter<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyWriter.html] providers. A chunk can be String, Long or JAXB bean serialized to XML or JSON or any other dacustom type for which a `MessageBodyWriter<T>` is available.

The resource method that returns `ChunkedOutput` informs the Jersey runtime that the response will be chunked and that the processing works asynchronously as such. You do not need to inject `AsyncResponse` to start the asynchronous processing mode in this case. Returning a `ChunkedOutput` instance from the method is enough to indicate the asynchronous processing. Response headers will be sent to a client when the resource method returns and the client will wait for the stream of chunked data which you will be able to write from different thread using the same `ChunkedOutput` instance returned from the resource method earlier. The following example demonstrates this use case:

Example 10.4. ChunkedOutput example

```
 1 @Path("/resource")
 2 public class AsyncResource {
 3     @GET
 4     public ChunkedOutput<String> getChunkedResponse() {
 5         final ChunkedOutput<String> output = new ChunkedOutput<String>(String.
 6
 7         new Thread() {
 8             public void run() {
 9                 try {
10                     String chunk;
11
12                     while ((chunk = getNextString()) != null) {
13                         output.write(chunk);
14                     }
15                 } catch (IOException e) {
16                     // IOException thrown when writing the
17                     // chunks of response: should be handled
18                 } finally {
19                     output.close();
20                         // simplified: IOException thrown from
21                         // this close() should be handled here...
22                 }
23             }
24         }.start();
25
26         // the output will be probably returned even before
27         // a first chunk is written by the new thread
28         return output;
29     }
30
31     private String getNextString() {
32         // ... long running operation that returns
33         //     next string or null if no other string is accessible
34     }
35 }
```

The example above defines a `GET` method that returns a `ChunkedOutput` instance. The generic type of `ChunkedOutput` defines the chunk types (in this case chunks are Strings). Before the instance is returned a new thread is started that writes individual chunks into the chunked output instance named `output`. Once the original thread returns from the resource method, Jersey runtime writes headers to the container response but does not close the client connection yet and waits for the response data to be written to the chunked `output`. New thread in a loop calls the method `getNextString()` which returns a next String or `null` if no other String exists (the method could for example load latest data from the database). Returned Strings are written to the chunked `output`. Such a written chunks are internally written to the container response and client can read them. At the end the chunked output is closed which determines the end of the chunked response. Please note that you must close the output explicitly in order to close the client connection as Jersey does not implicitly know when you are finished with writing the chunks.

A chunked output can be processed also from threads created from another request as it is explained in the sections above. This means that one resource method may e.g. only return a `ChunkedOutput` instance and other resource method(s) invoked from another request thread(s) can write data into the chunked output and/or close the chunked response.

10.2. Client API

The client API supports asynchronous processing too. Simple usage of asynchronous client API is shown in the following example:

Example 10.5. Simple client async invocation

```
1 final AsyncInvoker asyncInvoker = target().path("http://example.com/resource/"
2         .request().async();
3 final Future<Response> responseFuture = asyncInvoker.get();
4 System.out.println("Request is being processed asynchronously.");
5 final Response response = responseFuture.get();
6     // get() waits for the response to be ready
7 System.out.println("Response received.");
```

The difference against synchronous invocation is that the http method call `get()` is not called on SyncInvoker [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/SyncInvoker.html] but on AsyncInvoker [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/AsyncInvoker.html]. The `AsyncInvoker` is returned from the call of method `Invocation.Builder.async()` as shown above. `AsyncInvoker` offers methods similar to `SyncInvoker` only these methods do not return a response synchronously. Instead a `Future<...>` representing response data is returned. These method calls also return immediately without waiting for the actual request to complete. In order to get the response of the invoked `get()` method, the `responseFuture.get()` is invoked which waits for the response to be finished (this call is blocking as defined by the Java SE `Future` contract).

Asynchronous Client API in JAX-RS is fully integrated in the fluent JAX-RS Client API flow, so that the async client-side invocations can be written fluently just like in the following example:

Example 10.6. Simple client fluent async invocation

```
1 final Future<Response> responseFuture = target().path("http://example.com/reso
2         .request().async().get();
```

To work with asynchronous results on the client-side, all standard `Future` API facilities can be used. For example, you can use the `isDone()` method to determine whether a response has finished to avoid the use of a blocking call to `Future.get()`.

10.2.1. Asynchronous Client Callbacks

Similarly to the server side, in the client API you can register asynchronous callbacks too. You can use these callbacks to be notified when a response arrives instead of waiting for the response on `Future.get()` or checking the status by `Future.isDone()` in a loop. A client-side asynchronous invocation callback can be registered as shown in the following example:

Example 10.7. Client async callback

```
final Future<Response> responseFuture = target().path("http://example.com/reso
        .request().async().get(new InvocationCallback<Response>() {
            @Override
            public void completed(Response response) {
                System.out.println("Response status code "
                        + response.getStatus() + " received.");
            }

            @Override
            public void failed(Throwable throwable) {
                System.out.println("Invocation failed.");
                throwable.printStackTrace();
            }
        });
```

The registered callback is expected to implement the InvocationCallback [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/InvocationCallback.html] interface that defines two methods. First method `completed(Response)` gets invoked when an invocation successfully finishes. The result response is passed as a parameter to the callback method. The second method `failed(Throwable)` is invoked in case the invocation fails and the exception describing the failure is passed to the method as a parameter. In this case since the callback generic type is `Response`, the `failed(Throwable)` method would only invoked in case the invocation fails because of an internal client-side processing error. It would not be invoked in case a server responds with an HTTP error code, for example if the requested resource is not found on the server and HTTP `404` response code is returned. In such case `completed(Response)` callback method would be invoked and the response passed to the method would contain the returned error response with HTTP `404` error code. This is a special behavior in case the generic callback return type is `Response`. In the next example an exception is thrown (or `failed(Throwable)` method on the invocation callback is invoked) even in case a non-`2xx` HTTP error code is returned.

As with the synchronous client API, you can retrieve the response entity as a Java type directly without requesting a `Response` first. In case of an `InvocationCallback`, you need to set its generic type to the expected response entity type instead of using the `Response` type as demonstrated in the example bellow:

Example 10.8. Client async callback for specific entity

```
final Future<String> entityFuture = target().path("http://example.com/resource
        .request().async().get(new InvocationCallback<String>() {
            @Override
            public void completed(String response) {
                System.out.println("Response entity '" + response + "' receive
            }

            @Override
            public void failed(Throwable throwable) {
                System.out.println("Invocation failed.");
                throwable.printStackTrace();
            }
        });
System.out.println(entityFuture.get());
```

Here, the generic type of the invocation callback information is used to unmarshall the HTTP response content into a desired Java type.

Important

Please note that in this case the method `failed(Throwable throwable)` would be invoked even for cases when a server responds with a non HTTP-`2xx` HTTP error code. This is because in this case the user does not have any other means of finding out that the server returned an error response.

10.2.2. Chunked input

In an earlier section the `ChunkedOutput` was described. It was shown how to use a chunked output on the server. In order to read chunks on the client the ChunkedInput [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ChunkedInput.html] can be used to complete the story.

You can, of course, process input on the client as a standard input stream but if you would like to leverage Jersey infrastructure to provide support of translating message chunk data into Java types using a `ChunkedInput` is much more straightforward. See the usage of the `ChunkedInput` in the following example:

Example 10.9. ChunkedInput example

```
1 final Response response = target().path("http://example.com/resource/")
2         .request().get();
3 final ChunkedInput<String> chunkedInput =
4         response.readEntity(new GenericType<ChunkedInput<String>>() {});
5 String chunk;
6 while ((chunk = chunkedInput.read()) != null) {
7     System.out.println("Next chunk received: " + chunk);
8 }
9
```

The response is retrieved in a standard way from the server. The entity is read as a `ChunkedInput` entity. In order to do that the GenericEntity<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/GenericEntity.html] is used to preserve a generic information at run time. If you would not use `GenericEntity<T>`, Java language generic type erasure would cause that the generic information would get lost at compile time and an exception would be thrown at run time complaining about the missing chunk type definition.

In the next lines in the example, individual chunks are being read from the response. Chunks can come with some delay, so they will be written to the console as they come from the server. After receiving last chunk the `null` will be returned from the `read()` method. This will mean that the server has sent the last chunk and closed the connection. Note that the `read()` is a blocking operation and the invoking thread is blocked until a new chunk comes.

Writing chunks with `ChunkedOutput` is simple, you only call method `write()` which writes exactly one chunk to the output. With the input reading it is slightly more complicated. The `ChunkedInput` does not know how to distinguish chunks in the byte stream unless being told by the developer. In order to define custom chunks boundaries, the `ChunkedInput` offers possibility to register a ChunkParser [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ChunkParser.html] which reads chunks from the input stream and separates them. Jersey provides several chunk parser implementation and you can implement your own parser to separate your chunks if you need. In our example above the default parser provided by Jersey is used that separates chunks based on presence of a `\r\n` delimiting character sequence.

Each incoming input stream is firstly parsed by the `ChunkParser`, then each chunk is processed by the proper MessageBodyReader<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/

MessageBodyReader.html]. You can define the media type of chunks to aid the selection of a proper `MessageBodyReader<T>` in order to read chunks correctly into the requested entity types (in our case into Strings).

Chapter 11. URIs and Links

11.1. Building URIs

A very important aspect of REST is hyperlinks, URIs, in representations that clients can use to transition the Web service to new application states (this is otherwise known as "hypermedia as the engine of application state"). HTML forms present a good example of this in practice.

Building URIs and building them safely is not easy with URI [http://docs.oracle.com/javase/6/docs/api/java/net/URI.html], which is why JAX-RS has the UriBuilder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriBuilder.html] class that makes it simple and easy to build URIs safely. `UriBuilder` can be used to build new URIs or build from existing URIs. For resource classes it is more than likely that URIs will be built from the base URI the web service is deployed at or from the request URI. The class UriInfo [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriInfo.html] provides such information (in addition to further information, see next section).

The following example shows URI building with `UriInfo` and `UriBuilder` from the bookmark example:

Example 11.1. URI building

```
 1 @Path("/users/")
 2 public class UsersResource {
 3
 4     @Context
 5     UriInfo uriInfo;
 6
 7     ...
 8
 9     @GET
10     @Produces("application/json")
11     public JSONArray getUsersAsJsonArray() {
12         JSONArray uriArray = new JSONArray();
13         for (UserEntity userEntity : getUsers()) {
14             UriBuilder ub = uriInfo.getAbsolutePathBuilder();
15             URI userUri = ub.
16                     path(userEntity.getUserid()).
17                     build();
18             uriArray.put(userUri.toASCIIString());
19         }
20         return uriArray;
21     }
22 }
```

`UriInfo` is obtained using the @Context annotation, and in this particular example injection onto the field of the root resource class is performed, previous examples showed the use of @Context on resource method parameters.

`UriInfo` can be used to obtain URIs and associated `UriBuilder` instances for the following URIs: the base URI the application is deployed at; the request URI; and the absolute path URI, which is the request URI minus any query components.

The `getUsersAsJsonArray` method constructs a `JSONArrray`, where each element is a URI identifying a specific user resource. The URI is built from the absolute path of the request URI by

calling UriInfo.getAbsolutePathBuilder() [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriInfo.html#getAbsolutePathBuilder()]. A new path segment is added, which is the user ID, and then the URI is built. Notice that it is not necessary to worry about the inclusion of `'/'` characters or that the user ID may contain characters that need to be percent encoded. `UriBuilder` takes care of such details.

`UriBuilder` can be used to build/replace query or matrix parameters. URI templates can also be declared, for example the following will build the URI `"http://localhost/segment?name=value"`:

Example 11.2. Building URIs using query parameters

```
1 UriBuilder.fromUri("http://localhost/")
2     .path("{a}")
3     .queryParam("name", "{value}")
4     .build("segment", "value");
```

11.2. Resolve and Relativize

JAX-RS 2.0 introduced additional URI resolution and relativization methods in the `UriBuilder`:

- UriInfo.resolve(java.net.URI) [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriInfo.html#resolve(java.net.URI)]
- UriInfo.relativize(java.net.URI) [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriInfo.html#relativize(java.net.URI)]
- UriBuilder.resolveTemplate(...) [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriBuilder.html#resolveTemplate(java.lang.String, java.lang.Object)] (various arguments)

Resolve and relativize methods in UriInfo [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriInfo.html] are essentially counterparts to the methods listed above - UriInfo.resolve(java.net.URI) [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriInfo.html#resolve(java.net.URI)] resolves given relative URI to an absolute URI using application context URI as the base URI; UriInfo.relativize(java.net.URI) [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriInfo.html#relativize(java.net.URI)] then transforms an absolute URI to a relative one, using again the applications context URI as the base URI.

`UriBuilder` also introduces a set of methods that provide ways of resolving URI templates by replacing individual templates with a provided value(s). A short example:

```
1 final URI uri = UriBuilder.fromUri("http://{host}/{path}?q={param}")
2     .resolveTemplate("host", "localhost")
3     .resolveTemplate("path", "myApp")
4     .resolveTemplate("param", "value").build();
5
6 uri.toString(); // returns "http://localhost/myApp?q=value"
```

See the UriBuilder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriBuilder.html] javadoc for more details.

11.3. Link

JAX-RS 2.0 introduces Link [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Link.html] class, which serves as a representation of Web Link defined in RFC 5988 [http://tools.ietf.org/html/

rfc5988]. The JAX-RS `Link` class adds API support for providing additional metadata in HTTP messages, for example, if you are consuming a REST interface of a public library, you might have a resource returning description of a single book. Then you can include links to related resources, such as a book category, author, etc. to make the produced response concise but complete at the same time. Clients are then able to query all the additional information they are interested in and are not forced to consume details they are not interested in. At the same time, this approach relieves the server resources as only the information that is truly requested is being served to the clients.

A `Link` can be serialized to an HTTP message (tyically a response) as additional HTTP header (there might be multiple `Link` headers provided, thus multiple links can be served in a single message). Such HTTP header may look like:

```
Link: <http://example.com/TheBook/chapter2>; rel="prev"; title="previous chapter"
```

Producing and consuming Links with JAX-RS API is demonstrated in the following example:

```
// server side - adding links to a response:
Response r = Response.ok().
    link("http://oracle.com", "parent").
    link(new URI("http://jersey.java.net"), "framework").
    build();

...

// client-side processing:
final Response response = target.request().get();

URI u = response.getLink("parent").getUri();
URI u = response.getLink("framework").getUri();
```

Instances of `Link` can be also created directly by invoking one of the factory methods on the Link [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Link.html] API that returns a Link.Builder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Link.Builder.html] that can be used to configure and produce new links.

Chapter 12. Programmatic API for Building Resources

12.1. Introduction

A standard approach of developing JAX-RS application is to implement resource classes annotated with @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] with resource methods annotated with HTTP method annotations like @GET [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/GET.html] or @POST [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/POST.html] and implement needed functionality. This approach is described in the chapter JAX-RS Application, Resources and Sub-Resources [jaxrs-resources.html]. Besides this standard JAX-RS approach, Jersey offers an API for constructing resources programmatically.

Imagine a situation where a deployed JAX-RS application consists of many resource classes. These resources implement standard HTTP methods like @GET [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/GET.html], @POST [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/POST.html], @DELETE [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/DELETE.html]. In some situations it would be useful to add an @OPTIONS [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/OPTIONS.html] method which would return some kind of meta data about the deployed resource. Ideally, you would want to expose the functionality as an additional feature and you want to decide at the deploy time whether you want to add your custom `OPTIONS` method. However, when custom `OPTIONS` method are not enabled you would like to be `OPTIONS` requests handled in the standard way by JAX-RS runtime. To achieve this you would need to modify the code to add or remove custom `OPTIONS` methods before deployment. Another way would be to use programmatic API to build resource according to your needs.

Another use case of programmatic resource builder API is when you build any generic RESTful interface which depends on lot of configuration parameters or for example database structure. Your resource classes would need to have different methods, different structure for every new application deploy. You could use more solutions including approaches where your resource classes would be built using Java byte code manipulation. However, this is exactly the case when you can solve the problem cleanly with the programmatic resource builder API. Let's have a closer look at how the API can be utilized.

12.2. Programmatic Hello World example

Jersey Programmatic API was designed to fully support JAX-RS resource model. In other words, every resource that can be designed using standard JAX-RS approach via annotated resource classes can be also modelled using Jersey programmatic API. Let's try to build simple hello world resource using standard approach first and then using the Jersey programmatic resource builder API.

The following example shows standard JAX-RS "Hello world!" resource class.

Example 12.1. A standard resource class

```
 1
 2                    @Path("helloworld")
 3                    public class HelloWorldResource {
 4
 5                        @GET
 6                        @Produces("text/plain")
 7                        public String getHello() {
 8                            return "Hello World!";
 9                        }
10                    }
11
```

This is just a simple resource class with one GET method returning "Hello World!" string that will be serialized as text/plain media type.

Now we will design this simple resource using programmatic API.

Example 12.2. A programmatic resource

```
 1
 2                    package org.glassfish.jersey.examples.helloworld;
 3
 4                    import javax.ws.rs.container.ContainerRequestContext;
 5                    import javax.ws.rs.core.Application;
 6                    import javax.ws.rs.core.Response;
 7                    import org.glassfish.jersey.process.Inflector;
 8                    import org.glassfish.jersey.server.ResourceConfig;
 9                    import org.glassfish.jersey.server.model.Resource;
10                    import org.glassfish.jersey.server.model.ResourceMethod;
11
12
13                    public static class MyResourceConfig extends ResourceConfi
14
15                        public MyResourceConfig() {
16                            final Resource.Builder resourceBuilder = Resource.
17                            resourceBuilder.path("helloworld");
18
19                            final ResourceMethod.Builder methodBuilder = resou
20                            methodBuilder.produces(MediaType.TEXT_PLAIN_TYPE)
21                                    .handledBy(new Inflector<ContainerRequestC
22
23                                @Override
24                                public String apply(ContainerRequestContext co
25                                    return "Hello World!";
26                                }
27                            });
28
29                            final Resource resource = resourceBuilder.build();
30                            registerResources(resource);
31                        }
32                    }
33
```

First, focus on the content of the `MyResourceConfig` constructor in the example. The Jersey programmatic resource model is constructed from `Resource`s that contain `ResourceMethod`s. In the example, a single resource would be constructed from a `Resource` containing one `GET` resource method that returns "Hello World!". The main entry point for building programmatic resources in Jersey is the *`Resource.Builder`* class. `Resource.Builder` contains just a few methods like the `path` method used in the example, which sets the name of the path. Another useful method is a `addMethod(String path)` which adds a new method to the resource builder and returns a resource method builder for the new method. Resource method builder contains methods which set consumed and produced media type, define name bindings, timeout for asynchronous executions, etc. It is always necessary to define a resource method handler (i.e. the code of the resource method that will return "Hello World!"). There are more options how a resource method can be handled. In the example the implementation of `Inflector` is used. The Jersey `Inflector` interface defines a simple contract for transformation of a request into a response. An inflector can either return a Response [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Response.html] or directly an entity object, the way it is shown in the example. Another option is to setup a Java method handler using `handledBy(Class<?> handlerClass, Method method)` and pass it the chosen `java.lang.reflect.Method` instance together with the enclosing class.

A resource method model construction can be explicitly completed by invoking `build()` on the resource method builder. It is however not necessary to do so as the new resource method model will be built automatically once the parent resource is built. Once a resource model is built, it is registered into the resource config at the last line of the `MyResourceConfig` constructor in the example.

12.2.1. Deployment of programmatic resources

Let's now look at how a programmatic resources are deployed. The easiest way to setup your application as well as register any programmatic resources in Jersey is to use a Jersey `ResourceConfig` utility class, an extension of Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] class. If you deploy your Jersey application into a Servlet container you will need to provide a Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] subclass that will be used for configuration. You may use a `web.xml` where you would define a `org.glassfish.jersey.servlet.ServletContainer` Servlet entry there and specify initial parameter `javax.ws.rs.Application` with the class name of your JAX-RS Application that you wish to deploy. In the example above, this application will be `MyResourceConfig` class. This is the reason why `MyResourceConfig` extends the `ResourceConfig` (which, if you remember extends the `javax.ws.rs.Application`).

The following example shows a fragment of `web.xml` that can be used to deploy the `ResourceConfig` JAX-RS application.

Example 12.3. A programmatic resource

```
 1
 2                        ...
 3                        <servlet>
 4                            <servlet-name>org.glassfish.jersey.examples.hellow
 5                            <servlet-class>org.glassfish.jersey.servlet.Servle
 6                            <init-param>
 7                                <param-name>javax.ws.rs.Application</param-nam
 8                                <param-value>org.glassfish.jersey.examples.hel
 9                            </init-param>
10                            <load-on-startup>1</load-on-startup>
11                        </servlet>
12                        <servlet-mapping>
13                            <servlet-name>org.glassfish.jersey.examples.hellow
14                            <url-pattern>/*</url-pattern>
15                        </servlet-mapping>
16                        ...
17
```

If you use another deployment options and you have accessible instance of ResourceConfig which you use for configuration, you can register programmatic resources directly by `registerResources()` method called on the ResourceConfig. Please note that the method registerResources() replaces all the previously registered resources.

Because Jersey programmatic API is not a standard JAX-RS feature the `ResourceConfig` must be used to register programmatic resources as shown above. See deployment chapter for more information.

12.3. Additional examples

Example 12.4. A programmatic resource

```
 1
 2                    final Resource.Builder resourceBuilder = Resource.builder(
 3                    resourceBuilder.addMethod("OPTIONS")
 4                        .handledBy(new Inflector<ContainerRequestContext, Resp
 5                            @Override
 6                            public Response apply(ContainerRequestContext cont
 7                                return Response.ok("This is a response to an O
 8                            }
 9                        });
10                    final Resource resource = resourceBuilder.build();
11
```

In the example above the `Resource` is built from a `HelloWorldResource` resource class. The resource model built this way contains a `GET` method producing `'text/plain'` responses with "Hello World!" entity. This is quite important as it allows you to whatever Resource objects based on introspecting existing JAX-RS resources and use builder API to enhance such these standard resources. In the example we used already implemented `HelloWorldResource` resource class and enhanced it by `OPTIONS` method. The `OPTIONS` method is handled by an Inflector which returns Response [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Response.html].

The following sample shows how to define sub-resource methods (methods that contains sub-path).

Example 12.5. A programmatic resource

```
1
2                       final Resource.Builder resourceBuilder = Resource.builder(
3
4                       final Resource.Builder childResource = resourceBuilder.add
5                       childResource.addMethod("GET").handledBy(new GetInflector(
6
7                       final Resource resource = resourceBuilder.build();
8
```

Sub-resource methods are defined using so called *child resource models*. Child resource models (or child resources) are programmatic resources build in the same way as any other programmatic resource but they are registered as a child resource of a parent resource. The child resource in the example is build directly from the parent builder using method `addChildResource(String path)`. However, there is also an option to build a child resource model separately as a standard resource and then add it as a child resource to a selected parent resource. This means that child resource objects can be reused to define child resources in different parent resources (you just build a single child resource and then register it in multiple parent resources). Each child resource groups methods with the same sub-resource path. Note that a child resource cannot have any child resources as there is nothing like sub-sub-resource method concept in JAX-RS. For example if a sub resource method contains more path segments in its path (e.g. "/root/sub/resource/method" where "root" is a path of the resource and "sub/resource/method" is the sub resource method path) then parent resource will have path "root" and child resource will have path "sub/resource/method" (so, there will not be any separate nested sub-resources for "sub", "resource" and "method").

See the javadocs of the resource builder API for more information.

12.4. Model processors

Jersey gives you an option to register so called *model processor providers*. These providers are able to modify or enhance the application resource model during application deployment. This is an advanced feature and will not be needed in most use cases. However, imagine you would like to add `OPTIONS` resource method to each deployed resource as it is described at the top of this chapter. You would want to do it for every programmatic resource that is registered as well as for all standard JAX-RS resources.

To do that, you first need to register a model processor provider in your application, which implements `org.glassfish.jersey.server.model.ModelProcessor` extension contract. An example of a model processor implementation is shown here:

Example 12.6. A programmatic resource

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.core.Application;
import javax.ws.rs.core.Configuration;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.Provider;

import org.glassfish.jersey.process.Inflector;
import org.glassfish.jersey.server.model.ModelProcessor;
import org.glassfish.jersey.server.model.Resource;
import org.glassfish.jersey.server.model.ResourceMethod;
import org.glassfish.jersey.server.model.ResourceModel;

@Provider
public static class MyOptionsModelProcessor implements Mod

    @Override
    public ResourceModel processResourceModel(ResourceMode
        // we get the resource model and we want to enhanc
        ResourceModel.Builder newResourceModelBuilder = ne
        for (final Resource resource : resourceModel.getRe
            // for each resource in the resource model we
            final Resource.Builder resourceBuilder = Resou

            // we add a new OPTIONS method to each resourc
            // note that we should check whether the metho
            resourceBuilder.addMethod("OPTIONS")
                .handledBy(new Inflector<ContainerRequestC
                    @Override
                    public String apply(ContainerRequestCo
                        return "On this path the resource
                            + " methods is deployed.";
                    }
                    }).produces(MediaType.TEXT_PLAIN);

            // we add to the model new resource which is a
            // by the OPTIONS method
            newResourceModelBuilder.addResource(resourceBu
        }

        final ResourceModel newResourceModel = newResource
        // and we return new model
        return newResourceModel;
    };

    @Override
    public ResourceModel processSubResource(ResourceModel
        // we just return the original subResourceModel wh
        return subResourceModel;
    }
}

```

The `MyOptionsModelProcessor` from the example is a relatively simple model processor which iterates over all registered resources and for all of them builds a new resource that is equal to the old resource except it is enhanced with a new `OPTIONS` method.

Note that you only need to register such a ModelProcessor as your custom extension provider in the same way as you would register any standard JAX-RS extension provider. During an application deployment, Jersey will look for any registered model processor and execute them. As you can seem, model processors are very powerful as they can do whatever manipulation with the resource model they like. A model processor can even, for example, completely ignore the old resource model and return a new custom resource model with a single "Hello world!" resource, which would result in only the "Hello world!" resource being deployed in your application. Of course, it would not not make much sense to implement such model processor, but the scenario demonstrates how powerful the model processor concept is. A better, real-life use case scenario would, for example, be to always add some custom new resource to each application that might return additional metadata about the deployed application. Or, you might want to filter out particular resources or resource methods, which is another situation where a model processor could be used successfully.

Also note that `processSubResource(ResourceModel subResourceModel, Configuration configuration)` method is executed for each sub resource returned from the sub resource locator. The example is simplified and does not do any manipulation but probably in such a case you would want to enhance all sub resources with a new `OPTIONS` method handlers too.

It is important to remember that any model processor must always return valid resource model. As you might have already noticed, in the example above this important rule is not followed. If any of the resources in the original resource model would already have an `OPTIONS` method handler defined, adding another handler would cause the application fail during the deployment in the resource model validation phase. In order to retain the consistency of the final model, a model processor implementation would have to be more robust than what is shown in the example.

Chapter 13. Server-Sent Events (SSE) Support

13.1. What are Server-Sent Events

In a standard HTTP request-response scenario a client opens a connection, sends a HTTP request to the server (for example a HTTP `GET` request), then receives a HTTP response back and the server closes the connection once the response is fully sent/received. The initiative *always* comes from a client when the client requests all the data. In contrast, *Server-Sent Events (SSE)* is a mechanism that allows server to asynchronously push the data from the server to the client once the client-server connection is established by the client. Once the connection is established by the client, it is the server who provides the data and decides to send it to the client whenever new "chunk" of data is available. When a new data event occurs on the server, the data event is sent by the server to the client. Thus the name Server-Sent Events. Note that at high level there are more technologies working on this principle, a short overview of the technologies supporting server-to-client communication is in this list:

Polling
: With polling a client repeatedly sends new requests to a server. If the server has no new data, then it send appropriate indication and closes the connection. The client then waits a bit and sends another request after some time (after one second, for example).

Long-polling
: With long-polling a client sends a request to a server. If the server has no new data, it just holds the connection open and waits until data is available. Once the server has data (message) for the client, it uses the connection and sends it back to the client. Then the connection is closed.

Server-Sent events
: SSE is similar to the long-polling mechanism, except it does not send only one message per connection. The client sends a request and server holds a connection until a new message is ready, then it sends the message back to the client while still keeping the connection open so that it can be used for another message once it becomes available. Once a new message is ready, it is sent back to the client on the same initial connection. Client processes the messages sent back from the server individually without closing the connection after processing each message. So, SSE typically reuses one connection for more messages (called events). SSE also defines a dedicated media type that describes a simple format of individual evnets sent from the server to the client. SSE also offers standard javascript client API implemented most modern browsers. For more information about SSE, see the SSE API specification [http://www.w3.org/TR/2009/WD-eventsource-20091029/].

WebSocket
: WebSocket technology is different from previous technologies as it provides a real full duplex connection. The initiator is again a client which sends a request to a server with a special HTTP header that informs the server that the HTTP connection may be "upgraded" to a full duplex TCP/IP WebSocket connection. If server supports WebSocket, it may choose to do so. Once a WebSocket connection is established, it can be used for bi-directional communication between the client and the server. Both client and server can then send data to the other party at will whenever it is needed. The communication on the new WebSocket connection is no longer based on

HTTP protocol and can be used for example for for online gaming or any other applications that require fast exchange of small chunks of data in flowing in both directions.

13.2. When to use Server-Sent Events

As explained above, SSE is a technology that allows clients to subscribe to event notifications that originate on a server. Server generates new events and sends these events back to the clients subscribed to receive the notifications. In other words, SSE offers a solution for a one-way publish-subscribe model.

A good example of the use case where SSE can be used is a simple message exchange RESTful service. Clients `POST` new messages to the service and subscribe to receive messages from other clients. Let's call the resource `messages`. While `POST`ing a new message to this resource involves a typical HTTP request-response communication between a client and the `messages` resource, subscribing to receive all new message notifications would be hard and impractical to model with a sequence of standard request-response message exchanges. Using Server-sent events provides a much more practical approach here. You can use SSE to let clients subscribe to the `messages` resource via standard `GET` request (use a SSE client API, for example javascript API or Jersey Client SSE API) and let the server broadcast new messages to all connected clients in the form of individual events (in our case using Jersey Server SSE API). Note that with Jersey a SSE support is implemented as an usual JAX-RS resource method. There's no need to do anything special to provide a SSE support in your Jersey/JAX-RS applications, your SSE-enabled resources are a standard part of your RESTful Web application that defines the REST API of your application. The following chapters describes SSE support in Jersey in more details.

Important

Note, that while SSE in Jersey is supported with standard JAX-RS resources, Jersey SSE APIs are not part of the JAX-RS specification. SSE support and related APIs are a Jersey specific feature that extends JAX-RS.

13.3. Jersey Server-Sent Events API

This chapter briefly describes the Jersey support for SSE. Details and examples will be covered in chapters below.

Jersey contains support for SSE for both - server and client. SSE in Jersey is implemented as an extension supporting a new media type, which means that SSE really treated as just another media type that can be returned from a resource method and processed by the client. There is only a minimal additional support for "chunked" messages added to Jersey which could not be implemented as standard JAX-RS media type extension.

Before you start working with Jersey SSE, in order to add support for SSE you need to include the dependency to the *SSE media type module*:

```
1 <dependency>
2     <groupId>org.glassfish.jersey.media</groupId>
3     <artifactId>jersey-media-sse</artifactId>
4 </dependency>
```

Then you need register SseFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/SseFeature.html]. The `SseFeature` is a feature that can be registered for both, the client and the server.

`SseFeature` adds new supported entity (representation) Java types, namely OutboundEvent [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/OutboundEvent.html] for the outbound server events and InboundEvent [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/InboundEvent.html] for inbound client events. These types are serialized by OutboundEvent [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/OutboundEventWriter.html] and de-serialized by InboundEventReader [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/InboundEventReader.html]. There is no restriction for a media type used in individual event messages; however the media type used for a SSE stream as whole is *"text/event-stream"* and this media type should be set on messages that are used to serve SSE events (for example on the server side using @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] on the method that returns an `EventOutput` - see bellow). The `InboundEvent` and `OutboundEvent` contain all the fields needed for composing and processing individual SSE events. These entities represent the *chunks* sent or received over an open server-to-client connection that is represented by an ChunkedOutput [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ChunkedOuptut.html] on the servers side and ChunkedInput [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ChunkedInput.html] on the client side (if you are not familiar with `ChunkedOutput` and `ChunkedInput`, see the Async chapter first for more details). In other words, our resource method that is used to open a SSE connection to a client does not return individual `OutboundEvent`s. Instead, a new instance of EventOutput [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/EventOutput.html] is returned. `EventOutput` is a typed extension of `ChunkedOutput<OutboundEvent>`. Similarly, to receive `InboundEvent`s on a client side, Jersey SSE API provides a `EventInput` that represents a typed extension of `ChunkedInput<InboundEvent>`.

The Jersey server SSE API also contains a SseBroadcaster [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/SseBroadcaster.html] utility, that provides a convenient way of grouping multiple `EventOutput` instances that represent individual client connections into a group, and exposes methods for broadcasting new events to all the client connections grouped in the broadcaster. The `SseBroadcaster` inherits from Broadcaster [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/Broadcaster.html] which is the generic broadcaster implementation of the Jersey chunked message processing facility. On the he client side, the Jersey SSE API contains additional EventSource [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/EventSource.html] and EventListener [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/EventListener.html] classes that further improve convenience of processing new inbound SSE events.

13.4. Implementing SSE support in a JAX-RS resource

13.4.1. Simple SSE resource method

Firstly you need to add a Jersey SSE module dependency to your project as shown in the earlier section and register the SseFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/SseFeature.html] from this module in your Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] or ResourceConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ResourceConfig.html]. Once done, you are ready to add SSE support to your resource:

Example 13.1. Simple SSE resource method

```
 1 ...
 2 import org.glassfish.jersey.media.sse.EventOutput;
 3 import org.glassfish.jersey.media.sse.OutboundEvent;
 4 import org.glassfish.jersey.media.sse.SseFeature;
 5 ...
 6
 7 @Path("events")
 8 public static class SseResource {
 9
10     @GET
11     @Produces(SseFeature.SERVER_SENT_EVENTS)
12     public EventOutput getServerSentEvents() {
13         final EventOutput eventOutput = new EventOutput();
14         new Thread(new Runnable() {
15             @Override
16             public void run() {
17                 try {
18                     for (int i = 0; i < 10; i++) {
19                         // ... code that waits 1 second
20                         final OutboundEvent.Builder eventBuilder
21                         = new OutboundEvent.Builder();
22                         eventBuilder.name("message-to-client");
23                         eventBuilder.data(String.class,
24                             "Hello world " + i + "!");
25                         final OutboundEvent event = eventBuilder.build();
26                         eventOutput.write(event);
27                     }
28                 } catch (IOException e) {
29                     throw new RuntimeException(
30                         "Error when writing the event.", e);
31                 } finally {
32                     try {
33                         eventOutput.close();
34                     } catch (IOException ioClose) {
35                         throw new RuntimeException(
36                             "Error when closing the event output.", ioClose);
37                     }
38                 }
39             }
40         }).start();
41         return eventOutput;
42     }
43 }
```

The code above defines the resource deployed on URI `"/events"`. This resource has a single @GET [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/GET.html] resource method which returns as an entity EventOutput [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/EventOutput.html] - an extension of generic Jersey ChunkedOutput [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ChunkedOuptut.html] API for output chunked message processing.

Note

If you are not familiar with `ChunkedOutput` and `ChunkedInput`, see the Async chapter first for more details.

After the `eventOutput` is returned from the method, the Jersey runtime recognizes that this is a `ChunkedOutput` extension and does not close the client connection immediately. Instead, it writes the HTTP headers to the response stream and waits for more chunks (SSE events) to be sent. At this point the client can read headers and starts listening for individual events.

Note

Since Jersey runtime does not implicitly close the connection to the client (similarly to asynchronous processing), closing the connection is a responsibility of the resource method or the client listening on the open connection for new events (see following example).

In the Example 13.1, “Simple SSE resource method”, the resource method creates a new thread that sends a sequence of 10 events. There is a 1 second delay between two subsequent events as indicated in a comment. Each event is represented by `OutboundEvent` type and is built with a helpf of an outbound event `Builder`. The `OutboundEvent` reflects the standardized format of SSE messages and contains properties that represent `name` (for named events), `comment`, `data` or `id`. The code also sets the event data media type using the `mediaType(MediaType)` method on the `eventBuilder`. The media type, together with the data type set by the `data(Class, Object>` method (in our case `String.class`), is used for serialization of the event data. Note that the event data media type will not be written to any headers as the response `Content-type` header is already defined by the `@Produces` and set to `"text/event-stream"` using constant from the `SseFeature`. The event media type is used for serialization of event `data`. Event data media type and Java type are used to select the proper MessageBodyWriter<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyWriter.html] for event data serialization and are passed to the selected writer that serializes the event `data` content. In our case the string `"Hello world " + i + "!"` is serialized as `"text/plain"`. In event `data` you can send any Java entity and associate it with any media type that you would be able to serialize with an available `MessageBodyWriter<T>`. Typically, you may want to send e.g. JSON data, so you would fill the `data` with a JAXB annotated bean instance and define media type to JSON.

Note

If the event media type is not set explicitly, the `"text/plain"` media type is used by default.

Once an outbound event is ready, it can be written to the `eventOutput`. At that point the event is serialized by internal `OutboundEventWriter` which uses an appropriate `MessageBodyWriter<T>` to serialize the `"Hello world " + i + "!"` string. You can send as many messages as you like. At the end of the thread execution the response is closed which also closes the connection to the client. After that, no more messages can be send to the client on this connection. If the client would like to receive more messages, it would have to send a new request to the server to initiate a new SSE streaming connection.

A client connecting to our SSE-enabled resource will receive the following data from the entity stream:

```
event: message-to-client
data: Hello world 0!

event: message-to-client
data: Hello world 1!
```

```
event: message-to-client
data: Hello world 2!

event: message-to-client
data: Hello world 3!

event: message-to-client
data: Hello world 4!

event: message-to-client
data: Hello world 5!

event: message-to-client
data: Hello world 6!

event: message-to-client
data: Hello world 7!

event: message-to-client
data: Hello world 8!

event: message-to-client
data: Hello world 9!
```

Each message is received with a delay of one second.

13.4.2. Broadcasting with Jersey SSE

Jersey SSE server API defines SseBroadcaster [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/SseBroadcaster.html] which allows to broadcast individual events to multiple clients. A simple broadcasting implementation is shown in the following example:

Example 13.2. Broadcasting SSE messages

```
 1 ...
 2 import org.glassfish.jersey.media.sse.SseBroadcaster;
 3 ...
 4
 5 @Singleton
 6 @Path("broadcast")
 7 public static class BroadcasterResource {
 8
 9     private SseBroadcaster broadcaster = new SseBroadcaster();
10
11     @POST
12     @Produces(MediaType.TEXT_PLAIN)
13     @Consumes(MediaType.TEXT_PLAIN)
14     public String broadcastMessage(String message) {
15         OutboundEvent.Builder eventBuilder = new OutboundEvent.Builder();
16         OutboundEvent event = eventBuilder.name("message")
17             .mediaType(MediaType.TEXT_PLAIN_TYPE)
18             .data(String.class, message)
19             .build();
20
21         broadcaster.broadcast(event);
22
23         return "Message was '" + message + "' broadcast.";
24     }
25
26     @GET
27     @Produces(SseFeature.SERVER_SENT_EVENTS)
28     public EventOutput listenToBroadcast() {
29         final EventOutput eventOutput = new EventOutput();
30         this.broadcaster.add(eventOutput);
31         return eventOutput;
32     }
33 }
```

Let's explore the example together. The `BroadcasterResource` resource class is annotated with @Singleton [http://docs.oracle.com/javaee/6/api/javax/inject/Singleton.html] annotation which tells Jersey runtime that only a single instance of the resource class should be used to serve all the incoming requests to `/broadcast` path. This is needed as we want to keep an application-wide single reference to the private `broadcaster` field so that we can use the same instance for all requests. Clients that want to listen to SSE events first send a `GET` request to the `BroadcasterResource`, that is handled by the `listenToBroadcast()` resource method. The method creates a new `EventOutput` representing the connection to the requesting client and registers this `eventOutput` instance with the singleton `broadcaster`, using its `add(EventOutput)` method. The method then returns the `eventOutput` which causes Jersey to bind the `eventOutput` instance with the requesting client and send the response HTTP headers to the client. The client connection remains open and the client is now waiting ready to receive new SSE events. All the events are written to the `eventOutput` by `broadcaster` later on. This way developers can conveniently handle sending new events to all the clients that subscribe to them.

When a client wants to broadcast new message to all the clients listening on their SSE connections, it sends a `POST` request to `BroadcasterResource` resource with the message content. The method `broadcastMessage(String)` is invoked on `BroadcasterResource` resource with the message content as an input parameter. A new SSE outbound event is built in the standard

way and passed to the broadcaster. The broadcaster internally invokes `write(OutboundEvent)` on all registered `EventOutput`s. After that the method just return a standard text response to the `POST`ing client to inform the client that the message was successfully broadcast. As you can see, the `broadcastMessage(String)` resource method is just a simple JAX-RS resource method.

In order to implement such a scenario, you may have noticed, that the Jersey `SseBroadcaster` is not mandatory to complete the use case. individual EventOutput [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/EventOutput.html]s can be just stored in a collection and iterated over in the `broadcastMessage` method. However, the `SseBroadcaster` internally identifies and handles also client disconnects. When a client closes the connection the broadcaster detects this and removes the stale connection from the internal collection of the registered EventOutput [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/EventOutput.html]s as well as it frees all the server-side resources associated with the stale connection. Additionally, the `SseBroadcaster` is implemented to be thread-safe, so that clients can connect and disconnect in any time and `SseBroadcaster` will always broadcast messages to the most recent collection of registered and active set of clients.

13.5. Consuming SSE events with Jersey clients

On the client side, Jersey exposes APIs that support receiving and processing SSE events using two programming models:

Pull model - pulling events from a EventInput [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/EventInput.html], or
Push model - listening for asynchronous notifications of `EventSource`
Both models will be described.

13.5.1. Reading SSE events with `EventInput`

The events can be read on the client side from a EventInput [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/EventInput.html]. See the following code:

```
 1 Client client = ClientBuilder.newBuilder()
 2         .register(SseFeature.class).build();
 3 WebTarget target = client.target("http://localhost:9998/events");
 4
 5 EventInput eventInput = target.request().get(EventInput.class);
 6 while (!eventInput.isClosed()) {
 7     final InboundEvent inboundEvent = eventInput.read();
 8     if (inboundEvent == null) {
 9         // connection has been closed
10         break;
11     }
12     System.out.println(inboundEvent.getName() + "; "
13         + inboundEvent.getData(String.class));
14 }
```

In this example, a client connects to the server where the `SseResource` from the Example 13.1, “Simple SSE resource method” is deployed. At first, a new JAX-RS/Jersey `client` instance is created with a `SseFeature` registered. Then a WebTarget [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/WebTarget.html] instance is retrieved from the `client` and is used to invoke a HTTP request. The returned response entity is directly read as a `EventInput` Java type, which is an extension of Jersey `ChunkedInput` that provides generic support for consuming chunked message payloads.

The code in the example then process starts a loop to process the inbound SSE events read from the `eventInput` response stream. Each chunk read from the input is a `InboundEvent`. The method `InboundEvent.getData(Class)` provides a way for the client to indicate what Java type should be used for the event data de-serialization. In our example, individual events are de-serialized as `String` Java type instances. This method internally finds and executes a proper MessageBodyReader<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyReader.html] which is the used to do the actual de-serialization. This is similar to reading an entity from the Response [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Response.html] by `readEntity(Class)`. The method `getData` can also throw an IO exception.

The `null` check on `inboundEvent` is necessary to make sure that the chunk was properly read and connection has not been closed by the server. Once the connection is closed, the loop terminates and the program completes execution. The client code produces the following console output:

```
message-to-client; Hello world 0!
message-to-client; Hello world 1!
message-to-client; Hello world 2!
message-to-client; Hello world 3!
message-to-client; Hello world 4!
message-to-client; Hello world 5!
message-to-client; Hello world 6!
message-to-client; Hello world 7!
message-to-client; Hello world 8!
message-to-client; Hello world 9!
```

13.5.2. Asynchronous SSE processing with `EventSource`

The main Jersey SSE client API component used to read SSE events asynchronously is EventSource [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/EventSource.html]. The usage of the `EventSource` is shown on the following example.

Example 13.3. Registering `EventListener` with `EventSource`

```
 1 Client client = ClientBuilder.newBuilder()
 2                       .register(SseFeature.class).build();
 3 WebTarget target = client.target("http://example.com/events");
 4 EventSource eventSource = new EventSource(target, false);
 5 EventListener listener = new EventListener() {
 6         @Override
 7         public void onEvent(InboundEvent inboundEvent) {
 8             try {
 9                 System.out.println(inboundEvent.getName() + "; "
10                         + inboundEvent.getData(String.class));
11             } catch (IOException e) {
12                 throw new RuntimeException(
13                         "Error when deserializing of data.");
14             }
15         }
16     };
17 eventSource.register(listener, "message-to-client");
18 eventSource.open();
19 ...
20 eventSource.close();
```

In this example, the client code again connects to the server where the `SseResource` from the Example 13.1, “Simple SSE resource method” is deployed. The Client [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/Client.html] instance is again created and initialized with SseFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/SseFeature.html]. Then the WebTarget [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/WebTarget.html] is built. In this case a request to the web target is not made directly in the code, instead, the web target instance is used to initialize a new `EventSource` instance. The second parameter `false` of the `EventSource` constructor tells the `EventSource` to not automatically connect to the `target` as part of its initialization logic in the constructor. The connection is established later by calling `eventSource.open()`. A custom EventListener [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/EventListener.html] implementation is used to listen to and process incoming SSE events. The method getData(Class) says that the event data should be de-serialized from a received InboundEvent [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/InboundEvent.html] instance into a `String` Java type. This method call internally executes MessageBodyReader<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyReader.html] which de-serializes the event data. This is similar to reading an entity from the Response [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Response.html] by `readEntity(Class)`. The method `getData` can throw an IO exception.

The custom event source listener is registered in the event source via `EventSource.register(EventListener, String)` method. The next method arguments define the names of the events to receive and can be omitted. If names are defined, the listener will be associated with the named events and will only invoked for events with a name from the set of defined event names. It will not be invoked for events with any other name or for events without a name.

Important

It is a common mistake to think that unnamed events will be processed by listeners that are registered to process events from a particular name set. That is NOT the case! Unnamed events are only processed by listeners that are not name-bound. The same limitation applied to HTML5 Javascript SSE Client API supported by modern browsers.

After a connection to the server is opened by calling the `open()` method on the event source, the `eventSource` starts listening to events. When an event named `"message-to-client"` comes, the listener will be executed by the event source. If any other event comes (with a name different from `"message-to-client"`), the registered listener is not invoked. Once the client is done with processing and does not want to receive events anymore, it closes the connection by calling the `close()` method on the event source.

The listener from the example above will print the following output:

```
message-to-client; Hello world 0!
message-to-client; Hello world 1!
message-to-client; Hello world 2!
message-to-client; Hello world 3!
message-to-client; Hello world 4!
message-to-client; Hello world 5!
message-to-client; Hello world 6!
message-to-client; Hello world 7!
message-to-client; Hello world 8!
message-to-client; Hello world 9!
```

When browsing through the Jersey SSE API documentation, you may have noticed that the EventSource [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/EventSource.html] implements EventListener [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/media/sse/EventListener.html] and provides an empty implementation for the `onEvent(InboundEvent`

`inboundEvent)` listener method. This adds more flexibility to the Jersey client-side SSE API. Instead of defining and registering a separate event listener, in simple scenarios you can also choose to derive directly from the `EventSource` and override the empty listener method to handle the incoming events. This programming model is shown in the following example:

Example 13.4. Overriding `EventSource.onEvent(InboundEvent)` method

```
 1 Client client = ClientBuilder.newBuilder()
 2                       .register(SseFeature.class).build();
 3 WebTarget target = client.target("http://example.com/events");
 4 EventSource eventSource = new EventSource(target) {
 5     @Override
 6     public void onEvent(InboundEvent inboundEvent) {
 7         if ("message-to-client".equals(inboundEvent.getName())) {
 8             try {
 9                 System.out.println(inboundEvent.getName() + "; "
10                         + inboundEvent.getData(String.class));
11             } catch (IOException e) {
12                 throw new RuntimeException(
13                         "Error when deserializing of data.");
14             }
15         }
16     }
17 };
18 ...
19 eventSource.close();
```

The code above is very similar to the code in Example 13.3, “Registering `EventListener` with `EventSource`”. The `EventSource` is constructed without the second `boolean open` argument. This means, that the connection to the server is by default automatically opened in the event source constructor. The implementation of the `EventListener` has been moved into the overridden `EventSource.onEvent(...)` method. However, this time, the listener method will be executed for all events - unnamed as well as with any `name`. Therefore the code checks the name whether it is an event with the name "message-to-client" that we want to handle. Note that you can still register additional `EventListener`s later on. The overridden method on the event source allows you to handle messages even when no additional listeners are registered yet.

Chapter 14. Security

14.1. Securing server

14.1.1. SecurityContext

Security information is available by injecting a SecurityContext [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/SecurityContext.html] instance using @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html] annotation, that provides essentially the equivalent of the functionality available on HttpServletRequest [http://docs.oracle.com/javaee/5/api/javax/servlet/http/HttpServletRequest.html] API. The injected security context depends on the actual Jersey application deployment. For example, if a Jersey application is deployed in a Servlet container, the Jersey `SecurityContext` will return information of the security context retrieved from Servlet request. For Jersey applications deployed on a Grizzly server, the `SecurityContext` will return information retrieved from the Grizzly request.

`SecurityContext` can be used in conjunction with sub-resource locators to return different resources if the user principle is included in a certain role. For example, a sub-resource locator could return a different resource if a user is a preferred customer:

Example 14.1. Accessing `SecurityContext`

```
@Path("basket")
public ShoppingBasketResource get(@Context SecurityContext sc) {
    if (sc.isUserInRole("PreferredCustomer") {
        return new PreferredCustomerShoppingBasketResource();
    } else {
        return new ShoppingBasketResource();
    }
}
```

`SecurityContext` can be injected also to singleton resources and providers as a class field. In such case the proxy of the request-scoped `SecurityContext` will be injected.

Example 14.2. Injecting `SecurityContext` into a singleton resource

```
@Path("resource")
@Singleton
public static class MyResource {
    @Context
    // Jersey will inject proxy of Security Context
    SecurityContext securityContext;

    @GET
    public String getUserPrincipal() {
        return securityContext.getUserPrincipal().getName();
    }
}
```

14.1.1.1. Initialize SecurityContext with Servlets

As described above, the `SecurityContext` by default (if not overwritten by filters) only offers information from an underlying container. In the case you deploy a Jersey application in a Servlet

container, you need to setup the `<security-constraint>`, `<auth-constraint>` and user to roles mappings in order to pass correct information to the `SecurityContext`.

14.1.1.2. SecurityContext in ContainerRequestContext

The `SecurityContext` can be retrieved also from ContainerRequestContext [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/ContainerRequestContext.html] via `getSecurityContext()` method. You can also set the `SecurityContext` into the request using method `setSecurityContext(SecurityContext)`. If you set a new `SecurityContext` in the ContainerRequestFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/ContainerRequestFilter.html] into the `ContainerRequestContext`, then this security context will be used for injections in resource classes (wrapped into the proxy). This way you can implement a custom authentication filter that may setup your own `SecurityContext` to be used. To ensure the early execution of your custom authentication request filter, set the filter priority to `AUTHENTICATION` using constants from Priorities [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Priorities.html]. An early execution of you authentication filter will ensure that all other filters, resources, resource methods and sub-resource locators will execute with your custom `SecurityContext` instance.

14.1.2. Authorization - securing resources

14.1.2.1. Security resources with `web.xml`

In cases where a Jersey application is deployed in a Servlet container you can rely only on the standard Java EE Web application security mechanisms offered by the Servlet container and configurable via application's `web.xml` descriptor. You need to define the `<security-constraint>` elements in the `web.xml` and assign roles which are able to access these resources. You can also define HTTP methods that are allowed to be executed. See the following example.

Example 14.3. Injecting `SecurityContext` into singletons

```
 1 <security-constraint>
 2     <web-resource-collection>
 3         <url-pattern>/rest/admin/*</url-pattern>
 4     </web-resource-collection>
 5     <auth-constraint>
 6         <role-name>admin</role-name>
 7     </auth-constraint>
 8 </security-constraint>
 9 <security-constraint>
10     <web-resource-collection>
11         <url-pattern>/rest/orders/*</url-pattern>
12     </web-resource-collection>
13     <auth-constraint>
14         <role-name>customer</role-name>
15     </auth-constraint>
16 </security-constraint>
17 <login-config>
18     <auth-method>BASIC</auth-method>
19     <realm-name>my-defaul-realm</realm-name>
20 </login-config>
```

The example secures two kinds of URI namespaces using the HTTP Basic Authentication. `rest/admin/*` will be accessible only for user group "admin" and `rest/orders/*` will be accessible for "customer"

user group. This security configuration does not use JAX-RS or Jersey features at all as it is enforced by the Servlet container even before a request reaches the Jersey application. Keeping this security constrains up to date with your JAX-RS application might not be easy as whenever you change the @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] annotations on your resource classes you may need to update also the `web.xml` security configurations to reflect the changed JAX-RS resource paths. Therefore Jersey offers a more flexible solution based on placing standard Java EE security annotations directly on JAX-RS resource classes and methods.

14.1.2.2. Securing JAX-RS resources with annotations

With Jersey you can define the access to resources based on the user group using annotations. You can for example define that only a user group "admin" can execute specific resource method. To do that you firstly need to register RolesAllowedDynamicFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/RolesAllowedDynamicFeature.html] as a provider. The following example shows how to register the feature if your deployment is based on a ResourceConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ResourceConfig.html].

Example 14.4. Registering RolesAllowedDynamicFeature using ResourceConfig

```
1 final ResourceConfig resourceConfig = new ResourceConfig(MyResource.class);
2 resourceConfig.register(RolesAllowedDynamicFeature.class);
```

Then you can use annotations from package `javax.annotation.security` defined by JSR-250. See the following example.

Example 14.5. Injecting `SecurityContext` into singletons

```
 1 @Path("/")
 2 @PermitAll
 3 public class Resource {
 4     @RolesAllowed("user")
 5     @GET
 6     public String get() { return "GET"; }
 7
 8     @RolesAllowed("admin")
 9     @POST
10     public String post(String content) { return content; }
11
12     @Path("sub")
13     public SubResource getSubResource() {
14         return new SubResource();
15     }
16 }
```

The resource class `Resource` defined in the example is annotated with a @PermitAll [http://docs.oracle.com/javaee/6/api/javax/annotation/security/PermitAll.html] annotation. This means that all methods in the class which do not this annotation will be permitted for all user groups (no restrictions are defined). In our example, the annotation will only apply to the `getSubResource()` method as it is the only method that does not override the annotation by defining custom role-based security settings using the @RolesAllowed [http://docs.oracle.com/javaee/6/api/javax/annotation/security/RolesAllowed.html] annotation. `@RolesAllowed` annotation present on other methods defines a role or a set of roles that are allowed to execute a particular method.

These Java EE security annotations are processed internally in the request filter registered using the Jersey `RolesAllowedDynamicFeature`. The roles defined in the annotations are tested against current roles

set in the `SecurityContext` using the `SecurityContext.isUserInRole(String role)` method. In case the caller is not in the role specified by the annotation, the HTTP `404` error response is returned.

14.2. Client Security

For details about client security please see the Client chapter. Jersey client allows to define parameters of SSL communication using `HTTPS` protocol. You can also use jersey built-in authentication filter which performs *HTTP Basic Authentication*. See the client chapter for more details.

14.3. OAuth

OAuth 1.x support has not been migrated from Jersey 1.x to Jersey 2.x yet. The documentation will be updated once the OAuth support feature will be released in Jersey 2.

Chapter 15. WADL Support

15.1. WADL introduction

Jersey contains support for Web Application Description Language (WADL [http://wadl.java.net/]). WADL is a XML description of a deployed RESTful web application. It contains model of the deployed resources, their structure, supported media types, HTTP methods and so on. In a sense, WADL is a similar to the WSDL (Web Service Description Language) which describes SOAP web services. WADL is however specifically designed to describe RESTful Web resources.

Let's start with the simple WADL example. In the example there is a simple `CountryResource` deployed and we request a wadl of this resource. The context root path of the application is `http://localhost:9998`.

Example 15.1. A simple WADL example - JAX-RS resource definition

```
 1 @Path("country/{id}")
 2 public static class CountryResource {
 3
 4     private CountryService countryService;
 5
 6     public CountryResource() {
 7         // init countryService
 8     }
 9
10     @GET
11     @Produces(MediaType.APPLICATION_XML)
12     public Country getCountry(@PathParam("countryId") int countryId) {
13         return countryService.getCountry(countryId);
14     }
15 }
```

The WADL of a Jersey application that contains the resource above can be requested by a HTTP `GET` request to `http://localhost:9998/application.wadl`. The Jersey will return a response with a WADL content similar to the one in the following example:

Example 15.2. A simple WADL example - WADL content

```
30              <method name="OPTIONS" id="apply">
31                  <request>
32                      <representation mediaType="*/*"/>
33                  </request>
34                  <response>
35                      <representation mediaType="*/*"/>
36                  </response>
37              </method>
38          </resource>
39          <resource path="application.wadl">
40              <method name="GET" id="getWadl">
41                  <response>
42                      <representation mediaType="application/vnd.sun.wadl+xml"/>
43                      <representation mediaType="application/xml"/>
44                  </response>
45              </method>
46              <method name="OPTIONS" id="apply">
47                  <request>
48                      <representation mediaType="*/*"/>
49                  </request>
50                  <response>
51                      <representation mediaType="text/plain"/>
52                  </response>
53              </method>
54              <method name="OPTIONS" id="apply">
55                  <request>
56                      <representation mediaType="*/*"/>
57                  </request>
58                  <response>
59                      <representation mediaType="*/*"/>
60                  </response>
61              </method>
62              <resource path="{path}">
63                  <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
64                      type="xs:string" style="template" name="path"/>
65                  <method name="GET" id="geExternalGrammar">
66                      <response>
67                          <representation mediaType="application/xml"/>
68                      </response>
69                  </method>
70                  <method name="OPTIONS" id="apply">
71                      <request>
72                          <representation mediaType="*/*"/>
73                      </request>
74                      <response>
75                          <representation mediaType="text/plain"/>
76                      </response>
77                  </method>
78                  <method name="OPTIONS" id="apply">
79                      <request>
80                          <representation mediaType="*/*"/>
81                      </request>
82                      <response>
83                          <representation mediaType="*/*"/>
84                      </response>
85                  </method>
86              </resource>
87          </resource>
88      </resources>
89  </application>
```

In the example above the returned application WADL is shown in full. WADL schema is defined by the WADL specification. The root WADL document element is the `application`. It contains global information about the deployed JAX-RS application. Under this element there is a nested element `resources` which contains zero or more `resource` elements. Each `resource` element describes a single deployed resource. In our example, there are only two root resources - `"country/{id}"` and `"application.wadl"`. The `"application.wadl"` resource is the resource that was just requested in order to receive the application WADL document. Even though WADL support is an additional feature in Jersey it is still a resource deployed in the resource model and therefore it is itself present in the returned WADL document. The first resource element with the `path="country/{id}"` is the element that describes our custom deployed resource. This resource contains a `GET` method and three `OPTIONS` methods. The `GET` method is our getCountry() method defined in the sample. There is a method name in the `id` attribute and @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] is described in the `response/representation` WADL element. `OPTIONS` methods are the methods that are automatically added by Jersey to each resource. There is an `OPTIONS` method returning `"text/plain"` media type, that will return a response with a string entity containing the list of methods deployed on this resource (this means that instead of WADL you can use this `OPTIONS` method to get similar information in a textual representation). Another `OPTIONS` method returning `*/*` will return a response with no entity and `Allow` header that will contain list of methods as a String. The last `OPTIONS` method producing `"application/vnd.sun.wadl+xml"` returns a WADL description of the resource `"country/{id}"`. As you can see, all `OPTIONS` methods return information about the resource to which the HTTP `OPTIONS` request is made.

Second resource with a path "application.wadl" has, again, similar `OPTIONS` methods and one `GET` method which return this WADL. There is also a sub-resource with a path defined by path param `{path}`. This means that you can request a resource on the URI `http://localhost:9998/application.wadl/something`. This is used only to return an external grammar if there is any attached. Such a external grammar can be for example an `XSD` schema of the response entity which if the response entity is a JAXB bean. An external grammar support via Jersey *extended WADL support* is described in sections below.

Let's now send an HTTP `OPTIONS` request to `"country/{id}"` resource using the the `curl` command:

```
curl -X OPTIONS -H "Allow: application/vnd.sun.wadl+xml" \
    -v http://localhost:9998/country/15
```

We should see a WADL returned similar to this one:

Example 15.3. OPTIONS method returning WADL

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
    <doc xmlns:jersey="http://jersey.java.net/"
        jersey:generatedBy="Jersey: 2.0-SNAPSHOT ${buildNumber}"/>
    <grammars/>
    <resources base="http://localhost:9998/">
        <resource path="country/15">
            <method name="GET" id="getCountry">
                <response>
                    <representation mediaType="application/xml"/>
                </response>
            </method>
            <method name="OPTIONS" id="apply">
                <request>
                    <representation mediaType="*/*"/>
                </request>
                <response>
                    <representation mediaType="application/vnd.sun.wadl+xml"/>
                </response>
            </method>
            <method name="OPTIONS" id="apply">
                <request>
                    <representation mediaType="*/*"/>
                </request>
                <response>
                    <representation mediaType="text/plain"/>
                </response>
            </method>
            <method name="OPTIONS" id="apply">
                <request>
                    <representation mediaType="*/*"/>
                </request>
                <response>
                    <representation mediaType="*/*"/>
                </response>
            </method>
        </resource>
    </resources>
</application>
```

The returned WADL document has the standard WADL structure that we saw in the WADL document returned for the whole Jersey application earlier. The main difference here is that the only `resource` is the resource to which the `OPTIONS` HTTP request was sent. The resource has now path `"country/15"` and not `"country/{id}"` as the path parameter `{id}` was already specified in the request to this concrete resource.

Another, a more complex WADL example is shown in the next example.

Example 15.4. More complex WADL example - JAX-RS resource definition

```
@Path("customer/{id}")
public static class CustomerResource {
    private CustomerService customerService;

    @GET
    public Customer get(@PathParam("id") int id) {
        return customerService.getCustomerById(id);
    }

    @PUT
    public Customer put(Customer customer) {
        return customerService.updateCustomer(customer);
    }

    @Path("address")
    public CustomerAddressSubResource getCustomerAddress(@PathParam("id") int
        return new CustomerAddressSubResource(id);
    }

    @Path("additional-info")
    public Object getAdditionalInfoSubResource(@PathParam("id") int id) {
        return new CustomerAddressSubResource(id);
    }

}


public static class CustomerAddressSubResource {
    private final int customerId;
    private CustomerService customerService;

    public CustomerAddressSubResource(int customerId) {
        this.customerId = customerId;
        this.customerService = null; // init customer service here
    }

    @GET
    public String getAddress() {
        return customerService.getAddressForCustomer(customerId);
    }

    @PUT
    public void updateAddress(String address) {
        customerService.updateAddressForCustomer(customerId, address);
    }

    @GET
    @Path("sub")
    public String getDeliveryAddress() {
        return customerService.getDeliveryAddressForCustomer(customerId);
    }
}
```

The `GET` request to `http://localhost:9998/application.wadl` will return the following WADL document:

Example 15.5. More complex WADL example - WADL content

```
                </method>
                <method name="PUT" id="updateAddress"/>
                <resource path="sub">
                    <method name="GET" id="getDeliveryAddress">
                        <response/>
                    </method>
                </resource>
            </resource>
            <resource path="application.wadl">
                <method name="GET" id="getWadl">
                    <response>
                        <representation mediaType="application/vnd.sun.wadl+xml"/>
                        <representation mediaType="application/xml"/>
                    </response>
                </method>
                <method name="OPTIONS" id="apply">
                    <request>
                        <representation mediaType="*/*"/>
                    </request>
                    <response>
                        <representation mediaType="text/plain"/>
                    </response>
                </method>
                <method name="OPTIONS" id="apply">
                    <request>
                        <representation mediaType="*/*"/>
                    </request>
                    <response>
                        <representation mediaType="*/*"/>
                    </response>
                </method>
                <resource path="{path}">
                    <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
                        type="xs:string" style="template" name="path"/>
                    <method name="GET" id="geExternalGrammar">
                        <response>
                            <representation mediaType="application/xml"/>
                        </response>
                    </method>
                    <method name="OPTIONS" id="apply">
                        <request>
                            <representation mediaType="*/*"/>
                        </request>
                        <response>
                            <representation mediaType="text/plain"/>
                        </response>
                    </method>
                    <method name="OPTIONS" id="apply">
                        <request>
                            <representation mediaType="*/*"/>
                        </request>
                        <response>
                            <representation mediaType="*/*"/>
                        </response>
                    </method>
                </resource>
            </resource>
        </resources>
    </application>
```

The `resource` with `path="customer/{id}"` is similar to the country resource from the previous example. There is a path parameter which identifies the customer by `id`. The resource contains 2 user-declared methods and again auto-generated `OPTIONS` methods added by Jersey. THe resource declares 2 sub-resource locators which are represented in the returned WADL document as nested `resource` elements. Note that the sub-resource locator `getCustomerAddress()` returns a type CustomerAddressSubResource in the method declaration and also in the WADL there is a `resource` element for such a sub resource with full internal description. The second method `getAdditionalInfoSubResource()` returns only an `Object` in the method declaration. While this is correct from the JAX-RS perspective as the real returned type can be computed from a request information, it creates a problem for WADL generator because WADL is generated based on the static configuration of the JAX-RS application resources. The WADL generator does not know what type would be actually returned to a request at run time. That is the reason why the nested `resource` element with `path="additional-info"` does not contain any information about the supported resource representations.

The `CustomerAddressSubResource` sub-resource described in the nested element `<resource path="address">` does not contain an `OPTIONS` method. While these methods are in fact generated by Jersey for the sub-resource, Jersey WADL generator does not currently support adding these methods to the sub-resource description. This should be addressed in the near future. Still, there are two user-defined resource methods handling HTTP `GET` and `PUT` requests. The sub-resource method `getDeliveryAddress()` is represented as a separate nested resource with `path="sub"`. Should there be more sub-resource methods defined with `path="sub"`, then all these method descriptions would be placed into the same `resource` element. In other words, sub-resource methods are grouped in WADL as sub-resources based on their `path` value.

15.2. Configuration

WADL generation is enabled in Jersey by default. This means that `OPTIONS` methods are added by default to each resource and an auto-generated `/application.wadl` resource is deployed too. To override this default behavior and disable WADL generation in Jersey, setup the configuration property in your application:

```
jersey.config.server.wadl.disableWadl=true
```

This property can be setup in a `web.xml` if the Jersey application is deployed in the servlet with `web.xml` or the property can be returned from the Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html]. `getProperties()`. See Deployment chapter for more information on setting the application configuration properties in various deployments.

WADL support in Jersey is implemented via ModelProcessor [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/model/ModelProcessor.html] extension. This implementation enhances the application resource model by adding the WADL providing resources. WADL `ModelProcessor` priority value is high (i.e. the priority is low) as it should be executed as one of the last model processors. Therefore, any `ModelProcessor` executed before will not see WADL extensions in the resource model. WADL handling resource model extensions (resources and `OPTIONS` resource methods) are not added to the application resource model if there is already a matching resource or a resource method detected in the model. In other words, if you define for example your own `OPTIONS` method that would produce `"application.wadl"` response content, this method will not be overridden by WADL model processor. See Resource builder chapter for more information on `ModelProcessor` extension mechanism.

15.3. Extended WADL support

Please note that the API of extended WADL support is going to be changed in one of the future releases of Jersey 2.x (see below).

Jersey supports extension of WADL generation called *extended WADL*. Using the extended WADL support you can enhance the generated WADL document with additional information, such as resource method javadoc-based documentation of your REST APIs, adding general documentation, adding external grammar support, or adding any custom WADL extension information.

The documentation of the existing extended WADL can be found here: Extended WADL in Jersey 1 [https://wikis.oracle.com/display/Jersey/WADL]. This contains description of an extended WADL generation in Jersey 1.x that is currently supported also by Jersey 2.x.

Again, note that the extended WADL in Jersey 2.x is NOT the intended final version and API is going to be changed. The existing set of features and functionality will be preserved but the APIs will be significantly re-designed to support additional use cases. This impacts mainly the APIs of WadlGenerator [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/wadl/WadlGenerator.html], WadlGeneratorConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/wadl/WadlGeneratorConfig.html] as well as any related classes. The API changes may impact your code if you are using a custom `WadlGenerator` or plan to implement one.

Chapter 16. Bean Validation Support

Validation is a process of verifying that some data obeys one or more pre-defined constraints. This chapter describes support for Bean Validation [http://beanvalidation.org/] in Jersey in terms of the needed dependencies, configuration, registration and usage. For more detailed description on how JAX-RS provides native support for validating resource classes based on the Bean Validation refer to the chapter in the JAX-RS spec [http://jcp.org/en/jsr/detail?id=339].

16.1. Bean Validation Dependencies

Bean Validation support in Jersey is provided as an extension module and needs to be mentioned explicitly in your `pom.xml` file (in case of using Maven):

```
<dependency>
    <groupId>org.glassfish.jersey.ext</groupId>
    <artifactId>jersey-bean-validation</artifactId>
    <version>2.1</version>
</dependency>
```

Note

If you're not using Maven make sure to have also all the transitive dependencies (see jersey-bean-validation [https://jersey.java.net/project-info/2.1/jersey/project/jersey-bean-validation/dependencies.html]) on the classpath.

This module depends directly on Hibernate Validator [http://www.hibernate.org/subprojects/validator.html] which provides a most commonly used implementation of the Bean Validation API spec.

If you want to use a different implementation of the Bean Validation API, use standard Maven mechanisms to exclude Hibernate Validator from the modules dependencies and add a dependency of your own.

```
<dependency>
    <groupId>org.glassfish.jersey.ext</groupId>
    <artifactId>jersey-bean-validation</artifactId>
    <version>2.1</version>
    <exclusions>
        <exclusion>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-validator</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

16.2. Enabling Bean Validation in Jersey

As stated in Section 4.1, “Auto-Discoverable Features”, Jersey Bean Validation is one of the modules where you don't need to explicitly register it's `Feature`s (ValidationFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/validation/ValidationFeature.html]) on the server as it's features are automatically discovered and registered when you add the `jersey-bean-validation` module to your classpath. There are three Jersey specific properties that could disable automatic discovery and registration of Jersey Bean Validation integration module:

- CommonProperties.FEATURE_AUTO_DISCOVERY_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/CommonProperties.html#FEATURE_AUTO_DISCOVERY_DISABLE]
- ServerProperties.FEATURE_AUTO_DISCOVERY_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#FEATURE_AUTO_DISCOVERY_DISABLE]
- ServerProperties.BV_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#BV_FEATURE_DISABLE]

Note

Jersey does not support Bean Validation on the client at the moment.

16.3. Configuring Bean Validation Support

Configuration of Bean Validation support in Jersey is twofold - there are few specific properties that affects Jersey behaviour (e.g. sending validation error entities to the client) and then there is ValidationConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/validation/ValidationConfig.html] class that configures Validator [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/Validator.html] used for validating resources in JAX-RS application.

To configure Jersey specific behaviour you can use the following properties:

ServerProperties.BV_DISABLE_VALIDATE_ON_EXECUTABLE_OVERRIDE_CHECK [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#BV_DISABLE_VALIDATE_ON_EXECUTABLE_OVERRIDE_CHECK] Disables @ValidateOnExecution check. More on this is described in Section 16.5, “@ValidateOnExecution”.

ServerProperties.BV_SEND_ERROR_IN_RESPONSE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#BV_SEND_ERROR_IN_RESPONSE] Enables sending validation errors in response entity to the client. More on this in Section 16.7.1, “ValidationError”.

Example 16.1. Configuring Jersey specific properties for Bean Validation.

```
1 new ResourceConfig()
2     // Now you can expect validation errors to be sent to the client.
3     .property(ServerProperties.BV_SEND_ERROR_IN_RESPONSE, true)
4     // @ValidateOnExecution annotations on subclasses won't cause errors.
5     .property(ServerProperties.BV_DISABLE_VALIDATE_ON_EXECUTABLE_OVERRIDE_CHEC
6     // Further configuration of ResourceConfig.
7     .register( ... );
```

Customization of the `Validator` used in validation of resource classes/methods can be done using `ValidationConfig` class and exposing it via ContextResolver<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/ContextResolver.html] mechanism as shown in Example 16.2, “Using `ValidationConfig` to configure `Validator`.”. You can set custom instances for the following interfaces from the Bean Validation API:

- MessageInterpolator [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/MessageInterpolator.html] - interpolates a given constraint violation message.
- TraversableResolver [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/TraversableResolver.html] - determines if a property can be accessed by the Bean Validation provider.

- ConstraintValidatorFactory [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/ConstraintValidatorFactory.html] - instantiates a `ConstraintValidator` instance based off its class. Note that by setting a custom `ConstraintValidatorFactory` you may loose injection of available resources/providers at the moment. See Section 16.6, “Injecting” how to handle this.

- ParameterNameProvider [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/ParameterNameProvider.html] - provides names for method and constructor parameters.

Tip

In the latest versions of Jersey, the old-style setter methods (`set*`) on ValidationConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/validation/ValidationConfig.html] are deprecated and replaced with methods that allow the fluent use of the API (e.g. `ValidationConfig#messageInterpolator(MessageInterpolator)`). Use of the new fluent methods is encouraged as the old setters will be removed from the API soon.

Example 16.2. Using `ValidationConfig` to configure `Validator`.

```
/**
 * Custom configuration of validation. This configuration defines custom:
 * <ul>
 *     <li>ConstraintValidationFactory - so that validators are able to inject Jer
 *     <li>ParameterNameProvider - if method input parameters are invalid, this cl
 *     instead of the default ones ({@code arg0, arg1, ..})</li>
 * </ul>
 */
public class ValidationConfigurationContextResolver implements ContextResolver<Val

    @Context
    private ResourceContext resourceContext;

    @Override
    public ValidationConfig getContext(final Class<?> type) {
        final ValidationConfig config = new ValidationConfig();
        config.setConstraintValidatorFactory(resourceContext.getResource(Injecting
        config.setParameterNameProvider(new CustomParameterNameProvider());
        return config;
    }

    /**
     * See ContactCardTest#testAddInvalidContact.
     */
    private class CustomParameterNameProvider implements ParameterNameProvider {

        private final ParameterNameProvider nameProvider;

        public CustomParameterNameProvider() {
            nameProvider = Validation.byDefaultProvider().configure().getDefaultPa
        }

        @Override
        public List<String> getParameterNames(final Constructor<?> constructor) {
            return nameProvider.getParameterNames(constructor);
        }

        @Override
        public List<String> getParameterNames(final Method method) {
            // See ContactCardTest#testAddInvalidContact.
            if ("addContact".equals(method.getName())) {
                return Arrays.asList("contact");
            }
            return nameProvider.getParameterNames(method);
        }
    }
}
```

Register this class in your app:

```
final Application application = new ResourceConfig()
        // Validation.
        .register(ValidationConfigurationContextResolver.class)
        // Further configuration.
        .register( ... );
```

Note

This code snippet has been taken from Bean Validation example [https://github.com/jersey/jersey/tree/2.1/examples/bean-validation-webapp].

16.4. Validating JAX-RS resources and methods

JAX-RS specification states that constraint annotations are allowed in the same locations as the following annotations: `@MatrixParam`, `@QueryParam`, `@PathParam`, `@CookieParam`, `@HeaderParam` and `@Context`, *except* in class constructors and property setters. Specifically, they are allowed in resource method parameters, fields and property getters as well as resource classes, entity parameters and resource methods (return values). Jersey provides support for validation (see following sections) annotated input parameters and return value of the invoked resource method as well as validation of resource class (class constraints, field constraints) where this resource method is placed. Jersey does not support, and doesn't validate, constraints placed on constructors and Bean Validation groups (only `Default` group is supported at the moment).

16.4.1. Constraint Annotations

The JAX-RS Server API provides support for extracting request values and mapping them into Java fields, properties and parameters using annotations such as @HeaderParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/HeaderParam.html], @QueryParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/QueryParam.html], etc. It also supports mapping of the request entity bodies into Java objects via non-annotated parameters (i.e., parameters without any JAX-RS annotations).

The Bean Validation specification supports the use of *constraint annotations* as a way of declaratively validating beans, method parameters and method returned values. For example, consider resource class from Example 16.3, “Constraint annotations on input parameters” augmented with constraint annotations.

Example 16.3. Constraint annotations on input parameters

```
@Path("/")
class MyResourceClass {

    @POST
    @Consumes("application/x-www-form-urlencoded")
    public void registerUser(
            @NotNull @FormParam("firstName") String firstName,
            @NotNull @FormParam("lastName") String lastName,
            @Email @FormParam("email") String email) {
        ...
    }
}
```

The annotations `@NotNull` and `@Email` impose additional constraints on the form parameters `firstName`, `lastName` and `email`. The `@NotNull` constraint is built-in to the Bean Validation API; the `@Email` constraint is assumed to be user defined in the example above. These constraint annotations are not restricted to method parameters, they can be used in any location in which JAX-RS binding annotations are allowed with the exception of constructors and property setters.

Rather than using method parameters, the `MyResourceClass` shown above could have been written as in Example 16.4, “Constraint annotations on fields”.

Example 16.4. Constraint annotations on fields

```
@Path("/")
class MyResourceClass {

    @NotNull
    @FormParam("firstName")
    private String firstName;

    @NotNull
    @FormParam("lastName")
    private String lastName;

    private String email;

    @FormParam("email")
    public void setEmail(String email) {
        this.email = email;
    }

    @Email
    public String getEmail() {
        return email;
    }

    ...
}
```

Note that in this version, `firstName` and `lastName` are fields initialized via injection and `email` is a resource class property. Constraint annotations on properties are specified in their corresponding getters.

Constraint annotations are also allowed on resource classes. In addition to annotating fields and properties, an annotation can be defined for the entire class. Let us assume that `@NonEmptyNames` validates that one of the two *name* fields in `MyResourceClass` is provided. Using such an annotation, the example above can be extended to look like Example 16.5, “Constraint annotations on class”

Example 16.5. Constraint annotations on class

```
@Path("/")
@NonEmptyNames
class MyResourceClass {

    @NotNull
    @FormParam("firstName")
    private String firstName;

    @NotNull
    @FormParam("lastName")
    private String lastName;

    private String email;

    ...
}
```

Constraint annotations on resource classes are useful for defining cross-field and cross-property constraints.

16.4.2. Annotation constraints and Validators

Annotation constraints and validators are defined in accordance with the Bean Validation specification. The `@Email` annotation used in Example 16.4, “Constraint annotations on fields” is defined using the Bean Validation @Constraint [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/Constraint.html] meta-annotation, see Example 16.6, “Definition of a constraint annotation”.

Example 16.6. Definition of a constraint annotation

```
@Target({ METHOD, FIELD, PARAMETER })
@Retention(RUNTIME)
@Constraint(validatedBy = EmailValidator.class)
public @interface Email {

    String message() default "{com.example.validation.constraints.email}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

The `@Constraint` annotation must include a reference to the validator class that will be used to validate decorated values. The `EmailValidator` class must implement `ConstraintValidator<Email, T>` where `T` is the type of values being validated, as described in Example 16.7, “Validator implementation.”.

Example 16.7. Validator implementation.

```
public class EmailValidator implements ConstraintValidator<Email, String> {

    public void initialize(Email email) {
        ...
    }

    public boolean isValid(String value, ConstraintValidatorContext context) {
        ...
    }
}
```

Thus, `EmailValidator` applies to values annotated with `@Email` that are of type `String`. Validators for other Java types can be defined for the same constraint annotation.

16.4.3. Entity Validation

Request entity bodies can be mapped to resource method parameters. There are two ways in which these entities can be validated. If the request entity is mapped to a Java bean whose class is decorated with Bean Validation annotations, then validation can be enabled using @Valid [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/Valid.html] as in Example 16.8, “Entity validation”.

Example 16.8. Entity validation

```
@StandardUser
class User {

    @NotNull
    private String firstName;

    ...
}


@Path("/")
class MyResourceClass {

    @POST
    @Consumes("application/xml")
    public void registerUser(@Valid User user) {
        ...
    }
}
```

In this case, the validator associated with `@StandardUser` (as well as those for non-class level constraints like `@NotNull`) will be called to verify the request entity mapped to `user`.

Alternatively, a new annotation can be defined and used directly on the resource method parameter (Example 16.9, “Entity validation 2”).

Example 16.9. Entity validation 2

```
@Path("/")
class MyResourceClass {

    @POST
    @Consumes("application/xml")
    public void registerUser(@PremiumUser User user) {
        ...
    }
}
```

In the example above, `@PremiumUser` rather than `@StandardUser` will be used to validate the request entity. These two ways in which validation of entities can be triggered can also be combined by including `@Valid` in the list of constraints. The presence of `@Valid` will trigger validation of *all* the constraint annotations decorating a Java bean class.

Response entity bodies returned from resource methods can be validated in a similar manner by annotating the resource method itself. To exemplify, assuming both `@StandardUser` and `@PremiumUser` are required to be checked before returning a user, the `getUser` method can be annotated as shown in Example 16.10, “Response entity validation”.

Example 16.10. Response entity validation

```
@Path("/")
class MyResourceClass {

    @GET
    @Path("{id}")
    @Produces("application/xml")
    @Valid @PremiumUser
    public User getUser(@PathParam("id") String id) {
        User u = findUser(id);
        return u;
    }

    ...
}
```

Note that `@PremiumUser` is explicitly listed and `@StandardUser` is triggered by the presence of the `@Valid` annotation - see definition of `User` class earlier in this section.

16.4.4. Annotation Inheritance

The rules for inheritance of constraint annotation are defined in Bean Validation specification. It is worth noting that these rules are incompatible with those defined by JAX-RS. Generally speaking, constraint annotations in Bean Validation are cumulative (can be strengthen) across a given type hierarchy while JAX-RS annotations are inherited or, overridden and ignored.

For Bean Validation annotations Jersey follows the constraint annotation rules defined in the Bean Validation specification.

16.5. @ValidateOnExecution

According to Bean Validation specification, validation is enabled by default only for the so called *constrained* methods. Getter methods as defined by the Java Beans specification are not constrained methods, so they will not be validated by default. The special annotation `@ValidateOnExecution` can be used to selectively enable and disable validation. For example, you can enable validation on method `getEmail` shown in Example 16.11, “Validate getter on execution”.

Example 16.11. Validate getter on execution

```
@Path("/")
class MyResourceClass {

    @Email
    @ValidateOnExecution
    public String getEmail() {
        return email;
    }

    ...
}
```

The default value for the `type` attribute of `@ValidateOnExecution` is `IMPLICIT` which results in method `getEmail` being validated.

Note

According to Bean Validation specification `@ValidateOnExecution` cannot be overridden once is declared on a method (i.e. in subclass/sub-interface) and in this situations a `ValidationException` should be raised. This default behaviour can be suppressed by setting ServerProperties.BV_DISABLE_VALIDATE_ON_EXECUTABLE_OVERRIDE_CHECK [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#BV_DISABLE_VALIDATE_ON_EXECUTABLE_OVERRIDE_CHECK] property (Jersey specific) to `true`.

16.6. Injecting

Jersey allows you to inject registered resources/providers into your ConstraintValidator [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/ConstraintValidator.html] implementation and you can inject Configuration [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/Configuration.html], ValidatorFactory [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/ValidatorFactory.html] and Validator [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/Validator.html] as required by Bean Validation spec.

Note

Injected Configuration [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/Configuration.html], ValidatorFactory [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/ValidatorFactory.html] and Validator [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/Validator.html] do not inherit configuration provided by ValidationConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/validation/ValidationConfig.html] and need to be configured manually.

Injection of JAX-RS components into `ConstraintValidator`s is supported via a custom `ConstraintValidatorFactory` provided by Jersey. An example is shown in Example 16.12, “Injecting UriInfo into a ConstraintValidator”.

Example 16.12. Injecting UriInfo into a ConstraintValidator

```
public class EmailValidator implements ConstraintValidator<Email, String> {

    @Context
    private UriInfo uriInfo;

    public void initialize(Email email) {
        ...
    }

    public boolean isValid(String value, ConstraintValidatorContext context) {
        // Use UriInfo.

        ...
    }
}
```

Using a custom ConstraintValidatorFactory [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/ConstraintValidatorFactory.html] of your own disables registration of the one

provided by Jersey and injection support for resources/providers (if needed) has to be provided by this new implementation. Example 16.13, “Support for injecting Jersey's resources/providers via ConstraintValidatorFactory.” shows how this can be achieved.

Example 16.13. Support for injecting Jersey's resources/providers via ConstraintValidatorFactory.

```
public class InjectingConstraintValidatorFactory implements ConstraintValidatorFac

    @Context
    private ResourceContext resourceContext;

    @Override
    public <T extends ConstraintValidator<?, ?>> T getInstance(final Class<T> key)
        return resourceContext.getResource(key);
    }

    @Override
    public void releaseInstance(final ConstraintValidator<?, ?> instance) {
        // NOOP
    }
}
```

Note

This behaviour may likely change in one of the next version of Jersey to remove the need of manually providing support for injecting resources/providers from Jersey in your own `ConstraintValidatorFactory` implementation code.

16.7. Error Reporting

Bean Validation specification defines a small hierarchy of exceptions (they all inherit from ValidationException [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/ValidationException.html]) that could be thrown during initialization of validation engine or (for our case more importantly) during validation of input/output values (ConstraintViolationException [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/javax/validation/ConstraintViolationException.html]). If a thrown exception is a subclass of `ValidationException` *except* `ConstraintViolationException` then this exception is mapped to a HTTP response with status code 500 (Internal Server Error). On the other hand, when a `ConstraintViolationException` is throw two different status code would be returned:

- 500 (Internal Server Error)

  If the exception was thrown while validating a method return type.

- 400 (Bad Request)

  Otherwise.

16.7.1. ValidationError

By default, (during mapping `ConstraintViolationException`s) Jersey doesn't return any entities that would include validation errors to the client. This default behaviour could be changed

by enabling ServerProperties.BV_SEND_ERROR_IN_RESPONSE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#BV_SEND_ERROR_IN_RESPONSE] property in your application (Example 16.1, “Configuring Jersey specific properties for Bean Validation.”). When this property is enabled then our custom ExceptionMapper<E extends Throwable> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/ExceptionMapper.html] (that is handling `ValidationException`s) would transform `ConstraintViolationException`(s) into ValidationError [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/validation/ValidationError.html](s) and set this object (collection) as the new response entity which Jersey is able to sent to the client. Four `MediaType`s are currently supported when sending `ValidationError`s to the client:

- `text/plain`
- `text/html`
- `application/xml`
- `application/json`

> **Note**
>
> Note: You need to register one of the JSON (JAXB) providers (e.g. MOXy) to marshall validation errors to JSON.

Let's take a look at ValidationError [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/validation/ValidationError.html] class to see which properties are send to the client:

```
@XmlRootElement
public final class ValidationError {

    private String message;

    private String messageTemplate;

    private String path;

    private String invalidValue;

    ...
}
```

The `message` property is the interpolated error message, `messageTemplate` represents a non-interpolated error message (or key from your constraint definition e.g. `{javax.validation.constraints.NotNull.message}`), `path` contains information about the path in the validated object graph to the property holding invalid value and `invalidValue` is the string representation of the invalid value itself.

Here are few examples of `ValidationError` messages sent to client:

Example 16.14. `ValidationError` to `text/plain`

```
HTTP/1.1 500 Internal Server Error
Content-Length: 114
Content-Type: text/plain
Vary: Accept
Server: Jetty(6.1.24)

Contact with given ID does not exist. (path = ContactCardResource.getContact.<retu
```

Example 16.15. `ValidationError` to `text/html`

```
HTTP/1.1 500 Internal Server Error
Content-Length: ...
Content-Type: text/plain
Vary: Accept
Server: Jetty(6.1.24)

<div class="validation-errors">
    <div class="validation-error">
        <span class="message">Contact with given ID does not exist.</span>
        (
        <span class="path">
            <strong>path</strong>
            = ContactCardResource.getContact.<return value>
        </span>
        ,
        <span class="invalid-value">
            <strong>invalidValue</strong>
            = null
        </span>
        )
    </div>
</div>
```

Example 16.16. `ValidationError` to `application/xml`

```
HTTP/1.1 500 Internal Server Error
Content-Length: ...
Content-Type: text/plain
Vary: Accept
Server: Jetty(6.1.24)

<?xml version="1.0" encoding="UTF-8"?>
<validationErrors>
    <validationError>
        <message>Contact with given ID does not exist.</message>
        <messageTemplate>{contact.does.not.exist}</messageTemplate>
        <path>ContactCardResource.getContact.&lt;return value&gt;</path>
    </validationError>
</validationErrors>
```

Example 16.17. `ValidationError` to `application/json`

```
HTTP/1.1 500 Internal Server Error
Content-Length: 174
Content-Type: application/json
Vary: Accept
Server: Jetty(6.1.24)

[ {
   "message" : "Contact with given ID does not exist.",
   "messageTemplate" : "{contact.does.not.exist}",
   "path" : "ContactCardResource.getContact.<return value>"
} ]
```

16.8. Example

To see a complete wroking example of using Bean Validation (JSR-349) with Jersey refer to the Bean Validation example [https://github.com/jersey/jersey/tree/2.1/examples/bean-validation-webapp].

Chapter 17. MVC Templates

Jersey provides an extension to support the Model-View-Controller (MVC) design pattern. In the context of Jersey components, the Controller from the MVC pattern corresponds to a resource class or method, the View to a template bound to the resource class or method, and the Model to a Java object (or a Java bean) returned from a resource method.

17.1. Dependencies

Jersey MVC templating support is provided by Jersey as a set of (three) extension modules:

- jersey-mvc [https://jersey.java.net/project-info/2.1/jersey/project/jersey-mvc/dependencies.html]

  The base module that provides API and extension SPI for MVC templating support in Jersey. This module is required by any particular MVC templating engine integration module that implements the exposed SPI.

- jersey-mvc-freemarker [https://jersey.java.net/project-info/2.1/jersey/project/jersey-mvc-freemarker/dependencies.html]

  An integration module with Freemarker-based templating engine. The module provides a custom `TemplateProcessor` for Freemarker templates and a set of related engine-specific configuration properties.

- jersey-mvc-jsp [https://jersey.java.net/project-info/2.1/jersey/project/jersey-mvc-jsp/dependencies.html]

  An integration module for JSP-based templating engine. The module provides a custom `TemplateProcessor` for JSP templates, custom tag implementation and a set of related engine-specific configuration properties.

  ### Note

  In a typical set-up projects using the Jersey MVC templating support would depend on the base module that provides the API and SPI and a single templating engine module for the templating engine of your choice. These modules need to be mentioned explicitly in your `pom.xml` file.

If you want to use just templating API infrastructure provided by Jersey for the MVC templating support in order to implement your custom support for a templating engine other than the ones provided by Jersey (JSP/Freemarker), you will need to add the base jersey-mvc [https://jersey.java.net/project-info/2.1/jersey/project/jersey-mvc/dependencies.html] module into the list of your dependencies:

```
<dependency>
    <groupId>org.glassfish.jersey.ext</groupId>
    <artifactId>jersey-mvc</artifactId>
    <version>2.1</version>
</dependency>
```

To use one of the templating engines for which Jersey provides the integration implementation (JSP/Freemarker) in your project, you need to add the jersey-mvc-jsp [https://jersey.java.net/project-info/2.1/jersey/project/jersey-mvc-jsp/dependencies.html] or jersey-mvc-freemarker [https://jersey.java.net/project-info/2.1/jersey/project/jersey-mvc-freemarker/dependencies.html] module to your `pom.xml` respectively:

```
<dependency>
    <groupId>org.glassfish.jersey.ext</groupId>
    <artifactId>jersey-mvc-jsp</artifactId>
    <version>2.1</version>
</dependency>
```

```
<dependency>
    <groupId>org.glassfish.jersey.ext</groupId>
    <artifactId>jersey-mvc-freemarker</artifactId>
    <version>2.1</version>
</dependency>
```

Both of these modules transitively depend on the base `jersey-mvc`, so it is not necessary to add the base `jersey-mvc` module explicitly into your dependency list, however it is a recommended Maven practice to do so.

If you are not using Maven you need to make sure to have all required transitive dependencies (see jersey-mvc [https://jersey.java.net/project-info/2.1/jersey/project/jersey-mvc/dependencies.html]/jersey-mvc-freemarker [https://jersey.java.net/project-info/2.1/jersey/project/jersey-mvc-freemarker/dependencies.html]/jersey-mvc-jsp [https://jersey.java.net/project-info/2.1/jersey/project/jersey-mvc-jsp/dependencies.html]) on the classpath.

17.2. Registration and Configuration

To use capabilities of Jersey MVC templating support in your JAX-RS/Jersey application you need to register Feature [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Feature.html]s provided by the modules mentioned above. For `jersey-mvc` it is MvcFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/mvc/MvcFeature.html], for `jersey-mvc-jsp` it's JspMvcFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/mvc/JspMvcFeature.html] and for `jersey-mvc-freemarker` it is FreemarkerMvcFeature [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/mvc/FreemarkerMvcFeature.html].

Note

Both `JspMvcFeature` and `FreemarkerMvcFeature` also register `MvcFeature` so you don't need to register it explicitly when using these JSP/Freemarker modules.

Example 17.1. Registering `MvcFeature`

```
new ResourceConfig()
    .register(org.glassfish.jersey.server.mvc.MvcFeature.class)
    // Further configuration of ResourceConfig.
    .register( ... );
```

Example 17.2. Registering `JspMvcFeature`

```
new ResourceConfig()
    .register(org.glassfish.jersey.server.mvc.jsp.JspMvcFeature.class)
    // Further configuration of ResourceConfig.
    .register( ... );
```

Important

Jersey web applications that want to use MVC templating support feature should be registered as Servlet filters rather than Servlets in the application's `web.xml`. The `web.xml`-less deployment

style introduced in Servlet 3.0 is not supported at the moment for web applications that require use of Jersey MVC templating support.

Each of the three MVC modules contains a `*Properties` (e.g. `FreemarkerMvcProperties`) file which defines a set of properties that could be set in a JAX-RS `Application` / `ResourceConfig` in order to take effect, see the Example 17.3, “Setting `MvcProperties.TEMPLATE_BASE_PATH` value in `ResourceConfig`” and Example 17.4, “Setting `FreemarkerMvcProperties.TEMPLATE_BASE_PATH` value in `web.xml`”.

Following list contains description of the available properties:

- `MvcProperties.TEMPLATE_BASE_PATH` - "`jersey.config.server.mvc.templateBasepath`"

  The base path where templates are located.

- `FreemarkerMvcProperties.TEMPLATE_BASE_PATH` - "`jersey.config.server.mvc.templateBasepath.freemarker`"

  The base path where Freemarker templates are located.

- `JspMvcProperties.TEMPLATE_BASE_PATH` - "`jersey.config.server.mvc.templateBasepath.jsp`"

  The base path where JSP templates are located.

Example 17.3. Setting `MvcProperties.TEMPLATE_BASE_PATH` value in `ResourceConfig`

```
new ResourceConfig()
    .property(MvcProperties.TEMPLATE_BASE_PATH, "templates")
    .register(MvcFeature.class)
    // Further configuration of ResourceConfig.
    .register( ... );
```

Example 17.4. Setting `FreemarkerMvcProperties.TEMPLATE_BASE_PATH` value in `web.xml`

```
<servlet>
    <servlet-name>org.glassfish.jersey.examples.freemarker.MyApplication</servlet-
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
        <param-name>javax.ws.rs.Application</param-name>
        <param-value>org.glassfish.jersey.examples.freemarker.MyApplication</param
    </init-param>
    <init-param>
        <param-name>jersey.config.server.mvc.templateBasePath.freemarker</param-na
        <param-value>freemarker</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

17.3. Explicit vs. Implicit View Templates

Note

Some of the passages/examples from this and the next section was taken from MVCJ [https://blogs.oracle.com/sandoz/entry/mvcj] blog article written by Paul Sandoz earlier.

In Jersey 2.0, the base MVC API (excluding the SPI part) consists of two classes (in the `org.glassfish.jersey.server.mvc` package in base MVC module) that we will explore in more detail now, namely Viewable [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/mvc/Viewable.html] and @Template [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/mvc/Template.html]. These classes determines which approach (explicit/implicit) you would be taking when working with Jersey MVC templating support.

17.3.1. `Viewable` - Explicit View Templates

In this approach a resource method explicitly returns a reference to a view template and the data model to be used. For this purpose the Viewable [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/mvc/Viewable.html] class has been introduced in Jersey 1 and is also present (under a different package) in Jersey 2. A simple example of usage can be seen in Example 17.5, “Using `Viewable` in a resource class”.

Example 17.5. Using `Viewable` in a resource class

```
package com.foo;

@Path("foo")
public class Foo {

    @GET
    public Viewable get() {
        return new Viewable("index", "FOO");
    }
}
```

In this example, the `Foo` JAX-RS resource class is the controller and the `Viewable` instance encapsulates the provided data model (`"FOO"` string) and a named reference to the associated view template (`"index"`).

The template name reference `"index"` is a relative value that Jersey will resolve to its absolute template reference using the fully qualified class name of `Foo` (more on resolving relative template name to the absolute one can be found in the JavaDoc of Viewable [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/mvc/Viewable.html] class), which, in our case, is:

```
"/com/foo/Foo/index"
```

Jersey will then search all the registered template processors (see Section 17.5, “Custom Templating Engines”) to find a template processor that can resolve the absolute template reference further to a "processable" template reference. If a template processor is found then the "processable" template is processed using the supplied data model.

Let's change the resource `GET` method in our `Foo` resource a little:

Example 17.6. Using absolute path to template in `Viewable`

```
@GET
public Viewable get() {
    return new Viewable("/index", "FOO");
}
```

In this case, since the template reference begins with `"/"`, Jersey will consider the reference to be absolute already and will not attempt to absolutize it again. The reference will be used "as is" when resolving it to a "processable" template reference as described earlier.

Tip

All HTTP methods may return `Viewable` instances. Thus a `POST` method may return a template reference to a template that produces a view that is the result of processing an HTML Form [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Form.html].

17.3.2. `@Template` - Implicit View Templates

17.3.2.1. Resource classes

A resource class can have templates implicitly associated with it via @Template [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/mvc/Template.html] annotation. For example, take a look at the resource class listing in Example 17.7, “Using `@Template` on a resource class”.

Example 17.7. Using `@Template` on a resource class

```
@Path("foo")
@Template
public class Foo {

    public String getFoo() {
        return "FOO";
    }
}
```

The example above uses a lot of conventions and requires some more explanation. First of all, you may have noticed that there is no resource method defined in this JAX-RS resource. Also, there is no template reference defined. In this case, since the `@Template` annotation placed on the resource class does not contain any information, the default relative template reference `"index"` will be used. Later it will get resolved to an absolute `"/com/foo/Foo/index"` template reference. As for the missing resource methods, a default `@GET` method will be implicitly generated by Jersey for the `Foo` resource (our MVC Controller). The implementation of the implicitly added resource method performs the equivalent of the following explicit resource method:

```
@GET
public Viewable get() {
    return new Viewable("index", this);
}
```

As you can see, the resource class serves in this case also as a model. Producible media types are determined based on the `@Produces` annotation declared on the resource class, if any.

Note

In case of a resource class-based implicit MVC view template, the controller is also the model. In this case the template reference `"index"` is special, it is the template reference associated with the controller itself.

Implicit sub-resource templates are also supported, for example, for a template reference `"bar"` that resolves to an absolute template reference `"/com/foo/Foo/bar"` that in turn resolves to a processable template reference. Following `@GET` method is also implicitly added to the `Foo` controller that performs the equivalent of the following explicit sub-resource method:

```
@GET
                        @Path("{implicit-view-path-parameter}")
                        public Viewable get(@PathParameter("{implicit-view-path-pa
                        return new Viewable(template, this);
                        }
```

In other words, a HTTP `GET` request to a `"/foo/bar"` would be handled by this auto-generated method in the `Foo` resource and would delegate the request to a registered template processor supports processing of the absolute template reference `"/com/foo/Foo/bar"`, while the model is still an instance of the JAX-RS resource class `Foo`.

17.3.2.2. Resource methods

In case a resource method is annotated with @Template [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/mvc/Template.html] annotation then the return value of the method defines the MVC model part. The processing of such a method is then essentially the same as if the return type of the method was an instance of the Viewable [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/mvc/Viewable.html] class. If a method is annotated with `@Template` and is also returning a `Viewable` instance then the values (`resolvingClass`) from the `Viewable` instance take precedence over those defined in the annotation. Producible media types are determined from the method's `@Produces` annotation.

Note

Implicit view templates support works dynamically (as is the case for explicit MVC) so it is possible (if the deployment system is configured correctly) to add or modify templates while the application is running.

17.4. JSP

As stated earlier, Jersey provides support for JSP templates in jersey-mvc-jsp [https://jersey.java.net/project-info/2.1/jersey/project/jersey-mvc-jsp/dependencies.html] extension module. There is a JSP template processor that resolves absolute template references to processable template references represented as JSP pages as follows:

Procedure 17.1. Resolving JSP processable template reference

1. if the absolute template reference does not end in `".jsp"` append this suffix to the reference; and
2. if `Servlet.getResource` returns a non-`null` value for the appended reference then return the appended reference as the processable template reference otherwise return `null` (to indicate the absolute reference has not been resolved by the JSP template processor).

Thus the absolute template reference `"/com/foo/Foo/index"` would be resolved to `"/com/foo/Foo/index.jsp"`, provided there exists a `"/com/foo/Foo/index.jsp"` JSP page in the web application.

Jersey will assign the model instance to the attribute named "`it`". So in the case of the implicit example it is possible to referece the `foo` property on the `Foo` resource from the JSP template as follows:

```
<h1>${it.foo}</h1>
```

17.5. Custom Templating Engines

To add support for other (custom) templating engines into Jersey MVC Templating facility, you need to implement the TemplateProcessor [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/mvc/TemplateProcessor.html] and register this class into your application.

Tip

When writing template processors it is recommend that you use an appropriate unique suffix for the processable template references. In such case it is then possible to easily support mixing of multiple templating engines in a single application without any conflicts.

Example 17.8. Custom TemplateProcessor [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/mvc/TemplateProcessor.html]

```
@Provider
class MyTemplateProcessor implements TemplateProcessor<String> {

    @Override
    public String resolve(String path, final MediaType mediaType) {
        final String extension = ".testp";

        if (!path.endsWith(extension)) {
            path = path + extension;
        }

        final URL u = this.getClass().getResource(path);
        return u == null ? null : path;
    }

    @Override
    public void writeTo(String templateReference,
                        Viewable viewable,
                        MediaType mediaType,
                        OutputStream out) throws IOException {
        final PrintStream ps = new PrintStream(out);
        ps.print("path=");
        ps.print(templateReference);
        ps.println();
        ps.print("model=");
        ps.print(viewable.getModel().toString());
        ps.println();
    }

}
```

Example 17.9. Registering custom TemplateProcessor [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/mvc/TemplateProcessor.html]

```
new ResourceConfig()
    .register(MyTemplateProcessor.class)
    // Further configuration of ResourceConfig.
    .register( ... );
```

17.6. Other Examples

To see an example of MVC (JSP) templating support in Jersey refer to the MVC (Bookstore) example [https://github.com/jersey/jersey/tree/2.1/examples/bookstore-webapp].

Chapter 18. Jersey Test Framework

Jersey Test Framework originated as an internal tool used for verifying the correct implementation of server-side components. Testing RESTful applications became a more pressing issue with "modern" approaches like test-driven development and users started to look for a tool that could help with designing and running the tests as fast as possible but with many options related to test execution environment.

Current implementation of Jersey Test Framework supports the following set of features:

- pre-configured client to access deployed application
- support for multiple containers - grizzly, in-memory, jdk, simple
- able to run against any external container
- automated configurable traffic logging

Jersey Test Framework is based on JUnit and works almost out-of-the box. It is easy to integrate it within your Maven-based project. While it is usable on all environments where you can run JUnit, we support primarily the Maven-based setups.

18.1. Basics

```
 1 public class SimpleTest extends JerseyTest {
 2
 3     @Path("hello")
 4     public static class HelloResource {
 5         @GET
 6         public String getHello() {
 7             return "Hello World!";
 8         }
 9     }
10
11     @Override
12     protected Application configure() {
13         return new ResourceConfig(HelloResource.class);
14     }
15
16     @Test
17     public void test() {
18         final String hello = target("hello").request().get(String.class);
19         assertEquals("Hello World!", hello);
20     }
21 }
```

If you want to develop a test using Jersey Test Framework, you need to subclass JerseyTest [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/org/glassfish/jersey/test/JerseyTest.html] and configure the set of resources and/or providers that will be deployed as part of the test application. This short code snippet shows basic resource class `HelloResource` used in tests defined as part of the `SimpleTest` class. The overridden `configure` method returns a ResourceConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ResourceConfig.html] of the test application,that contains only the `HelloResource` resource class. `ResourceConfig` is a sub-class of

JAX-RS Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html]. It is a Jersey convenience class for configuring JAX-RS applications. `ResourceConfig` also implements JAX-RS Configurable [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Configurable.html] interface to make the application configuration more flexible.

18.2. Supported Containers

JerseyTest [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/org/glassfish/jersey/test/JerseyTest.html] supports deploying applications on various containers, all (except the external container wrapper) need to have some "glue" code to be supported. Currently Jersey Test Framework provides support for Grizzly, In-Memory, JDK (`com.sun.net.httpserver.HttpServer`) and Simple HTTP container (`org.simpleframework.http`).

A test container is selected based on various inputs. JerseyTest#getTestContainerFactory() [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/test/JerseyTest.html#getTestContainerFactory()] is always executed, so if you override it and provide your own version of TestContainerFactory [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/org/glassfish/jersey/test/spi/TestContainerFactory.html], nothing else will be considered. Setting a system variable TestProperties#CONTAINER_FACTORY [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/test/TestProperties.html#CONTAINER_FACTORY] has similar effect. This way you may defer the decision on which containers you want to run your tests from the compile time to the test execution time. Default implementation of `TestContainerFactory` looks for container factories on classpath. If more than one instance is found and there is a Grizzly test container factory among them, it will be used; if not, a warning will be logged and the first found factory will be instantiated.

Following is a brief description of all containers supported in Jersey Test Framework.

- Grizzly container can run as a light-weight, plain HTTP container. Almost all Jersey tests are using Grizzly by default.

```
<dependency>
    <groupId>org.glassfish.jersey.test-framework.providers</groupId>
    <artifactId>jersey-test-framework-provider-grizzly2</artifactId>
    <version>2.1</version>
</dependency>
```

- In-Memory container is not a real container. It starts Jersey application and directly calls internal APIs to handle request created by client provided by test framework. There is no network communication involved. This containers does not support servlet and other container dependent features, but it is a perfect choice for simple unit tests.

```
<dependency>
    <groupId>org.glassfish.jersey.test-framework.providers</groupId>
    <artifactId>jersey-test-framework-provider-inmemory</artifactId>
    <version>2.1</version>
</dependency>
```

- `HttpServer` from Oracle JDK is another supported test container.

```
<dependency>
    <groupId>org.glassfish.jersey.test-framework.providers</groupId>
    <artifactId>jersey-test-framework-provider-jdk-http</artifactId>
    <version>2.1</version>
</dependency>
```

- Simple container (`org.simpleframework.http`) is another light-weight HTTP container that integrates with Jersey and is supported by Jersey Test Framework.

```
<dependency>
    <groupId>org.glassfish.jersey.test-framework.providers</groupId>
    <artifactId>jersey-test-framework-provider-simple</artifactId>
    <version>2.1</version>
</dependency>
```

18.3. Advanced features

18.3.1. `JerseyTest` Features

`JerseyTest` provide enable(...) [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/test/JerseyTest.html#enable(java.lang.String)], forceEnable(...) [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/test/JerseyTest.html#forceEnable(java.lang.String)] and disable(...) [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/test/JerseyTest.html#disable(java.lang.String)] methods, that give you control over configuring values of the properties defined and described in the `TestProperties` class. A typical code that overrides the default property values is listed bellow:

```
 1 public class SimpleTest extends JerseyTest {
 2     // ...
 3
 4     @Override
 5     protected Application configure() {
 6         enable(TestProperties.LOG_TRAFFIC);
 7         enable(TestProperties.DUMP_ENTITY);
 8
 9         // ...
10
11     }
12 }
```

The code in the example above enables test traffic logging (inbound and outbound headers) as well as dumping the HTTP message entity as part of the traffic logging.

18.3.2. External container

Complicated test scenarios may require fully started containers with complex setup configuration, that is not easily doable with current Jersey container support. To address these use cases, Jersey Test Framework providers general fallback mechanism - an External Test Container Factory. Support of this external container "wrapper" is provided as the following module:

```
<dependency>
    <groupId>org.glassfish.jersey.test-framework.providers</groupId>
    <artifactId>jersey-test-framework-provider-external</artifactId>
    <version>2.1</version>
</dependency>
```

As indicated, the "container" exposed by this module is just a wrapper or stub, that redirects all request to a configured host and port. Writing tests for this container is same as for any other but you have to provide the information about host and port during the test execution:

```
mvn test -Djersey.test.host=myhost.org -Djersey.config.test.container.port=8080
```

18.3.3. Test Client configuration

Tests might require some advanced client configuration. This is possible by overriding configureClient(ClientConfig clientConfig) [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/test/JerseyTest.html#configureClient(org.glassfish.jersey.client.ClientConfig)] method. Typical use case for this is registering more providers, such as MessageBodyReader<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyReader.html]s or MessageBodyWriter<T> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/MessageBodyWriter.html]s, or enabling additional features.

18.3.4. Accessing the logged test records programmatically

Sometimes you might need to check a logged message as part of your test assertions. For this purpose Jersey Test Framework provides convenient access to the logged records via JerseyTest#getLastLoggedRecord() [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/test/JerseyTest.html#getLastLoggedRecord()] and JerseyTest#getLoggedRecords() [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/test/JerseyTest.html#getLoggedRecords()] methods. Note that this feature is not enabled by default, see TestProperties#RECORD_LOG_LEVEL [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/test/TestProperties.html#RECORD_LOG_LEVEL] for more information.

Chapter 19. Building and Testing Jersey

19.1. Checking Out the Source

Jersey source code is available on GitHub. You can browse the sources at https://github.com/jersey/jersey.

In case you are not familiar with Git, we recommend reading some of the many "Getting Started with Git" articles you can find on the web. For example this DZone RefCard [http://refcardz.dzone.com/refcardz/getting-started-git].

To clone the Jersey repository you can execute the following command on the command-line (provided you have a command-line Git client installed on your machine):

```
git clone git://github.com/jersey/jersey.git
```

This creates read-only copy of Jersey workspace. If you want to contribute, please use "pull request": https://help.github.com/articles/creating-a-pull-request.

Milestones and releases of Jersey are tagged. You can list the tags by executing the standard Git command in the repository directory:

```
git tag -l
```

or by visiting https://github.com/jersey/jersey/tags.

19.2. Building the Source

Jersey source code requires Java SE 6 or greater. The build is based on Maven. Maven 3 or greater is highly recommended. Also it is recommended you use the following Maven options when building the workspace (can be set in `MAVEN_OPTS` environment variable):

```
-Xmx1048m -XX:PermSize=64M -XX:MaxPermSize=128M
```

It is recommended to build all of Jersey after you cloned the source code repository. To do that execute the following commands in the directory where jersey source repository was cloned (typically the directory named "jersey"):

```
mvn -Dmaven.test.skip=true clean install
```

This command will build Jersey, but skip the test execution. If you don't want to skip the tests, execute the following instead:

```
mvn clean install
```

Building the whole Jersey project including tests could take significant amount of time.

19.3. Testing

Jersey contains many tests. Unit tests are in the individual Jersey modules, integration and end-to-end tests are in `jersey/tests/e2e` directory. You can run tests related to a particular area using the following command:

```
mvn -Dtest=<pattern> test
```

where `pattern` may be a comma separated set of names matching tests classes or individual methods (like `LinkTest#testDelimiters`).

19.4. Using NetBeans

NetBeans IDE [http://netbeans.org] has excellent maven support. The Jersey maven modules can be loaded, built and tested in NetBeans without any additional NetBeans-specific project files.

Chapter 20. Migrating from Jersey 1.x

This chapter is a migration guide for people switching from Jersey 1.x. Since many of the Jersey 1.x features became part of JAX-RS 2.0 standard which caused changes in the package names, we decided it is a good time to do a more significant incompatible refactoring, which will allow us to introduce some more interesting new features in the future. As the result, there are many incompatiblities between Jersey 1.x and Jersey 2.0. This chapter summarizes how to migrate the concepts found in Jersey 1.x to Jersey/ JAX-RS 2.0 concepts.

20.1. Server API

Jersey 1.x contains number of proprietary server APIs. This section covers migration of application code relying on those APIs.

20.1.1. Injecting custom objects

Jersey 1.x have its own internal dependency injection framework which handles injecting various parameters into field or methods. It also provides a way how to register custom injection provider in Singleton or PerRequest scopes. Jersey 2.x uses HK2 as dependency injection framework and users are also able to register custom classes or instances to be injected in various scopes.

Main difference in Jersey 2.x is that you don't need to create special classes or providers for this task; everything should be achievable using HK2 API. Custom injectables can be registered at ResourceConfig level by adding new HK2 Module or by dynamically adding binding almost anywhere using injected HK2 Services instance.

Jersey 1.x Singleton:

```
ResourceConfig resourceConfig = new DefaultResourceConfig();
resourceConfig.getSingletons().add(
        new SingletonTypeInjectableProvider<Context, SingletonType>(
                SingletonType.class, new SingletonType()) {});
```

Jersey 1.x PerRequest:

```
ResourceConfig resourceConfig = new DefaultResourceConfig();
resourceConfig.getSingletons().add(
        new PerRequestTypeInjectableProvider<Context, PerRequestType>() {
            @Override
            public Injectable<PerRequestType> getInjectable(ComponentContext ic, C
                //...
            }
        });
```

Jersey 2.0 HK2 Module:

```
public static class MyBinder extends AbstractBinder {

    @Override
    protected void configure() {
        // request scope binding
```

```
        bind(MyInjectablePerRequest.class).to(MyInjectablePerRequest.class).in(Req
        // singleton binding
        bind(MyInjectableSingleton.class).in(Singleton.class);
        // singleton instance binding
        bind(new MyInjectableSingleton()).to(MyInjectableSingleton.class);
    }

}

// register module to ResourceConfig (can be done also in constructor)
ResourceConfig rc = new ResourceConfig();
rc.addClasses(/* ... */);
rc.addBinders(new MyBinder());
```

Jersey 2.0 dynamic binding:

```
public static class MyApplication extends Application {

    @Inject
    public MyApplication(ServiceLocator serviceLocator) {
        System.out.println("Registering injectables...");

        DynamicConfiguration dc = Injections.getConfiguration(serviceLocator);

        // request scope binding
        Injections.addBinding(
        Injections.newBinder(MyInjectablePerRequest.class).to(MyInjectablePerReque
                dc);

        // singleton binding
        Injections.addBinding(
                Injections.newBinder(MyInjectableSingleton.class)
                        .to(MyInjectableSingleton.class)
                        .in(Singleton.class),
                dc);

        // singleton instance binding
        Injections.addBinding(
                Injections.newBinder(new MyInjectableSingleton())
                        .to(MyInjectableSingleton.class),
                dc);

        // request scope binding with specified custom annotation
        Injections.addBinding(
                Injections.newBinder(MyInjectablePerRequest.class)
                        .to(MyInjectablePerRequest.class)
                        .qualifiedBy(new MyAnnotationImpl())
                        .in(RequestScoped.class),
                dc);

        // commits changes
        dc.commit();
    }
```

```
@Override
public Set<Class<?>> getClasses() {
    return ...
}}
```

20.1.2. ResourceConfig Reload

In Jersey 1, the reload functionality is based on two interfaces:

1. com.sun.jersey.spi.container.ContainerListener

2. com.sun.jersey.spi.container.ContainerNotifier

Containers, which support the reload functionality implement the `ContainerListener` interface, so that once you get access to the actual container instance, you could call it's `onReload` method and get the container re-load the config. The second interface helps you to obtain the actual container instance reference. An example on how things are wired together follows.

Example 20.1. Jersey 1 reloader implementation

```
 1 public class Reloader implements ContainerNotifier {
 2     List<ContainerListener> ls;
 3
 4     public Reloader() {
 5         ls = new ArrayList<ContainerListener>();
 6     }
 7
 8     public void addListener(ContainerListener l) {
 9         ls.add(l);
10     }
11
12     public void reload() {
13         for (ContainerListener l : ls) {
14             l.onReload();
15         }
16     }
17 }
```

Example 20.2. Jersey 1 reloader registration

```
1 Reloader reloader = new Reloader();
2 resourceConfig.getProperties().put(ResourceConfig.PROPERTY_CONTAINER_NOTIFIER,
```

In Jersey 2, two interfaces are involved again, but these have been re-designed.

1. org.glassfish.jersey.server.spi.Container

2. org.glassfish.jersey.server.spi.ContainerLifecycleListener

The `Container` interface introduces two `reload` methods, which you can call to get the application re-loaded. One of these methods allows to pass in a new `ResourceConfig` instance. You can register your implementation of `ContainerLifecycleListener` the same way as any other provider (i.e. either by annotating it by @Provider [http://jax-rs-spec.java.net/nonav/2.0/

apidocs/javax/ws/rs/ext/Provider.html] annotation or adding it to the Jersey ResourceConfig [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ResourceConfig.html] directly either using the class (using `ResourceConfig.addClasses()`) or registering a particular instance using `ResourceConfig.addSingletons()` method.

An example on how things work in Jersey 2 follows.

Example 20.3. Jersey 2 reloader implementation

```
 1 public class Reloader implements ContainerLifecycleListener {
 2
 3     Container container;
 4
 5     public void reload(ResourceConfig newConfig) {
 6         container.reload(newConfig);
 7     }
 8
 9     public void reload() {
10         container.reload();
11     }
12
13     @Override
14     public void onStartup(Container container) {
15         this.container = container;
16     }
17
18     @Override
19     public void onReload(Container container) {
20         // ignore or do whatever you want after reload has been done
21     }
22
23     @Override
24     public void onShutdown(Container container) {
25         // ignore or do something after the container has been shutdown
26     }
27 }
```

Example 20.4. Jersey 2 reloader registration

```
1 Reloader reloader = new Reloader();
2 resourceConfig.addSingletons(reloader);
3
```

20.1.3. MessageBodyReaders and MessageBodyWriters ordering

JAX-RS 2.0 defines new order of MessageBodyWorkers - whole set is sorted by declaration distance, media type and source (custom providers having higher priority than default ones provided by Jersey). JAX-RS 1.x ordering can still be forced by setting parameter `MessageProperties.LEGACY_WORKERS_ORDERING` (`"jersey.config.workers.legacyOrdering"`) to `true` in `ResourceConfig` or `ClientConfig` properties.

20.2. Migrating Jersey Client API

JAX-RS 2.0 provides functionality that is equivalent to the Jersey 1.x proprietary client API. Here is a rough mapping between the Jersey 1.x and JAX-RS 2.0 Client API classes:

Table 20.1. Mapping of Jersey 1.x to JAX-RS 2.0 client classes

Jersey 1.x Class	JAX-RS 2.0 Class	Notes
com.sun.jersey.api.client.Client [http://jersey.java.net/nonav/apidocs/1.17/jersey/com/sun/jersey/api/client/Client.html]	ClientBuilder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientBuilder.html]	For the static factory methods and constructors.
	Client [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/Client.html]	For the instance methods.
com.sun.jersey.api.client.WebResource [http://jersey.java.net/nonav/apidocs/1.17/jersey/com/sun/jersey/api/client/WebResource.html]	WebTarget [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/WebTarget.html]	
com.sun.jersey.api.client.AsyncWebResource [http://jersey.java.net/nonav/apidocs/1.17/	WebTarget [http://jax-rs-spec.java.net/	You can access async versions of the async methods by calling `WebTarget.request().async()`

Jersey 1.x Class	JAX-RS 2.0 Class
com.sun.jersey.api.client.WebResource	javax.ws.rs.client.WebTarget [https://jersey.java.net/nonav/apidocs/2.0/jersey/javax/ws/rs/client/WebTarget.html]

The following sub-sections show code examples.

20.2.1. Making a simple client request

Jersey 1.x way:

```
Client client = Client.create();
WebResource webResource = client.resource(restURL).path("myresource/{param}");
String result = webResource.pathParam("param", "value").get(String.class);
```

JAX-RS 2.0 way:

```
Client client = ClientFactory.newClient();
WebTarget target = client.target(restURL).path("myresource/{param}");
String result = target.pathParam("param", "value").get(String.class);
```

20.2.2. Registering filters

Jersey 1.x way:

```
Client client = Client.create();
WebResource webResource = client.resource(restURL);
webResource.addFilter(new HTTPBasicAuthFilter(username, password));
```

JAX-RS 2.0 way:

```
Client client = ClientFactory.newClient();
WebTarget target = client.target(restURL);
target.register(new HttpBasicAuthFilter(username, password));
```

20.2.3. Setting "Accept" header

Jersey 1.x way:

```
Client client = Client.create();
WebResource webResource = client.resource(restURL).accept("text/plain");
ClientResponse response = webResource.get(ClientResponse.class);
```

JAX-RS 2.0 way:

```
Client client = ClientFactory.newClient();
WebTarget target = client.target(restURL);
```

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

20.2.4. Attaching entity to request

Jersey 1.x way:

```
Client client = Client.create();
WebResource webResource = client.resource(restURL);
ClientResponse response = webResource.post(ClientResponse.class, "payload");
```

JAX-RS 2.0 way:

```
Client client = ClientFactory.newClient();
WebTarget target = client.target(restURL);
Response response = target.request().post(Entity.text("payload"));
```

20.2.5. Setting SSLContext and/or HostnameVerifier

Jersey 1.x way:

```
HTTPSProperties prop = new HTTPSProperties(hostnameVerifier, sslContext);
DefaultClientConfig dcc = new DefaultClientConfig();
dcc.getProperties().put(HTTPSProperties.PROPERTY_HTTPS_PROPERTIES, prop);
Client client = Client.create(dcc);
```

Jersey 2.0 way:

```
Client client = ClientBuilder.newBuilder()
        .sslContext(sslContext)
        .hostnameVerifier(hostnameVerifier)
        .build();
```

Appendix A. Configuration Properties

A.1. Common (client/server) configuration properties

List of common configuration properties that can be found in CommonProperties [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/CommonProperties.html] class. All of these properties can be overridden by their server/client counterparts.

Table A.1. List of common configuration properties

Constant	Value	Description
CommonProperties.FEATURE_AUTO_DISCOVERY_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/CommonProperties.html#FEATURE_AUTO_DISCOVERY_DISABLE]	jersey.config.disableAutoDiscovery	Disable/enable auto discovery globally on client/server. Default value is `false`.
CommonProperties.JSON_PROCESSING_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/CommonProperties.html#JSON_PROCESSING_FEATURE_DISABLE]	jersey.config.disableJsonProcessing	Disable/enable configuration of Json Processing (JSR-353) feature. Default value is `false`.
CommonProperties.METAINF_SERVICES_LOOKUP_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/CommonProperties.html#METAINF_SERVICES_LOOKUP_DISABLE]	jersey.config.disableMetainfServicesLookup	Disable/enable META-INF/services lookup globally on client/server. Default value is `false`.
CommonProperties.MOXY_JSON_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/CommonProperties.html#MOXY_JSON_FEATURE_DISABLE]	jersey.config.disableMoxyJson	Disable/enable configuration of MOXy Json feature. Default value is `false`.
CommonProperties.OUTBOUND_CONTENT_LENGTH_BUFFER [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/CommonProperties.html#OUTBOUND_CONTENT_LENGTH_BUFFER]	jersey.config.contentLength.buffer	An integer value that defines the buffer size used to buffer the outbound message entity in order to determine its size and set the value of HTTP `Content-Length` header. Default value is `8192`.

A.2. Server configuration properties

List of server configuration properties that can be found in ServerProperties [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html] class.

Table A.2. List of server configuration properties

Constant	Value	Description
ServerProperties.BV_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#BV_FEATURE_DISABLE]	jersey.config.beanValidation.disable	Disables Bean Validation support. Default value is `false`.
ServerProperties.BV_DISABLE_VALIDATE_ON_EXECUTABLE_OVERRIDE_CHECK [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#BV_DISABLE_VALIDATE_ON_EXECUTABLE_OVERRIDE_CHECK]	jersey.config.beanValidation.disable.validateOnExecutableOverrideCheck	Disables `@ValidateOnExecution` check. Default value is `false`.
ServerProperties.BV_SEND_ERROR_IN_RESPONSE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#BV_SEND_ERROR_IN_RESPONSE]	jersey.config.beanValidation.enableOutputValidationErrorEntity.server	Enables sending validation error information to the client. Default value is `false`.
ServerProperties.FEATURE_AUTO_DISCOVERY_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#FEATURE_AUTO_DISCOVERY_DISABLE]	jersey.config.disableAutoDiscovery	Disables feature auto discovery on server. Default value is `false`.
ServerProperties.HTTP_METHOD_OVERRIDE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#HTTP_METHOD_OVERRIDE]	jersey.config.server.httpMethodOverride	Defines configuration of HTTP method overriding. This property is used by HttpMethodOverrideFilter [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/filter/HttpMethodOverrideFilter.html] to determine where it should look for method override information (e.g. request header or query parameters).
ServerProperties.JSON_PROCESSING_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#JSON_PROCESSING_FEATURE_DISABLE]	jersey.config.disableJsonProcessing	Disables configuration of Json Processing (JSR-353) feature. Default value is `false`.
ServerProperties.LANGUAGE_MAPPINGS [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#LANGUAGE_MAPPINGS]	jersey.config.server.languageMappings	Defines mappings of URI extensions to languages. The property is used by UriConnegFilter [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/filter/UriConnegFilter.html].
ServerProperties.MEDIA_TYPE_MAPPINGS [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/	jersey.config.server.mediaTypeMappings	Defines mappings of URI extensions to media types. The property is used by UriConnegFilter [https://

Constant	Value	Description
jersey/server/ ServerProperties.html#MEDIA_TYPE_MAPPINGS]		jersey.java.net/apidocs/2.1/jersey/ org/glassfish/jersey/server/filter/ UriConnegFilter.html].
ServerProperties.METAINF_SERVICES_LOOKUP_DISABLE [https://jersey.java.net/ apidocs/2.1/jersey/org/glassfish/ jersey/server/ ServerProperties.html#METAINF_SERVICES_LOOKUP_DISABLE]	jersey.config.server.disableMetainfServicesLookup	Disables META-INF/services lookup on server. Default value is `false`.
ServerProperties.MOXY_JSON_FEATURE_DISABLE [https://jersey.java.net/ apidocs/2.1/jersey/org/glassfish/ jersey/server/ ServerProperties.html#MOXY_JSON_FEATURE_DISABLE]	jersey.config.disableMoxyJson	Disables configuration of MOXy Json feature. Default value is `false`.
ServerProperties.OUTBOUND_CONTENT_LENGTH_BUFFER [https://jersey.java.net/ apidocs/2.1/jersey/org/glassfish/ jersey/server/ ServerProperties.html#OUTBOUND_CONTENT_LENGTH_BUFFER] (Jersey 2.2 or later)	jersey.config.contentLength.buffer	An integer value that defines the buffer size used to buffer the outbound message entity in order to determine its size and set the value of HTTP `Content-Length` header. Default value is `8192`.
ServerProperties.PROVIDER_CLASSNAMES [https://jersey.java.net/ apidocs/2.1/jersey/org/glassfish/ jersey/server/ ServerProperties.html#PROVIDER_CLASSNAMES]	jersey.config.server.provider.classnames	Defines one or more class names that implement application-specific resources and providers. If the property is set, the specified classes will be instantiated and registered as either application JAX-RS root resources or providers.
ServerProperties.PROVIDER_CLASSPATH [https://jersey.java.net/ apidocs/2.1/jersey/org/glassfish/ jersey/server/ ServerProperties.html#PROVIDER_CLASSPATH]	jersey.config.server.provider.classpath	Defines a classpath that contains application-specific resources and providers. If the property is set, the specified packages will be scanned for JAX-RS root resources and providers.
ServerProperties.PROVIDER_PACKAGES [https://jersey.java.net/ apidocs/2.1/jersey/org/glassfish/ jersey/server/ ServerProperties.html#PROVIDER_PACKAGES]	jersey.config.server.provider.packages	Defines one or more packages that contain application-specific resources and providers. If the property is set, the specified packages will be scanned for JAX-RS root resources and providers.
ServerProperties.PROVIDER_SCANNING_RECURSIVE [https://jersey.java.net/ apidocs/2.1/jersey/org/glassfish/ jersey/server/ ServerProperties.html#PROVIDER_SCANNING_RECURSIVE]	jersey.config.server.provider.scanning.recursive	Provider scanning strategy for package scanning. Default value is `true`.
ServerProperties.RESOURCE_VALIDATION_DISABLE [https://jersey.java.net/ apidocs/2.1/jersey/org/glassfish/	jersey.config.server.resource.validation.disable	Disable resource validation. Default value is `false`.

Constant	Value	Description
jersey/server/ ServerProperties.html#RESOURCE_VALIDATION_DISABLE]		
ServerProperties.RESOURCE_VALIDATION_IGNORE_ERRORS [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#RESOURCE_VALIDATION_IGNORE_ERRORS]	jersey.config.server.resource.validation.ignoreErrors	Resource validation validation of application resource models should fail even in case of a fatal validation errors. Default value is false.
ServerProperties.WADL_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#WADL_FEATURE_DISABLE]	jersey.config.server.wadl.disableWadl	Disables WADL generation. Default value is false.
ServerProperties.WADL_GENERATOR_CONFIG [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/ServerProperties.html#WADL_GENERATOR_CONFIG]	jersey.config.server.wadl.generatorConfig	Define generator configuration that provides a WadlGenerator [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/server/wadl/WadlGenerator.html].

A.3. Client configuration properties

List of client configuration properties that can be found in ClientProperties [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html] class.

Table A.3. List of client configuration properties

Constant	Value	Description
ClientProperties.ASYNC_THREADPOOL_SIZE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html#ASYNC_THREADPOOL_SIZE]	jersey.config.client.async.threadPoolSize	Async thread pool size. Default value is not set. *NOT SUPPORTED.*
ClientProperties.BUFFER_RESPONSE_ENTITY_ON_EXCEPTION [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html#BUFFER_RESPONSE_ENTITY_ON_EXCEPTION]	jersey.config.client.bufferResponseEntityOnException	Buffer response entity in case of an exception. Default value is true. *NOT SUPPORTED.*
ClientProperties.CHUNKED_ENCODING_SIZE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html#CHUNKED_ENCODING_SIZE]	jersey.config.client.chunkedEncodingSize	Chunked encoding size. Default value is not set. *NOT SUPPORTED.*
ClientProperties.CONNECT_TIMEOUT [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html#CONNECT_TIMEOUT]	jersey.config.client.connectTimeout	Connect timeout interval, in milliseconds. Default value is 0 (infinity).
ClientProperties.FEATURE_AUTO_DISCOVERY_DISABLE [https://jersey.java.net/	jersey.config.disableAutoDiscovery	Disables auto discovery on client. Default value is false.

Constant	Value	Description
apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html#FEATURE_AUTO_DISCOVERY_DISABLE]		
ClientProperties.FOLLOW_REDIRECTS [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html#FOLLOW_REDIRECTS]	`jersey.config.client.followRedirects`	Declares if the client will automatically redirect to the URI declared in 3xx responses. Default value is `true`.
TTP_URL_CONNECTION_SET_METHOD_WORKAROUND ClientProperties [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html#HTTP_URL_CONNECTION_SET_METHOD_WORKAROUND]	`jersey.config.client .httpUrlConnection.setMethodWorkaround`	Declares configuration of workaround to set unsupported HTTP method to `HttpURLConnection` via reflection. Default value is `false`.
ClientProperties.JSON_PROCESSING_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html#JSON_PROCESSING_FEATURE_DISABLE]	`jersey.config.client.jsonProcessing.disable`	Disables configuration of Json Processing (JSR-353) feature. Default value is `false`.
ClientProperties.METAINF_SERVICES_LOOKUP_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html#METAINF_SERVICES_LOOKUP_DISABLE]	`jersey.config.client.disableMetainfServicesLookup`	Disables `META-INF/services` lookup on client. Default value is `false`.
ClientProperties.MOXY_JSON_FEATURE_DISABLE [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html#MOXY_JSON_FEATURE_DISABLE]	`jersey.config.client.disableMoxyJson`	Disables configuration of MOXy Json feature. Default value is `false`.
ClientProperties.OUTBOUND_CONTENT_LENGTH_BUFFER [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html#OUTBOUND_CONTENT_LENGTH_BUFFER] (Jersey 2.2 or later)	`jersey.config.client.contentLength.buffer`	An integer value that defines the buffer size used to buffer the outbound message entity in order to determine its size and set the value of HTTP `Content-Length` header. Default value is `8192`.
ClientProperties.READ_TIMEOUT [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html#READ_TIMEOUT]	`jersey.config.client.readTimeout`	Read timeout interval, in milliseconds. Default value is `0` (infinity).
ClientProperties.USE_ENCODING [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/ClientProperties.html#USE_ENCODING]	`jersey.config.client.useEncoding`	Enforces the value of `Content-Encoding` property the EncodingFilter [https://jersey.java.net/apidocs/2.1/jersey/org/glassfish/jersey/client/filter/EncodingFilter.html] should be adding. Default value is not set.