Jersey 1.12 User Guide

Table of Contents

List of Tables

List of Examples

Preface

The user guide is not complete (see the JavaDoc API for all details) and will be added to on a continual basis. If you want to contribute to the guide please email users@jersey.java.net.

Chapter 1. Getting Started

Table of Contents

This chapter will present how to get started with Jersey using the embedded Grizzly server. The last section of this chapter presents a reference to equivalent functionality for getting started with a Web application.

First, it is necessary to depend on the correct Jersey artifacts as described in Chapter 11, *Dependencies*

Maven developers require a dependency on

- the jersey-server module,
- the jersey-grizzly2 module,

The following dependencies need to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-server</artifactId>
    <version>1.12</version>
</dependency>
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-grizzly2</artifactId>
    <version>1.12</version>
</dependency>
```

If you want to depend on Jersey snapshot versions the following repository needs to be added to the 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>
```

Non-maven developers require:

- grizzly-http-server.jar,
- grizzly-http.jar,
- grizzly-framework.jar,
- jersey-server.jar,
- jersey-grizzly2.jar,
- jersey-core.jar,
- jsr311-api.jar,
- asm.jar

For Ant developers the Ant Tasks for Maven may be used to add the following to the ant document such that the dependencies do not need to be downloaded explicitly:

```
<artifact:dependencies pathId="dependency.classpath">
  <dependency groupId="com.sun.jersey"
              artifactId="jersey-server"
              version="1.12"/>
  <dependency groupId="com.sun.jersey"
              artifactId="jersey-core"
              version="1.12"/>
  <dependency groupId="com.sun.jersey"
              artifactId="jersey-grizzly2"
              version="1.12"/>
  <dependency groupId="org.glassfish.grizzly"
              artifactId="grizzly-http-server"
              version="2.2.1"/>
  <dependency groupId="org.glassfish.grizzly"
              artifactId="grizzly-http"
              version="2.2.1"/>
  <dependency groupId="org.glassfish.grizzly"
              artifactId="grizzly-framework"
              version="2.2.1"/>
  <dependency groupId="javax/ws/rs"
              artifactId="jsr311-api"
              version="1.1.1"/>
  <dependency groupId="asm"
```

```
            artifactId="asm"
            version="3.3.1"/>
  <artifact:remoteRepository id="maven-repository.java.net"
                             url="http://maven.java.net/" />
  <artifact:remoteRepository id="maven1-repository.java.net"
                             url="http://download.java.net/maven/1"
                             layout="legacy" />
</artifact:dependencies>
```

The path id “dependency.classpath” may then be referenced as the classpath to be used for compiling or executing.

Second, create a new project (using your favourite IDE or just ant/maven) and add the dependences. (For those who want to skip the creation of their own project take a look at Section 1.4, “Here's one Paul created earlier”

1.1. Creating a root resource

Create the following Java class in your project:

```
 1     // The Java class will be hosted at the URI path "/helloworld"
 2     @Path("/helloworld")
 3     public class HelloWorldResource {
 4
 5         // The Java method will process HTTP GET requests
 6         @GET
 7         // The Java method will produce content identified by the MIME Media
 8         // type "text/plain"
 9         @Produces("text/plain")
10         public String getClichedMessage() {
11             // Return some cliched textual content
12             return "Hello World";
13         }
14     }
```

The `HelloWorldResource` class is a very simple Web resource. The URI path of the resource is "/helloworld" (line 2), it supports the HTTP GET method (line 6) and produces cliched textual content (line 12) of the MIME media type "text/plain" (line 9).

Notice the use of Java annotations to declare the URI path, the HTTP method and the media type. This is a key feature of JSR 311.

1.2. Deploying the root resource

The root resource will be deployed using the Grizzly Web container.

Create the following Java class in your project:

```
 1 import com.sun.jersey.api.container.grizzly2.GrizzlyServerFactory;
 2 import com.sun.jersey.api.core.DefaultResourceConfig;
 3 import com.sun.jersey.api.core.PackagesResourceConfig;
 4 import com.sun.jersey.api.core.ResourceConfig;
 5 import org.glassfish.grizzly.http.server.HttpServer;
 6
 7 import javax.ws.rs.core.UriBuilder;
 8 import java.io.IOException;
 9 import java.net.URI;
10 import java.util.HashMap;
11 import java.util.Map;
12 import java.util.Map.Entry;
13
14 public class Main {
15
16     private static URI getBaseURI() {
17         return UriBuilder.fromUri("http://localhost/").port(9998).build();
18     }
19
20     public static final URI BASE_URI = getBaseURI();
21
22     protected static HttpServer startServer() throws IOException {
23         System.out.println("Starting grizzly...");
24         ResourceConfig rc = new PackagesResourceConfig("com.sun.jersey.samples.helloworld.resources");
25         return GrizzlyServerFactory.createHttpServer(BASE_URI, rc);
26     }
27
28     public static void main(String[] args) throws IOException {
29         HttpServer httpServer = startServer();
30         System.out.println(String.format("Jersey app started with WADL available at "
31                 + "%sapplication.wadl\nTry out %shelloworld\nHit enter to stop it...",
32                 BASE_URI, BASE_URI));
33         System.in.read();
34         httpServer.stop();
35     }
36 }
```

The `Main` class deploys the `HelloWorldResource` using the Grizzly Web container.

Line 24 creates an initialization parameter that informs the Jersey runtime where to search for root resource classes to be deployed. In this case it assumes the root resource class in the package `com.sun.jersey.samples.helloworld.resources` (or in a sub-package of).

Line 25 deploys the root resource to the base URI "http://localhost:9998/" and returns a Grizzly HttpServer. The complete URI of the Hello World root resource is "http://localhost:9998/helloworld".

Notice that no deployment descriptors were needed and the root resource was setup in a few statements of Java code.

1.3. Testing the root resource

Goto the URI http://localhost:9998/helloworld in your favourite browser.

Or, from the command line use `curl`:

```
> curl http://localhost:9998/helloworld
```

1.4. Here's one Paul created earlier

The example code presented above is shipped as the HelloWorld sample in the Java.Net maven repository.

For developers wishing to get started by deploying a Web application an equivalent sample, as a Web application, is shipped as the HelloWorld-WebApp sample.

Chapter 2. Overview of JAX-RS 1.1

Table of Contents

This chapter presents an overview of the JAX-RS 1.1 features.

The JAX-RS 1.1 API may be found online here.

The JAX-RS 1.1 specification draft may be found online here.

2.1. Root Resource Classes

Root resource classes are POJOs (Plain Old Java Objects) that are annotated with @Path have at least one method annotated with @Path or a resource method designator annotation such as @GET, @PUT, @POST, @DELETE. 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.

Example 2.1. Simple hello world root resource class

```
 1 package com.sun.ws.rest.samples.helloworld.resources;
 2
 3 import javax.ws.rs.GET;
 4 import javax.ws.rs.Produces;
 5 import javax.ws.rs.Path;
 6
 7 // The Java class will be hosted at the URI path "/helloworld"
 8 @Path("/helloworld")
 9 public class HelloWorldResource {
10
11     // The Java method will process HTTP GET requests
12     @GET
13     // The Java method will produce content identified by the MIME Media
14     // type "text/plain"
15     @Produces("text/plain")
16     public String getClichedMessage() {
17         // Return some cliched textual content
18         return "Hello World";
19     }
20 }
```

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

2.1.1. @Path

The @Path 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 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 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 may be used on method parameter of a request method, for example:

Example 2.2. 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         ...
```

```
    }
}
```

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 value may or may not begin with a '/', it makes no difference. Likewise, by default, a @Path 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. However, Jersey has a redirection mechanism, which if enabled, automatically performs redirection to a request URL ending in a '/' if a request URL does not end in a '/' and the matching @Path does end in a '/'.

2.1.2. HTTP Methods

@GET, @PUT, @POST, @DELETE and @HEAD 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 2.3. PUT method

```
@PUT
public Response putContainer() {
    System.out.println("PUT CONTAINER " + container);

    URI uri =  uriInfo.getAbsolutePath();
    Container c = new Container(container, uri.toString());

    Response r;
    if (!MemoryStore.MS.hasContainer(c)) {
        r = Response.created(uri).build();
    } else {
        r = Response.noContent().build();
    }

    MemoryStore.MS.createContainer(c);
    return r;
}
```

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). For OPTIONS the Allow response header will be set to the set of HTTP methods support by the resource. In addition Jersey will return a WADL document describing the resource.

2.1.3. @Produces

The @Produces 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 can be applied at both the class and method levels. Here's an example:

Example 2.4. Specifying output MIME type

```
@Path("/myResource")
@Produces("text/plain")
public class SomeResource {
    @GET
    public String doGetAsPlainText() {
        ...
    }

    @GET
    @Produces("text/html")
    public String doGetAsHtml() {
        ...
    }
}
```

The `doGetAsPlainText` method defaults to the MIME type of the @Produces annotation at the class level. The `doGetAsHtml` method's @Produces annotation overrides the class-level @Produces 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 declared 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 declaration, for example:

Example 2.5. Using multiple output MIME types

```
@GET
@Produces({"application/xml", "application/json"})
public String doGetAsXmlOrJson() {
    ...
}
```

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.

The examples above refer 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.

2.1.4. @Consumes

The @Consumes annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client. The above example can be modified to set the cliched message as follows:

Example 2.6. 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.

@Consumes can be applied at both the class and method levels and more than one media type may be declared in the same @Consumes declaration.

2.2. Deploying a RESTful Web Service

JAX-RS provides a deployment agnostic abstract class Application 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 2.7. Deployment agnostic application model

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

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 2.8. Reusing Jersey implementation in your custom application model

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

There are multiple deployment options for the class that implements Application interface in the Servlet 3.0 container. For simple deployments, no `web.xml` is needed at all. Instead, an @ApplicationPath annotation can be used to annotate the user defined application class and specify the the base resource URI of all application resources:

Example 2.9. Deployment of a JAX-RS application using `@ApplicationPath` with Servlet 3.0

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

You also need to set maven-war-plugin attribute failOnMissingWebXml to false in pom.xml when building .war without web.xml file using maven:

Example 2.10. Configuration of maven-war-plugin in `pom.xml` with Servlet 3.0

```
1  <plugins>
2   ...
3   <plugin>
4     <groupId>org.apache.maven.plugins</groupId>
5     <artifactId>maven-war-plugin</artifactId>
6     <version>2.1.1</version>
7     <configuration>
8         <failOnMissingWebXml>false</failOnMissingWebXml>
9     </configuration>
10    </plugin>
11  ...
12  </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 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 2.11. Deployment of a JAX-RS application using `web.xml` with Servlet 3.0

```
1 <web-app>
2     <servlet>
3         <servlet-name>org.foo.rest.MyApplication</servlet-name>
4     </servlet>
5     ...
6     <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 implementation class name as one of the servlet's `init-param` entries:

Example 2.12. Deployment of your application using Jersey specific servlet

```
<web-app>
    <servlet>
        <servlet-name>Jersey Web Application</servlet-name>
        <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>org.foo.rest.MyApplication</param-value>
        </init-param>
        ...
    </servlet>
    ...
</web-app>
```

Alternatively a simpler approach is to let Jersey choose the `PackagesResourceConfig` implementation automatically by declaring the packages as follows:

Example 2.13. Using Jersey specific servlet without an application model instance

```
<web-app>
    <servlet>
        <servlet-name>Jersey Web Application</servlet-name>
        <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>com.sun.jersey.config.property.packages</param-name>
            <param-value>org.foo.rest;org.bar.rest</param-value>
        </init-param>
        ...
    </servlet>
    ...
</web-app>
```

JAX-RS also provides the ability to obtain a container specific artifact from an Application instance. For example, Jersey supports using Grizzly as follows:

```
SelectorThread st = RuntimeDelegate.createEndpoint(new MyApplication(), SelectorThread.class);
```

Jersey also provides Grizzly helper classes to deploy the ServletThread instance at a base URL for in-process deployment.

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

2.3. Extracting Request Parameters

Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request. A previous example presented the use @PathParam to extract a path parameter from the path component of the request URL that matched the path declared in @Path.

@QueryParam 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 2.14. Query parameters

```
@Path("smooth")
@GET
public Response smooth(
        @DefaultValue("2") @QueryParam("step") int step,
        @DefaultValue("true") @QueryParam("min-m") boolean hasMin,
        @DefaultValue("true") @QueryParam("max-m") boolean hasMax,
        @DefaultValue("true") @QueryParam("last-m") boolean hasLast,
        @DefaultValue("blue") @QueryParam("min-color") ColorParam minColor,
        @DefaultValue("green") @QueryParam("max-color") ColorParam maxColor,
        @DefaultValue("red") @QueryParam("last-color") ColorParam lastColor
        ) { ... }
```

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 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 2.15. Custom Java type for consuming request parameters

```
public class ColorParam extends Color {
    public ColorParam(String s) {
        super(getRGB(s));
    }

    private static int getRGB(String s) {
        if (s.charAt(0) == '#') {
            try {
                Color c = Color.decode("0x" + s.substring(1));
                return c.getRGB();
            } catch (NumberFormatException e) {
                throw new WebApplicationException(400);
            }
        } else {
            try {
                Field f = Color.class.getField(s);
```

```
17                 return ((Color)f.get(null)).getRGB();
18             } catch (Exception e) {
19                 throw new WebApplicationException(400);
20             }
21         }
22     }
23 }
```

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)`); or
4. 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 4) may be used to obtain all values.

If the @DefaultValue is not used in conjunction with @QueryParam 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 and the other parameter-based annotations, @MatrixParam, @HeaderParam, @CookieParam, @FormParam obey the same rules as @QueryParam. @MatrixParam extracts information from URL path segments. @HeaderParam extracts information from the HTTP headers. @CookieParam extracts information from the cookies declared in cookie related HTTP headers.

@FormParam 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 2.16. 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 2.17. 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 2.18. 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 can be used to obtain contextual Java types related to the request or response. For form parameters it is possible to do the following:

Example 2.19. 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 }
```

2.4. Representations and Java Types

Previous sections on @Produces and @Consumes referred to MIME media types of representations and showed resource methods that consume and produce the Java type String for a number of different media types. However, `String` is just one of many Java types that are required to be supported by JAX-RS implementations.

Java types such as `byte[]`, `java.io.InputStream`, `java.io.Reader` and `java.io.File` are supported. In addition JAXB beans are supported. Such beans are `JAXBElement` or classes annotated with @XmlRootElement or @XmlType. The samples jaxb and json-from-jaxb show the use of JAXB beans.

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. 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 2.20. Using `File` with a specific MIME 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 }
```

A `File` type can also be used when consuming, a temporary file will be created where the request entity is stored.

The `Content-Type` (if not set, see next section) can be automatically set from the MIME media types declared by @Produces if the most acceptable media type is not a wild card (one that contains a *, for example "application/" or "/*"). Given the following method:

Example 2.21. The most acceptable MIME type is used when multiple output MIME types allowed

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

if "application/xml" is the most acceptable then the `Content-Type` of the response will be set to "application/xml".

2.5. 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 and Response.ResponseBuilder. 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 2.22. 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 2.23. 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(createdContent).build();
7 }
```

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

2.6. Sub-resources

@Path may be used on classes and such classes are referred to as root resource classes. @Path 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 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 2.24. 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 printer) { ... }
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 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 may be used is on methods **not** annotated with resource method designators such as @GET or @POST. 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 2.25. 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 public class ItemContentResource {
16
17     @GET
18     public Response get() { ... }
19
20     @PUT
21     @Path("{version}")
22     public void put(
23             @PathParam("version") int version,
24             @Context HttpHeaders headers,
25             byte[] in) { ... }
26 }
```

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.

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).

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.

2.7. Building URIs

A very important aspects 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 java.net.URI, which is why JAX-RS has the UriBuilder 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 provides such information (in addition to further information, see next section).

The following example shows URI building with UriInfo and UriBuilder from the bookmark sample:

Example 2.26. URI building

```
 1 @Path("/users/")
 2 public class UsersResource {
 3
 4     @Context UriInfo uriInfo;
 5
 6     ...
 7
 8     @GET
 9     @Produces("application/json")
10     public JSONArray getUsersAsJsonArray() {
11         JSONArray uriArray = new JSONArray();
12         for (UserEntity userEntity : getUsers()) {
13             UriBuilder ub = uriInfo.getAbsolutePathBuilder();
14             URI userUri = ub.
15                     path(userEntity.getUserid()).
16                     build();
17             uriArray.put(userUri.toASCIIString());
18         }
19         return uriArray;
20     }
21 }
```

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(). 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 2.27. Building URIs using query parameters

```
UriBuilder.fromUri("http://localhost/").
    path("{a}").
    queryParam("name", "{value}").
    build("segment", "value");
```

2.8. WebApplicationException and Mapping Exceptions to Responses

Previous sections have shown how to return HTTP responses and it is possible to return HTTP errors using the same mechanism. However, sometimes when programming in Java it is more natural to use exceptions for HTTP errors.

The following example shows the throwing of a `NotFoundException` from the bookmark sample:

Example 2.28. Throwing Jersey specific exceptions to control response

```
@Path("items/{itemid}/")
public Item getItem(@PathParam("itemid") String itemid) {
    Item i = getItems().get(itemid);
    if (i == null)
        throw new NotFoundException("Item, " + itemid + ", is not found");

    return i;
}
```

This exception is a Jersey specific exception that extends WebApplicationException and builds a HTTP response with the 404 status code and an optional message as the body of the response:

Example 2.29. Jersey specific exception implementation

```
public class NotFoundException extends WebApplicationException {

    /**
     * Create a HTTP 404 (Not Found) exception.
     */
    public NotFoundException() {
        super(Responses.notFound().build());
    }

    /**
     * Create a HTTP 404 (Not Found) exception.
     * @param message the String that is the entity of the 404 response.
     */
    public NotFoundException(String message) {
        super(Response.status(Responses.NOT_FOUND).
                entity(message).type("text/plain").build());
    }

}
```

In other cases it may not be appropriate to throw instances of WebApplicationException, or classes that extend WebApplicationException, and instead it may be preferable to map an existing exception to a response. For such cases it is possible to use the ExceptionMapper<E extends Throwable> interface. For example, the following maps the EntityNotFoundException to a HTTP 404 (Not Found) response:

Example 2.30. Mapping generic exceptions to responses

```
@Provider
public class EntityNotFoundMapper implements
        ExceptionMapper<javax.persistence.EntityNotFoundException> {
    public Response toResponse(javax.persistence.EntityNotFoundException ex) {
        return Response.status(404).
            entity(ex.getMessage()).
            type("text/plain").
            build();
    }
}
```

The above class is annotated with @Provider, 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 instance that is configured. When an application throws an EntityNotFoundException the `toResponse` method of the `EntityNotFoundMapper` instance will be invoked.

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

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

JAX-RS provides support for conditional GETs using the contextual interface Request.

The following example shows conditional GET support from the sparklines sample:

Example 2.31. Conditional GET support

```
public SparklinesResource(
        @QueryParam("d") IntegerList data,
        @DefaultValue("0,100") @QueryParam("limits") Interval limits,
        @Context Request request,
        @Context UriInfo ui) {
    if (data == null)
        throw new WebApplicationException(400);

    this.data = data;

    this.limits = limits;
```

```
12
13      if (!limits.contains(data))
14          throw new WebApplicationException(400);
15
16      this.tag = computeEntityTag(ui.getRequestUri());
17      if (request.getMethod().equals("GET")) {
18          Response.ResponseBuilder rb = request.evaluatePreconditions(tag);
19          if (rb != null)
20              throw new WebApplicationException(rb.build());
21      }
22  }
```

The constructor of the `SparklinesResouce` root resource class computes an entity tag from the request URI and then calls the request.evaluatePreconditions 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 returns a pre-filled out response, with the 304 status code and entity tag set, that may be built and returned. Otherwise, evaluatePreconditions returns `null` and the normal response can be returned.

Notice that in this example the constructor of a resource class can be used perform actions that may otherwise have to be duplicated to invoked for each resource method.

2.10. Life-cycle of Root Resource Classes

By default the life-cycle of root resource classes is per-request, 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.

Jersey supports two further life-cycles using Jersey specific annotations. If a root resource class is annotated with @Singleton then only one instance is created per-web application. If a root resource class is annotated with @PerSession then one instance is created per web session and stored as a session attribute.

2.11. Security

Security information is available by obtaining the SecurityContext using @Context, which is essentially the equivalent functionality available on the HttpServletRequest.

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 2.32. Accessing `SecurityContext`

```
1 @Path("basket")
2 public ShoppingBasketResource get(@Context SecurityContext sc) {
3     if (sc.isUserInRole("PreferredCustomer") {
4        return new PreferredCustomerShoppingBaskestResource();
5     } else {
6        return new ShoppingBasketResource();
7     }
8 }
```

2.12. 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 2.33. Injection

```
1 @Path("id: \d+")
2 public class InjectedResource {
3     // Injection onto field
4     @DefaultValue("q") @QueryParam("p")
5     private String p;
6
7     // Injection onto constructor parameter
8     public InjectedResource(@PathParam("id") int id) { ... }
9
10    // Injection onto resource method parameter
11    @GET
12    public String get(@Context UriInfo ui) { ... }
13
14    // Injection onto sub-resource resource method parameter
15    @Path("sub-id")
16    @GET
17    public String get(@PathParam("sub-id") String id) { ... }
18
19    // Injection onto sub-resource locator method parameter
20    @Path("sub-id")
21    public SubResource getSubResource(@PathParam("sub-id") String id) { ... }
22
23    // Injection using bean setter method
24    @HeaderParam("X-header")
25    public void setHeader(String header) { ... }
26 }
```

There are some restrictions when injecting on to resource classes with a life-cycle other than per-request. In such cases it is not possible to injected onto fields for the annotations associated with extraction of request parameters. However, it is possible to use the @Context annotation on fields, in such cases a thread local proxy will be injected.

The @FormParam annotation is special and may only be utilized on resource and sub-resource methods. This is because it extracts information from a request entity.

2.13. Use of @Context

Previous sections have introduced the use of @Context. Chapter 5 of the JAX-RS specification presents all the standard JAX-RS Java types that may be used with @Context.

When deploying a JAX-RS application using servlet then ServletConfig, ServletContext, HttpServletRequest and HttpServletResponse are available using @Context.

2.14. Annotations Defined By JAX-RS

For a list of the annotations specified by JAX-RS see Appendix A of the specification.

Chapter 3. Client API

Table of Contents

3.1. Introduction

This section introduces the client API and some features such as filters and how to use them with security features in the JDK. The Jersey client API is a high-level Java based API for interoperating with RESTful Web services. It makes it very easy to interoperate with RESTful Web services and enables a developer to concisely and efficiently implement a reusable client-side solution that leverages existing and well established client-side HTTP implementations.

The Jersey client API can be utilized to interoperate with any RESTful Web service, implemented using one of many frameworks, and is not restricted to services implemented using JAX-RS. However, developers familiar with JAX-RS should find the Jersey client API complementary to their services, especially if the client API is utilized by those services themselves, or to test those services.

The goals of the Jersey 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 interoperate with RESTful Web services as JAX-RS makes it easy to build RESTful Web services; and
3. Leverage artifacts of the JAX-RS API for the client side. Note that JAX-RS is currently a server-side only API.

The Jersey Client API supports a pluggable architecture to enable the use of different underlying HTTP client implementations. Two such implementations are supported and leveraged: the `Http(s)URLConnection` classes supplied with the JDK; and the Apache HTTP client.

3.2. 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. modifiability 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 retured 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 `java.net.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 Jersey client API is an instance of the Java class WebResource, and encapsulates a URI. The fixed set of HTTP methods are methods on `WebResource` or if using the builder pattern (more on this later) are the last methods to be called when invoking an HTTP method on a resource. The representations are Java types, instances of which, may contain links that new instances of `WebResource` may be created from.

3.3. Ease of use and reusing JAX-RS artifacts

Since a resource is represented as a Java type it makes it easy to configure, pass around and inject in ways that is 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 and UriTemplate to safely build URIs;
2. Support for Java types of representations such as `byte[]`, `String`, `InputStream`, `File`, `DataSource` and JAXB beans in addition to Jersey specific features such as JSON support and MIME Multipart support.
3. Using the builder pattern to make it easier to construct requests.

Some APIs, like the Apache HTTP client or java.net.HttpURLConnection, can be rather hard to use and/or require too much code to do something relatively simple.

This is why the Jersey Client API provides support for wrapping HttpURLConnection and the Apache HTTP client. Thus it is possible to get the benefits of the established implementations and features while getting the ease of use benefit.

It is not intuitive to send a POST request with form parameters and receive a response as a JAXB object with such an API. For example with the Jersey API this is very easy:

Example 3.1. POST request with form parameters

```
Form f = new Form();
f.add("x", "foo");
f.add("y", "bar");

Client c = Client.create();
WebResource r = c.resource("http://localhost:8080/form");

JAXBBean bean = r.
    type(MediaType.APPLICATION_FORM_URLENCODED_TYPE)
    .accept(MediaType.APPLICATION_JSON_TYPE)
    .post(JAXBBean.class, f);
```

In the above code a Form is created with two parameters, a new WebResource instance is created from a Client then the `Form` instance is `POST`ed to the resource, identified with the form media type, and the response is requested as an instance of a JAXB bean with an acceptable media type identifying the Java Script Object Notation (JSON) format. The Jersey client API manages the serialization of the `Form` instance to produce the request and de-serialization of the response to consume as an instance of a JAXB bean.

If the code above was written using `HttpURLConnection` then the developer would have to write code to serialize the form sent in the POST request and de-serialize the response to the JAXB bean. In addition further code would have to be written to make it easy to reuse the same resource “http://localhost:8080/form” that is encapsulated in the `WebResource` type.

3.4. Getting started with the Jersey client

Refer to the dependencies chapter, and specifically the Core client section, for details on the dependencies when using the Jersey client with Maven and Ant.

Refer to the Java API documentation for details on the Jersey client API packages and classes.

Refer to the Java API Apache HTTP client documentation for details on how to use the Jersey client API with the Apache HTTP client.

3.5. Overview of the API

To utilize the client API it is first necessary to create an instance of a Client, for example:

```
Client c = Client.create();
```

3.5.1. Configuring a Client and WebResource

The client instance can then be configured by setting properties on the map returned from the `getProperties` methods or by calling the specific setter methods, for example the following configures the client to perform automatic redirection for appropriate responses:

```
c.getProperties().put(
    ClientConfig.PROPERTY_FOLLOW_REDIRECTS, true);
```

which is equivalent to the following:

```
c.setFollowRedirects(true);
```

Alternatively it is possible to create a `Client` instance using a ClientConfig object for example:

```
ClientConfig cc = new DefaultClientConfig();
cc.getProperties().put(
    ClientConfig.PROPERTY_FOLLOW_REDIRECTS, true);
Client c = Client.create(cc);
```

Once a client instance is created and configured it is then possible to obtain a WebResource instance, which will inherit the configuration declared on the client instance. For example, the following creates a reference to a Web resource with the URI “http://localhost:8080/xyz”:

```
WebResource r = c.resource("http://localhost:8080/xyz");
```

and redirection will be configured for responses to requests invoked on the Web resource.

`Client` instances are expensive resources. It is recommended a configured instance is reused for the creation of Web resources. The creation of Web resources, the building of requests and receiving of responses are guaranteed to be thread safe. Thus a `Client` instance and `WebResource` instances may be shared between multiple threads.

In the above cases a `WebResource` instance will utilize `HttpUrlConnection` or `HttpsUrlConnection`, if the URI scheme of the `WebResource` is “http” or “https” respectively.

3.5.2. Building a request

Requests to a Web resource are built using the builder pattern (see RequestBuilder) where the terminating method corresponds to an HTTP method (see UniformInterface). For example,

```
String response = r.accept(
    MediaType.APPLICATION_JSON_TYPE,
    MediaType.APPLICATION_XML_TYPE).
    header("X-FOO", "BAR").
    get(String.class);
```

The above sends a `GET` request with an `Accept` header of `application/json`, `application/xml` and a non-standard header `X-FOO` of `BAR`.

If the request has a request entity (or representation) then an instance of a Java type can be declared in the terminating HTTP method, for `PUT`, `POST` and `DELETE` requests. For example, the following sends a POST request:

```
String request = "content";
String response = r.accept(
    MediaType.APPLICATION_JSON_TYPE,
    MediaType.APPLICATION_XML_TYPE).
    header("X-FOO", "BAR").
    post(String.class, request);
```

where the String "content" will be serialized as the request entity (see the section "Java instances and types for representations" section for further details on the supported Java types). The `Content-Type` of the request entity may be declared using the `type` builder method as follows:

```
String response = r.accept(
     MediaType.APPLICATION_JSON_TYPE,
     MediaType.APPLICATION_XML_TYPE).
     header("X-FOO", "BAR").
     type(MediaType.TEXT_PLAIN_TYPE).
     post(String.class, request);
```

or alternatively the request entity and type may be declared using the entity method as follows:

```
String response = r.accept(
     MediaType.APPLICATION_JSON_TYPE,
     MediaType.APPLICATION_XML_TYPE).
     header("X-FOO", "BAR").
     entity(request, MediaType.TEXT_PLAIN_TYPE).
     post(String.class);
```

3.5.3. Receiving a response

If the response has a entity (or representation) then the Java type of the instance required is declared in the terminating HTTP method. In the above examples a response entity is expected and an instance of `String` is requested. The response entity will be de-serialized to a String instance.

If response meta-data is required then the Java type ClientResponse can be declared from which the response status, headers and entity may be obtained. For example, the following gets both the entity tag and response entity from the response:

```
ClientResponse response = r.get(ClientResponse.class);
EntityTag e = response.getEntityTag();
String entity = response.getEntity(String.class);
```

If the `ClientResponse` type is not utilized and the response status is greater than or equal to 300 then the runtime exception UniformInterfaceException is thrown. This exception may be caught and the `ClientResponse` obtained as follows:

```
try {
    String entity = r.get(String.class);
} catch (UniformInterfaceException ue) {
    ClientResponse response = ue.getResponse();
}
```

3.5.4. Creating new WebResources from a WebResource

A new WebResource can be created from an existing `WebResource` by building from the latter's URI. Thus it is possible to build the request URI before building the request. For example, the following appends a new path segment and adds some query parameters:

```
WebResource r = c.resource("http://localhost:8080/xyz");

MultivaluedMap<String, String> params = MultivaluedMapImpl();
params.add("foo", "x");
params.add("bar", "y");

String response = r.path("abc").
    queryParams(params).
    get(String.class);
```

that results in a GET request to the URI "http://localhost:8080/xyz/abc?foo=x&bar=y".

3.5.5. Java instances and types for representations

All the Java types for representations supported by the Jersey server side for requests and responses are also supported on the client side. This includes the standard Java types as specified by JAX-RS in section 4.2.4 in addition to JSON, Atom and Multipart MIME as supported by Jersey.

To process a response entity (or representation) as a stream of bytes use InputStream as follows:

```
InputStream in = r.get(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 `File` as follows:

```
File f = ...
String response = r.post(String.class, f);
```

Refer to the JAXB sample to see how JAXB with XML and JSON can be utilized with the client API (more specifically, see the unit tests).

3.6. Adding support for new representations

The support for new application-defined representations as Java types requires the implementation of the same provider-based interfaces as for the server side JAX-RS API, namely MessageBodyReader and MessageBodyWriter, respectively, for request and response entities (or inbound and outbound representations). Refer to the entity provider sample for such implementations utilized on the server side.

Classes or implementations of the provider-based interfaces need to be registered with a `ClientConfig` and passed to the `Client` for creation. The following registers a provider class `MyReader` which will be instantiated by Jersey:

```
ClientConfig cc = new DefaultClientConfig();
cc.getClasses().add(MyReader.class);
Client c = Client.create(cc);
```

The following registers an instance or singleton of MyReader:

```
ClientConfig cc = new DefaultClientConfig();
MyReader reader = ...
cc.getSingletons().add(reader);
Client c = Client.create(cc);
```

3.7. Using filters

Filtering requests and responses can provide useful functionality that is hidden 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.

The `Client` and `WebResource` classes extend from Filterable and that enables the addition of ClientFilter instances. A `WebResource` will inherit filters from its creator, which can be a `Client` or another `WebResource`. Additional filters can be added to a `WebResource` after it has been created. For requests, filters are applied in reverse order, starting with the `WebResource` filters and then moving to the inherited filters. For responses, filters are applied in order, starting with inherited filters and followed by the filters added to the `WebResource`. All filters are applied in the order in which they were added. For instance, in the following example the `Client` has two filters added, `filter1` and `filter2`, in that order, and the `WebResource` has one filter added, `filter3`:

```
ClientFilter filter1 = ...
ClientFilter filter2 = ...
Client c = Client.create();
c.addFilter(filter1);
c.addFilter(filter2);

ClientFilter filter3 = ...
WebResource r = c.resource(...);
r.addFilter(filter3);
```

After a request has been built the request is filtered by `filter3`, `filter2` and `filter1` in that order. After the response has been received the response is filtered by `filter1`, `filter2` and `filter3` in that order, before the response is returned.

Filters are implemented using the "russian doll" stack-based pattern where a filter is responsible for calling the next filter in the ordered list of filters (or the next filter in the "chain" of filters). The basic template for a filter is as follows:

```
class AppClientFilter extends ClientFilter {
    public ClientResponse handle(ClientRequest cr) {
        // Modify the request
        ClientRequest mcr = modifyRequest(cr);
        // Call the next filter
        ClientResponse resp = getNext().handle(mcr);
        // Modify the response
        return modifyResponse(resp);
    }
}
```

The filter modifies the request (if required) by creating a new ClientRequest or modifying the state of the passed `ClientRequest` before calling the next filter. The call to the next request will return the response, a `ClientResponse`. The filter modifies the response (if required) by creating a new `ClientResponse` or modifying the state of the returned `ClientResponse`. Then the filter returns the modified response. Filters are re-entrant and may be called by multiple threads performing requests and processing responses.

3.7.1. Supported filters

The Jersey Client API currently supports two filters:

1. A GZIP content encoding filter, GZIPContentEncodingFilter. If this filter is added then a request entity is compressed with the `Content-Encoding` of `gzip`, and a response entity if compressed with a `Content-Encoding` of `gzip` is decompressed. The filter declares an `Accept-Encoding` of `gzip`.

2. A logging filter, LoggingFilter. If this filter is added then the request and response headers as well as the entities are logged to a declared output stream if present, or to `System.out` if not. Often this filter will be placed at the end of the ordered list of filters to log the request before it is sent and the response after it is received.

The filters above are good examples that show how to modify or read request and response entities. Refer to the source code of the Jersey client for more details.

3.8. Testing services

The Jersey client API was originally developed to aid the testing of the Jersey server-side, primarily to make it easier to write functional tests in conjunction with the JUnit framework for execution and reporting. It is used extensively and there are currently over 1000 tests.

Embedded servers, Grizzly and a special in-memory server, are utilized to deploy the test-based services. Many of the Jersey samples contain tests that utilize the client API to server both for testing and examples of how to use the API. The samples utilize Grizzly or embedded Glassfish to deploy the services.

The following code snippets are presented from the single unit test `HelloWorldWebAppTest` of the helloworld-webapp sample. The `setUp` method, called before a test is executed, creates an instance of the Glassfish server, deploys the application, and a `WebResource` instance that references the base resource:

```
@Override
protected void setUp() throws Exception {
    super.setUp();

    // Start Glassfish
    glassfish = new GlassFish(BASE_URI.getPort());

    // Deploy Glassfish referencing the web.xml
    ScatteredWar war = new ScatteredWar(
        BASE_URI.getRawPath(), new File("src/main/webapp"),
        new File("src/main/webapp/WEB-INF/web.xml"),
        Collections.singleton(
            new File("target/classes").
                toURI().toURL()));
    glassfish.deploy(war);

    Client c = Client.create();
    r = c.resource(BASE_URI);
}
```

The `tearDown` method, called after a test is executed, stops the Glassfish server.

```
@Override
protected void tearDown() throws Exception {
    super.tearDown();
    glassfish.stop();
}
```

The `testHelloWorld` method tests that the response to a `GET` request to the Web resource returns "Hello World":

```
public void testHelloWorld() throws Exception {
   String responseMsg = r.path("helloworld").
       get(String.class);
   assertEquals("Hello World", responseMsg);
}
```

Note the use of the `path` method on the `WebResource` to build from the base `WebResource`.

3.9. Security with Http(s)URLConnection

3.9.1. With Http(s)URLConnection

The support for security, specifically HTTP authentication and/or cookie management with `Http(s)URLConnection` is limited due to constraints in the API. There are currently no specific features or properties on the `Client` class that can be set to support HTTP authentication. However, since the client API, by default, utilizes `HttpURLConnection` or `HttpsURLConnection`, it is possible to configure system-wide security settings (which is obviously not sufficient for multiple client configurations).

For HTTP authentication the `java.net.Authenticator` can be extended and statically registered. Refer to the Http authentication document for more details. For cookie management the `java.net.CookieHandler` can be extended and statically registered. Refer to the Cookie Management document for more details.

To utilize HTTP with SSL it is necessary to utilize the "https" scheme. For certificate-based authentication see the class HTTPSProperties for how to set `javax.net.ssl.HostnameVerifier` and `javax.net.ssl.SSLContext`.

3.9.2. With Apache HTTP client

The support for HTTP authentication and cookies is much better with the Apache HTTP client than with `HttpURLConnection`. See the Java documentation for the package com.sun.jersey.client.apache, ApacheHttpClientState and ApacheHttpClientConfig for more details.

Chapter 4. XML Support

Table of Contents

As you probably already know, Jersey uses MessageBodyWriters and MessageBodyReaders to parse incoming request 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.

4.1. Low level XML support

Jersey currently support several low level data types: `StreamSource`, `SAXSource`, `DOMSource` and `Document`. You can use these types as return type or method (resource) parameter. Lets say we want to test this feature and we have helloworld sample as starting point. All we need to do is add methods (resources) which consumes and produces XML and types mentioned above will be used.

Example 4.1. Low level XML test - methods added to HelloWorldResource.java

```
1     @Path("1")
2     @POST
3     public StreamSource get1(StreamSource streamSource) {
4         return streamSource;
5     }
6
7     @Path("2")
8     @POST
9     public SAXSource get2(SAXSource saxSource) {
10        return saxSource;
11    }
12
13    @Path("3")
14    @POST
15    public DOMSource get3(DOMSource domSource) {
16        return domSource;
17    }
18
19    @Path("4")
20    @POST
21    public Document get4(Document document) {
22        return document;
23    }
```

Both MessageBodyReaders and MessageBodyWriters are used in this case, all we need is do POST request with some XML document as a request entity. I want to keep this as simple as possible so I'm going to send only root element with no content: "`<test />`". You can create Jersey client to do that or use some other tool, for example `curl` as I did. (`curl -v http://localhost:9998/helloworld/1 -d "<test />"`). You should get exactly 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.

4.2. Getting started with JAXB

Good start for people which already have some experience with JAXB annotations is JAXB sample. You can see various usecases 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) 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 right chapter.

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

Example 4.2. Planet class

```
1 @XmlRootElement
2 public class Planet {
3     public int id;
4     public String name;
5     public double radius;
6 }
7
```

Example 4.3. Resource class

```
1 @Path("planet")
2 public class Resource {
3
4     @GET
5     @Produces(MediaType.APPLICATION_XML)
6     public Planet getPlanet() {
```

```
        Planet p = new Planet();
        p.id = 1;
        p.name = "Earth";
        p.radius = 1.0;

        return p;
    }
}
```

You can see there is some extra annotation declared on `Planet` class. Concretely `XmlRootelement`. What it does? This is a JAXB annotation which maps java class to XML element. We don't need specify anything else, because `Planet` is very simple class and all fields are public. In this case, XML element name will be derived from class name or you can set 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 Jersey client for making requests to this resource and this is easy as: `Planet planet = webResource.path("planet").accept(MediaType.APPLICATION_XML_TYPE).get(Planet.class);`. There is pre-created WebResource object which points to our applications context root and we simpli add path (in our clase 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 4.4. Method for consuming Planet

```
    @POST
    @Consumes(MediaType.APPLICATION_XML)
    public void setPlanet(Planet p) {
        System.out.println("setPlanet " + p);
    }

```

After valid request is made (with Jersey client you can do `webResource.path("planet").post(p);`), service will print out string representation of Planet, which can look like `Planet{id=2, name='Mars', radius=1.51}`.

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. Xjc is XML to java compiler and is part of JAXB. See JAXB home page for further details.

4.3. 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` class should help you. Let's redo planet resource but this time we won't have `XmlRootElement` annotation on `Planet` class.

Example 4.5. Resource class - JAXBElement

```
@Path("planet")
public class Resource {

    @GET
    @Produces(MediaType.APPLICATION_XML)
    public JAXBElement<Planet> getPlanet() {
        Planet p = new Planet();
        p.id = 1;
        p.name = "Earth";
        p.radius = 1.0;

        return new JAXBElement<Planet>(new QName("planet"), Planet.class, p);
    }

    @POST
    @Consumes(MediaType.APPLICATION_XML)
    public void setPlanet(JAXBElement<Planet> p) {
        System.out.println("setPlanet " + p.getValue());
    }
}
```

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

Example 4.6. Client side - JAXBElement

```
        // GET
        GenericType<JAXBElement<Planet>> planetType = new GenericType<JAXBElement<Planet>>() {};

        Planet planet = (Planet) webResource.path("planet").accept(MediaType.APPLICATION_XML_TYPE).get(planetType).getValue();
        System.out.println("### " + planet);

        // POST
        Planet p = new Planet();
        // ...

        webResource.path("planet").post(new JAXBElement<Planet>(new QName("planet"), Planet.class, p));
```

4.4. Using custom JAXBContext

In some scenarios you can take advantage of using custom `JAXBContext`. Creating `JAXBContext` is expensive operation and if you already have one created, same instance can be used by Jersey.

Other possible usecase for this is when you need to set some specific things to JAXBContext, for example set different classloader.

Example 4.7. 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 that this
                // 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 for reminder - you have to declare them in client config (cliend does not anything like package scanning done by server).

Example 4.8. Using Provider with Jersey client

```
                ClientConfig cc = new DefaultClientConfig();
                cc.getClasses().add(PlanetJAXBContextProvider.class);
                Client c = Client.create(cc);

```

Chapter 5. JSON Support

Table of Contents

Jersey JSON support comes as a set of JAX-RS MessageBodyReader<T> and MessageBodyWriter<T> providers distributed with *jersey-json* module. These providers enable using three basic approaches when working with JSON format:

- POJO support
- JAXB based JSON support
- Low-level, JSONObject/JSONArray based JSON 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 could be taken if you want to utilize certain JAXB features. The last, low-level, approach gives you the best fine-grained control over the outcoming JSON data format.

5.1. POJO support

POJO suppport represents the easiest way to convert your Java Objects to JSON and back. It is based on the Jackson library.

To use this approach, you will need to turn the *JSONConfiguration.FEATURE_POJO_MAPPING* feature on. This could be done in `web.xml` using the following servlet init parameter:

Example 5.1. POJO JSON support web.xml configuration

```
        <init-param>
            <param-name>com.sun.jersey.api.json.POJOMappingFeature</param-name>
            <param-value>true</param-value>
        </init-param>

```

The following snippet shows how to use the POJO mapping feature on the client side:

Example 5.2. POJO JSON support client configuration

```
  ClientConfig clientConfig = new DefaultClientConfig();
  clientConfig.getFeatures().put(JSONConfiguration.FEATURE_POJO_MAPPING, Boolean.TRUE);
  Client client = Client.create(clientConfig);

```

Jackson JSON processor could be futher controlled via providing custom Jackson ObjectMapper 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.

Download https://maven.java.net/service/local/artifact/maven/redirect?r=releases&g=com.sun.jersey.samples&a=jacksonjsonprovider&v=1.12&c=project&e=zip to get a complete example using POJO based JSON support.

5.2. 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. Because even then you will still be able to use a unified Java model. Another advantage is simplicity of working with such a model, as 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 could 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 things get serialized out and deserialized back.

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

Example 5.3. Simple JAXB bean implementation

```
@XmlRootElement
public class MyJaxbBean {
  public String name;
  public int age;

  public MyJaxbBean() {} // JAXB needs this

  public MyJaxbBean(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

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

Example 5.4. JAXB bean used to generate JSON representation

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

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"}
```

5.2.1. Configuration Options

JAXB itself enables you to control output JSON format to certain extent. Specifically renaming and ommiting items is easy to do directly using JAXB annotations. E.g. the following example depicts changes in the above mentioned MyJaxbBean that will result in `{"king":"Agamemnon"}` JSON output.

Example 5.5. Tweaking JSON format using JAXB

```
@XmlRootElement
public class MyJaxbBean {

    @XmlElement(name="king")
    public String name;

    @XmlTransient
    public int age;

    // several lines removed
}
```

To achieve more important JSON format changes, you will need to configure Jersey JSON procesor itself. Various configuration options could be set on an JSONConfiguration instance. The instance could be then further used to create a JSONConfigurated JSONJAXBContext, which serves as a main configuration point in this area. To pass your specialized JSONJAXBContext to Jersey, you will finally need to implement a JAXBContext ContextResolver<T>:

Example 5.6. An example of a JAXBContext resolver implementation

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

    private JAXBContext context;
    private Class[] types = {MyJaxbBean.class};

    public JAXBContextResolver() throws Exception {
        this.context =
      new JSONJAXBContext( (1)
        JSONConfiguration.natural().build(), types); (2)
    }

    public JAXBContext getContext(Class<?> objectType) {
        for (Class type : types) {
            if (type == objectType) {
                return context;
            }
        }
        return null;
    }
}
```

(1) Creation of our specialized JAXBContext
(2) Final JSON format is given by this JSONConfiguration instance

5.2.2. JSON Notations

JSONConfiguration allows you to use four various JSON notations. Each of these notations serializes JSON in a different way. Following is a list of supported notations:

- MAPPED (default notation)
- NATURAL

- JETTISON_MAPPED
- BADGERFISH

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

```
@XmlRootElement
public class Address {
    public String street;
    public String town;

    public Address(){}

    public Address(String street, String town) {
        this.street = street;
        this.town = town;
    }
}
```

Example 5.8. JAXB beans for JSON supported notations description, contact bean

```
@XmlRootElement
public class Contact {

    public int id;
    public String name;
    public List<Address> addresses;

    public Contact() {};

    public Contact(int id, String name, List<Address> addresses) {
        this.name = name;
        this.id = id;
        this.addresses =
            (addresses != null) ? new LinkedList<Address>(addresses) : null;
    }
}
```

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

Example 5.9. JAXB beans for JSON supported notations description, initialization

```
final Address[] addresses = {new Address("Long Street 1", "Short Village")};
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 apidocs at http://jersey.java.net/nonav/apidocs/1.12/jersey/com/sun/jersey/api/json/JSONConfiguration.html

5.2.2.1. Mapped notation

`JSONConfiguration` based on `mapped` notation could be build with

```
JSONConfiguration.mapped().build()
```

for usage in a `JAXBContext` resolver, Example 5.6, “An example of a JAXBContext resolver implementation”. Then a contact bean initialized with Example 5.9, “JAXB beans for JSON supported notations description, initialization”, will be serialized as

Example 5.10. JSON expression produced using `mapped` notation

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

The JSON representation seems fine, and will be working flawlessly with Java based Jersey client API.

However, at least one issue might appear once you start using it with a JavaScript based client. The information, that `addresses` item represents an array, is being lost for every single element array. If you added another address bean to the contact,

```
contact.addresses.add(new Address("Short Street 1000", "Long Village"));
```

, you would get

```
{ "id":"2"
 ,"name":"Bob"
 ,"addresses":[{"street":"Long Street 1","town":"Short Village"}
              ,{"street":"Short Street 1000","town":"Long Village"}]}
```

Both representations are correct, but you will not be able to consume them using a single JavaScript client, because to access `"Short Village"` value, you will write `addresses.town` in one case and `addresses[0].town` in the other. To fix this issue, you need to instruct the JSON processor, what items need to be treated as arrays by setting an optional property, `arrays`, on your `JSONConfiguration` object. For our case, you would do it with

Example 5.11. Force arrays in `mapped` JSON notation

```
JSONConfiguration.mapped().arrays("addresses").build()
```

You can use multiple string values in the `arrays` method call, in case you are dealing with more than one array item in your beans. Similar mechanism (one or more argument values) applies also for all below desribed options.

Another issue might be, that number value, `2`, for `id` item gets written as a string, `"2"`. To avoid this, you can use another optional property on `JSONConfiguration` called `nonStrings`.

Example 5.12. Force non-string values in mapped JSON notation

```
JSONConfiguration.mapped().arrays("addresses").nonStrings("id").build()
```

It might happen you use XML attributes in your JAXB beans. In mapped JSON notation, these attribute names are prefixed with @ character. If id was an attribute, it´s definition would look like:

```
...
@XmlAttribute
public int id;
...
```

and then you would get

```
{"@id":"2" ...
```

at the JSON output. In case, you want to get rid of the @ prefix, you can take advantage of another configuration option of JSONConfiguration, called attributeAsElement. Usage is similar to previous options.

Example 5.13. XML attributes as XML elements in mapped JSON notation

```
JSONConfiguration.mapped().attributeAsElement("id").build()
```

Mapped JSON notation was designed to produce the simplest possible JSON expression out of JAXB beans. While in XML, you must always have a root tag to start a XML document with, there is no such a constraint in JSON. If you wanted to be strict, you might have wanted to keep a XML root tag equivalent generated in your JSON. If that is the case, another configuration option is available for you, which is called rootUnwrapping. You can use it as follows:

Example 5.14. Keep XML root tag equivalent in JSON mapped JSON notation

```
JSONConfiguration.mapped().rootUnwrapping(false).build()
```

and get the following JSON for our Contact bean:

Example 5.15. XML root tag equivalent kept in JSON using mapped notation

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

rootUnwrapping option is set to true by default. You should switch it to false if you use inheritance at your JAXB beans. Then JAXB might try to encode type information into root element names, and by stripping these elements off, you could break unmarshalling.

In version 1.1.1-ea, XML namespace support was added to the MAPPED JSON notation. There is of course no such thing as XML namespaces in JSON, but when working from JAXB, XML infoset is used as an intermediary format. And then when various XML namespaces are used, ceratin information related to the concrete namespaces is needed even in JSON data, so that the JSON procesor could correctly unmarshal JSON to XML and JAXB. To make it short, the XML namespace support means, you should be able to use the very same JAXB beans for XML and JSON even if XML namespaces are involved.

Namespace mapping definition is similar to Example 5.20, “XML namespace to JSON mapping configuration for Jettison based mapped notation”

Example 5.16. XML namespace to JSON mapping configuration for mapped notation

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

Dot character (.) will be used by default as a namespace separator in the JSON identifiers. E.g. for the above mentioned example namespace and tag T, "example.T" JSON identifier will be generated. To change this default behaviour, you can use the nsSeparator method on the mapped JSONConfiguration builder: JSONConfiguration.mapped().xml2JsonNs(ns2json).nsSeparator(':').build(). Then you will get "example:T" instead of "example.T" generated. This option should be used carefully, as the Jersey framework does not even try to check conflicts between the user selected separator character and the tag and/or namespace names.

5.2.2.2. Natural notation

After using mapped JSON notation for a while, it was apparent, that a need to configure all the various things manually could be a bit problematic. To avoid the manual work, a new, natural, JSON notation was introduced in Jersey version 1.0.2. With natural notation, Jersey will automatically figure out how individual items need to be processed, so that you do not need to do any kind of manual configuration. Java arrays and lists are mapped into JSON arrays, even for single-element cases. Java numbers and booleans are correctly mapped into JSON numbers and booleans, and you do not need to bother with XML attributes, as in JSON, they keep the original names. So without any additional configuration, just using

```
JSONConfiguration.natural().build()
```

for configuring your JAXBContext, you will get the following JSON for the bean initialized at Example 5.9, “JAXB beans for JSON supported notations description, initialization”:

Example 5.17. JSON expression produced using natural notation

```
1 { "id":2
2  ,"name":"Bob"
3  ,"addresses":[{"street":"Long Street 1"
4                     ,"town":"Short Village"}]}
```

You might notice, that the single element array addresses remains an array, and also the non-string id value is not limited with double quotes, as natural notation automatically detects these things.

To support cases, when you use inheritance for your JAXB beans, an option was introduced to the natural JSON configuration builder to forbid XML root element stripping. The option looks pretty same as at the default mapped notation case (Example 5.14, “Keep XML root tag equivalent in JSON mapped JSON notation”).

Example 5.18. Keep XML root tag equivalent in JSON natural JSON notation

```
JSONConfiguration.natural().rootUnwrapping(false).build()
```

5.2.2.3. Jettison mapped notation

Next two notations are based on project Jettison. 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.

Jettison based `mapped` notation could be configured using:

```
JSONConfiguration.mappedJettison().build()
```

If nothing else is configured, you will get similar JSON output as for the default, `mapped`, notation:

Example 5.19. JSON expression produced using Jettison based `mapped` notation

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

The only difference is, your numbers and booleans will not be converted into strings, but you have no option for forcing arrays remain arrays in single-element case. Also the JSON object, representing XML root tag is being produced.

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

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

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

Example 5.20. 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 JSONJAXBContext(
4           JSONConfiguration.mappedJettison()
5                .xml2JsonNs(ns2json).build(), types);
```

Resulting JSON will look like in the example bellow.

Example 5.21. JSON expression with XML namespaces mapped into JSON

```
1 { "contact:{"example.id":2
2               ,"name":"Bob"
3               ,"addresses":{"street":"Long Street 1"
4                             ,"town":"Short Village"}}
```

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

5.2.2.4. Badgerfish notation

Badgerfish notation is the other notation based on Jettison. 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.

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

```
JSONConfiguration.badgerFish().build()
```

and the output JSON for Example 5.9, “JAXB beans for JSON supported notations description, initialization” will be as follows.

Example 5.22. JSON expression produced using `badgerfish` notation

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

5.2.3. Examples

Download https://maven.java.net/service/local/artifact/maven/redirect?r=releases&g=com.sun.jersey.samples&a=json-from-jaxb&v=1.12&c=project&e=zip or https://maven.java.net/service/local/artifact/maven/redirect?r=releases&g=com.sun.jersey.samples&a=jmaki-backend&v=1.12&c=project&e=zip to get a more complex example using JAXB based JSON support.

5.3. Low-Level JSON support

Using this approach means you will be using JSONObject and/or JSONArray classes for your data representations. These classes are actually taken from Jettison project, but conform to the description provided at http://www.json.org/java/index.html.

The biggest advantage here is, that you will gain full control over the JSON format produced and consumed. On the other hand, dealing with your data model objects will probably be a bit more complex, than when taking the JAXB based approach. Differencies are depicted at the following code snipets.

Example 5.23. JAXB bean creation

```
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 (in terms of resulting JSON expression), you would need several more lines of code.

Example 5.24. Constructing a JSONObject

```
1 JSONObject myObject = new JSONObject();
2 myObject.JSONObject myObject = new JSONObject();
3 try {
4   myObject.put("name", "Agamemnon");
5   myObject.put("age", 32);
6 } catch (JSONException ex) {
7   LOGGER.log(Level.SEVERE, "Error ...", ex);
8 }
```

5.3.1. Examples

Download https://maven.java.net/service/local/artifact/maven/redirect?r=releases&g=com.sun.jersey.samples&a=bookmark&v=1.12&c=project&e=zip to get a more complex example using low-level JSON support.

Chapter 6. Declarative Hyperlinking

Table of Contents

RESTful APIs must be hypertext-driven. JAX-RS currently offers UriBuilder to simplify URI creation but Jersey adds an additional annotation-based alternative that is described here.

6.1. Links in Representations

Links are added to representations using the `@Ref` annotation on entity class fields. The Jersey runtime is responsible for injecting the appropriate URI into the field prior to serialization by a message body writer. E.g. consider the following resource and entity classes:

```
@Path("widgets")
public class WidgetsResource {
  @GET
  public Widgets get() {
    return new Widgets();
  }
}

public class Widgets {
  @Ref(resource=WidgetsResource.class)
  URI u;
}
```

After a call to `WidgetsResource#get`, the Jersey runtime will inject the value "/context/widgets"[1] into the returned `Widgets` instance. If an absolute URI is desired instead of an absolute path then the annotation can be changed to `@Ref(resource=WidgetsResource.class, style=ABSOLUTE)`.

The above usage works nicely when there's already a URI template on a class that you want to reuse. If there's no such template available then you can use a literal value instead of a reference. E.g. the following is equivalent to the earlier example: `@Ref(value="widgets", style=ABSOLUTE)`.

6.2. Binding Template Parameters

Referenced or literal templates may contain parameters. Two forms of parameters are supported:

- URI template parameters, e.g. `widgets/{id}` where `{id}` represents a variable part of the URI.
- EL expressions, e.g. `widgets/${instance.id}` where `${instance.id}` is an EL expression.

Parameter values can be extracted from three implicit beans:

`instance`

: Represents the instance of the class that contains the annotated field.

`entity`

: Represents the entity class instance returned by the resource method.

`resource`

: Represents the resource class instance that returned the entity.

By default URI template parameter values are extracted from the implicit `instance` bean, i.e. the following two annotations are equivalent:

```
@Ref("widgets/{id}")
@Ref("widgets/${instance.id}")
```

The source for URI template parameter values can be changed using the `@Binding` annotation, E.g. the following two annotations are equivalent:

```
@Ref(value="widgets/{id}", bindings={
  @Binding(name="id" value="${resource.id}"}
)
@Ref("widgets/${resource.id}")
```

6.3. Conditional Link Injection

Link value injection can be made conditional by use of the `condition` property. The value of this property is a boolean EL expression and a link will only be injected if the condition expression evaluates to true. E.g.:

```
@Ref(value="widgets/${instance.id}/offers",
  condition="${instance.offers}")
URI offers;
```

In the above, a URI will only be injected into the `offers` field if the `offers` property of the instance is `true`.

6.4. Link Headers

HTTP Link headers can also be added to responses using annotations. Instead of annotating the fields of an entity class with `@Ref`, you instead annotate the entity class itself with `@Link`. E.g.:

```
@Link(
  value=@Ref("widgets/${resource.nextId}"),
  rel="next"
)
```

The above would insert a HTTP Link header into any response whose entity was thus annotated. The `@Link` annotation contains properties that map to the parameters of the HTTP Link header. The above specifies the link relation as `next`. All properties of the `@Ref` annotation may be used as described above.

Multiple link headers can be added by use of the `@Links` annotation which can contain multiple `@Link` annotations.

6.5. Configuration

Declarative hyperlinking support is provided in the form of a filter. First, the application must declare a dependency on the `jersey-server-linking` module:

```
<dependency>
  <groupId>com.sun.jersey</groupId>
  <artifactId>jersey-server-linking</artifactId>
  <version>${jersey-version}</version>
</dependency>
```

Second the filter needs to be installed in the application either programmatically by adding:

```
com.sun.jersey.server.linking.LinkFilter
```

to the list defined by:

```
com.sun.jersey.spi.container.ContainerResponseFilters
```

or using a `web.xml` init parameter:

```
<init-param>
  <param-name>com.sun.jersey.spi.container.ContainerResponseFilters</param-name>
  <param-value>com.sun.jersey.server.linking.LinkFilter</param-value>
</init-param>
```

See the jersey-server-linking-sample for more details.

---

[1] Where `/context` is the application deployment context.

Chapter 7. Jersey Test Framework

Table of Contents

This chapter will present how to write tests for your resources using Jersey Test Framework and how to run them in various containers. Additionally it will explain how to create new module for not yet supported container.

Jersey currently provides following modules:

- jersey-test-framework-grizzly
- jersey-test-framework-grizzly2
- jersey-test-framework-http
- jersey-test-framework-inmemory
- jersey-test-framework-embedded-glassfish
- jersey-test-framework-external

7.1. What is different in Jersey 1.2

There are some significant breaking changes in Jersey 1.2. In prior Jersey versions users were able to select container factory just by specifying it in some property. That was convenient from user perspective but not good from build perspective. Former test framework artifact was dependent on all containers which is useless in most cases (usually you test only with one container).

Solution to this is modularization - make module for each test container. It has one drawback: users will have to have other dependency in their applications, for example if you want to test on embedded grizzly container, you will declare (only) dependency on jersey test framework grizzly module. You can declare multiple test containers this way and select one by defining property `jersey.test.containerFactory`.

Another change (non-breaking) is renaming Jersey parameters which control container factory, used port and host name for external container. Old properties are still working but users are encouraged to use new ones.

Table 7.1. Property name changes

Prior Jersey 1.2	Jersey 1.2+
test.containerFactory	jersey.test.containerFactory
JERSEY_HTTP_PORT	jersey.test.port
JERSEY_HOST_NAME (used with external container)	jersey.test.host

7.2. Using test framework

When you want test your resources in maven-based project, you need to add dependency on one of the Jersey Test Framework modules. You can take a look at helloworld sample pom file.

There is declared dependency on:

```
<dependency>
<groupId>com.sun.jersey.jersey-test-framework</groupId>
<artifactId>jersey-test-framework-grizzly2</artifactId>
<version>${project.version}</version>
<scope>test</scope>
</dependency>
```

which means that Grizzly Web container (version 2.x) will be used for testing.

You can specify more than one module in dependencies and choose which module will be used by `jersey.test.containerFactory` property. Every module should contain at least one container factory.

jersey-test-framework-grizzly

```
com.sun.jersey.test.framework.spi.container.grizzly.web.GrizzlyWebTestContainerFactory
com.sun.jersey.test.framework.spi.container.grizzly.GrizzlyTestContainerFactory
```

jersey-test-framework-grizzly2

```
com.sun.jersey.test.framework.spi.container.grizzly2.web.GrizzlyWebTestContainerFactory
com.sun.jersey.test.framework.spi.container.grizzly2.GrizzlyTestContainerFactory
```

jersey-test-framework-http

```
com.sun.jersey.test.framework.spi.container.http.HTTPContainerFactory
```

jersey-test-framework-inmemory

```
com.sun.jersey.test.framework.spi.container.inmemory.InMemoryTestContainerFactory
```

jersey-test-framework-embedded-glassfish

```
com.sun.jersey.test.framework.spi.container.embedded.glassfish.EmbeddedGlassFishTestContainerFactory
```

jersey-test-framework-external

```
com.sun.jersey.test.framework.spi.container.external.ExternalTestContainerFactory
```

Basically you can just add dependency on single module and its container factory would be used. Problem is when you specify module which has more than one container factory or multiple modules. If this happen, test framework will choose factory using following rules:

```
if("jersey.test.containerFactory" not specified)
    look for factories
    if(factories.count == 1)
        use found factory
    else
        if(com.sun.jersey.test.framework.spi.container.grizzly2.web.GrizzlyWebTestContainerFactory is present)
            use it // current default jersey test container factory
        else
            use first found and log warning
else
    use factory class specified in "jersey.test.containerFactory"
```

That means if your project depends on multiple test framework modules and you want to control which will be used, you have to declare which one in property called "jersey.test.containerFactory", for example like this: `mvn clean install -Djersey.test.containerFactory=com.sun.jersey.test.framework.spi.container.inmemory.InMemoryTestContainerFactory`

7.3. Creating tests

Jersey Test Framework uses JUnit version 4.X, so if you can write standard unit tests, you can easily create Jersey Test. You need to declare test as a descendant of JerseyTest class.

```
public class MainTest extends JerseyTest {

    public MainTest()throws Exception {
        super("com.sun.jersey.samples.helloworld.resources");
    }

    @Test
    public void testHelloWorld() {
        WebResource webResource = resource();
        String responseMsg = webResource.path("helloworld").get(String.class);
        assertEquals("Hello World", responseMsg);
    }

}
```

Note `super` call in constructor - it passes list of package names to scan (it really is a list, JerseyTest constructor has variable argument count). Another useful method is `resource()` which returns WebResource instance with URI set to base URI of your application. You can get preconfigured Jersey Client instance similarly by calling `client()` method.

7.4. Creating own module

Creating your own module is pretty straightforward, you just have to implement `com.sun.jersey.test.framework.spi.container.TestContainerFactory` and `com.sun.jersey.test.framework.spi.container.TestContainer`. TestContainer factory is there basically for returning TestContainer instance and TestContainer has self-explanatory methods: `start()`, `stop()`, `getClient()` and `getBaseURI()`. I recommend taking look at source code and read javadoc of these two classes, all you need is there.

You should be avare of another thing when implementing own jersey test framework module. If you want it to be usable by running just `mvn clean install` (when only your module is specified), you need to add META-INF/services/com.sun.jersey.test.framework.spi.container.TestContainerFactory file into your jar and put there your factory class (fully classified) name.

7.5. Running tests outside Maven

Since Jersey is Maven based project, executing tests without Maven can be painful. You have to have everything needed present on classpath and by everything is meant following list:

- jersey-server
- jersey-core
- jsr311-api

- asm
- jersey-test-framework-grizzly
- jersey-test-framework-core
- jersey-client
- javax.servlet
- junit
- jaxb-impl
- jaxb-api
- stax-api
- activation
- grizzly-servlet-webserver
- grizzly-http
- grizzly-framework
- grizzly-utils
- grizzly-rcm
- grizzly-portunif
- grizzly-http-servlet
- servlet-api

This is needed to run `helloworld` sample tests, if you want run something more complex or with different test container (grizzly is used here), you may need to add other application specific dependencies (and remove some as well).

As was already written above, Jersey test is descendant of standard unit test so it can be run same way. You can execute it by executing `org.junit.runner.JUnitCore` and passing your test class name as parameter, from ant (junit-task) or whatever you are used to.

Chapter 8. OSGi

Table of Contents

8.1. Feature Overview

OSGi support was added to the Jersey version 1.2. Since then, you should be able to utilize standard OSGi means to run Jersey based web applications in OSGi runtime as described in the OSGi Service Platform Enterprise Specification. The specification could be downloaded from http://www.osgi.org/Download/Release4V42.

The two supported ways of running an OSGi web application are

- WAB (Web Application Bundle)
- Http Service

WAB is in fact just an OSGified WAR archive. Http Service feature allows you to publish Java EE Servlets in the OSGi runtime.

Two examples were added to the Jersey distribution to depict the above mentioned features and show how to use them with Jersey

- Hello world WAB
- Simple OSGi Http Service application

Both examples are multi-module maven projects and both consist of an application OSGi bundle module and a test module. The tests are based on Pax Exam framework. Both examples also include a readme file containing instructions how to manually run the application using Apache Felix framework.

The rest of the chapter describes how to run the above mentioned examples on GlassFish 3.1 application server. Since GlassFish utilizes Apache Felix, an OSGi runtime comes out of the box with GlassFish. However, for security reasons, the OSGi shell has been turned off. You can explicitly enable it. There is a system property called `org.glassfish.additionalOSGiBundlesToStart` in `domain.xml` that contains a list of additional bundles to be started by glassfish. One has to add `org.apache.felix.shell.remote` there. Here are a few ways to do it:

Option #1: You can delete the property and create it again using following `asadmin` commands:

```
asadmin delete-jvm-options --target server-config -
Dorg.glassfish.additionalOSGiBundlesToStart=org.apache.felix.shell,org.apache.felix.gogo.runtime,org.apache.felix.gogo.shell,org.apache.felix.gogo.command

asadmin create-jvm-options --target server-config -
Dorg.glassfish.additionalOSGiBundlesToStart=org.apache.felix.shell,org.apache.felix.gogo.runtime,org.apache.felix.gogo.shell,org.apache.felix.gogo.command,
```

Option #2: You can use GlassFish Admin Console to edit the `jvm-option`.

Option #3: You can edit the `domain.xml` directly.

Presuming you have the default GlassFish instance running, after enabling shell as described above, you should now be able to connect to the Felix console with

```
        telnet localhost 6666
```

You should then see Apache Felix prompt similar to following

```
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
____________________________
Welcome to Apache Felix Gogo

g!
```

8.2. WAB Example

As mentioned above, WAB is just an OSGified WAR archive. Besides the ususal OSGi headers it must in addition contain a special header, `Web-ContextPath`, specifying the web application context path. Our WAB has (beside some other) the following headers present in the manifest

```
Web-ContextPath: helloworld
Webapp-Context: helloworld
Bundle-ClassPath: WEB-INF/classes
```

where the second one is ignored by GlassFish, but is needed by other containers not fully compliant with the OSGi Enterprise Specification mentioned above. The third manifest header worth

mentioning is the `Bundle-ClassPath` specifying where to find the application Java classes within the bundle archive.

For more detailed information on the example please see the source code.

Following is the listing showing how to actually install and run the WAB on GlassFish. Be sure to replace <version> with the current Jersey version and <repository> with either snapshots or releases based on whether you depend on a snapshot or stable release version of Jersey respectively:

```
g! install https://maven.java.net/service/local/artifact/maven/redirect?r=<repository>&g=com.sun.jersey.samples.helloworld-osgi-webapp&a=war-bundle&v=<vers
Bundle ID: 246
g! start 246
```

In the above listing, the number 246 represents handler to the OSGi bundle you have just installed. Bundle numbers are allocated dynamically. It means, you might be given a different handler. It is important to always use the correct bundle number as specified by the Felix runtime in the "Bundle ID:" response.

After the WAB gets installed and activated by the above mentioned commands, you should be able to access the deployed Jersey resource at http://localhost:8080/helloworld/webresources/helloworld

8.3. Http Service Example

OSGi Http Service support currently does not come out of the box with GlassFish, but is provided with a separate OSGi bundle, http://search.maven.org/remotecontent?filepath=org/glassfish/osgi-http/3.2-b03/osgi-http-3.2-b03.jar. You will need to enable the feature first, by installing that bundle. After that, you can install and activate the Jersey application bundle. Following is the listing showing how both bundles could be installed and activated (Be sure to replace <version> with the current Jersey version and <repository> with either snapshots or releases based on whether you depend on a snapshot or stable release version of Jersey respectively):

```
g! install http://search.maven.org/remotecontent?filepath=org/glassfish/osgi-http/3.2-b03/osgi-http-3.2-b03.jar
Bundle ID: 247
g! install https://maven.java.net/service/local/artifact/maven/redirect?r=<repository>&g=com.sun.jersey.samples.osgi-http-service&a=bundle&v=<version>&e=ja
Bundle ID: 248
g! start 247 248
```

Now you should be able to access the Jersey resource at http://localhost:8080/osgi/jersey-http-service/status

Finally, to close the Felix console session just press `[Ctrl]-[d]`.

Chapter 9. JRebel support

Jersey-jrebel plugin is no longer provided by Jersey team, Zeroturnaround has its own implementation, see http://zeroturnaround.com/jrebel/features/.

Chapter 10. Experimental Features

Table of Contents

This chapter describes experimental features of Jersey that are only available in maven SNAPSHOT releases. Such features are not intended to be utilized for stable development as the APIs may change.

Providing such experimental features allows developers to experiment and provide feedback, please send feedback to users@jersey.java.net.

10.1. Hypermedia Actions

10.1.1. Introduction

It is generally understood that, in order to follow the REST style, URIs should be assigned to anything of interest (resources) and a few, well-defined operations should be used to interact with these resources. For example, the state of a purchase order resource can be updated by POSTing or PATCHing a new value for the its state field. However, there are actions that cannot be easily mapped to read or write operations on resources. These operations are inherently more complex and their details are rarely of interest to clients. For example, given a purchase order resource, the operation of setting its state to `REVIEWED` may involve a number of different steps such as:

1. checking the customer's credit status
2. reserving product inventory
3. verifying per-customer quantity limits.

Clearly, this *workflow* cannot be equated to simply updating a field on a resource. Moreover, clients are generally uninterested in the details behind these type of workflows, and in some cases computing the final state of a resource on the client side, as required for a PUT operation, is impractical or impossible.

We call these operations *actions* and, because they are of interest to us, we turn them into *action resources*. An action resource is a sub-resource defined for the purpose of exposing workflow-related operations on parent resources. As sub-resources, action resources are identified by URIs that are relative to their parent. For instance, the following are examples of action resources:

```
http://.../orders/1/review
http://.../orders/1/pay
http://.../orders/1/ship
```

for purchase order "1" identified by `http://.../orders/1`.

Action resources as first-class citizens provide the necessary tools for developers to implement HATEOAS. A set of action resources defines --via their link relationships-- a *contract* with clients that has the potential to evolve over time depending on the application's state. For instance, assuming purchase orders are only reviewed once, the `review` action will become unavailable and the `pay` action will become available after an order is reviewed.

10.1.2. Hypermedia by Example

The notion of action resources naturally leads to discussions about improved client APIs to support them. Given that action resources are identified by URIs, no additional API is really necessary, but the use of client-side proxies and method invocations to trigger these actions seems quite natural. Additionally, the use of client proxies introduces a level of indirection that enables better support for *server evolution*, i.e. the ability of a server's contract to support certain changes without breaking existing clients. Finally, it has been argued that using client proxies is simply more natural for developers and less error prone as fewer URIs need to be constructed.

Rather than presenting all these extensions abstractly, we shall illustrate their use via an example. The complete example can found in hypermedia-action-sample. The *Purchase Ordering System* exemplifies a system in which customers can submit orders and where orders are guided by a workflow that includes states like `REVIEWED`, `PAYED` and `SHIPPED`. The system's model is comprised of 4 entities: `Order`, `Product`, `Customer` and `Address`. These model entities are controlled by 3 resource classes: `OrderResource`, `CustomerResource` and `ProductResource`. Addresses are

sub-resources that are also controlled by `CustomerResource`. An order instance refers to a single customer, a single address (of that customer) and one or more products. The XML representation (or view) of a sample order is shown below.

```
<order>
    <id>1</id>
    <customer>http://.../customers/21</customer>
    <shippingAddress>http://.../customers/21/address/1</shippingAddress>
    <orderItems>
        <product>http://.../products/3345</product>
        <quantity>1</quantity>
    </orderItems>
    <status>RECEIVED</status>
</order>
```

Note the use of URIs to refer to each component of an order. This form of *serialization by reference* is supported in Jersey using JAXB beans and the `@XmlJavaTypeAdapter` annotation. An `XmlAdapter` is capable of mapping an object reference in the model to a URI. Refer to hypermedia-action-sample for more information on how this mapping is implemented.

10.1.3. Server API

The server API introduces 3 new annotation types: `@Action`, `@ContextualActionSet` and `@HypermediaController`. The `@Action` annotation identifies a sub-resource as a *named action*. The `@ContextualActionSet` is used to support contextual contracts and must annotate a method that returns a set of action names. Finally, `@HypermediaController` marks a resource class as a *hypermedia controller class*: a class with one more methods annotated with `@Action` and at most one method annotated with `@ContextualActionSet`.

The following example illustrates the use of all these annotation types to define the `OrderResource` controller.[2]

```
1     @Path("/orders/{id}")
2     @HypermediaController(
3         model=Order.class,
4         linkType=LinkType.LINK_HEADERS)
5     public class OrderResource {
6
7         private Order order;
8
9         ...
10
11        @GET @Produces("application/xml")
12        public Order getOrder(@PathParam("id") String id) {
13            return order;
14        }
15
16        @POST @Action("review") @Path("review")
17        public void review(@HeaderParam("notes") String notes) {
18            ...
19            order.setStatus(REVIEWED);
20        }
21
22        @POST @Action("pay") @Path("pay")
23        public void pay(@QueryParam("newCardNumber") String newCardNumber) {
24            ...
25            order.setStatus(PAYED);
26        }
27
28        @PUT @Action("ship") @Path("ship")
29        @Produces("application/xml")
30        @Consumes("application/xml")
31        public Order ship(Address newShippingAddress) {
32            ...
33            order.setStatus(SHIPPED);
34            return order;
35        }
36
37        @POST @Action("cancel") @Path("cancel")
38        public void cancel(@QueryParam("notes") String notes) {
39            ...
40            order.setStatus(CANCELED);
41        }
42    }
```

The `@HypermediaController` annotation above indicates that this resource class is a hypermedia controller for the `Order` class. Each method annotated with `@Action` defines a link relationship and associated action resource. These methods are also annotated with `@Path` to make a sub-resource. There does not appear to be a need to use `@Action` and `@Path` simultaneously, but without the latter some resource methods may become ambiguous. In the future, we hope to eliminate the use of `@Path` when `@Action` is present. The element `linkType` selects the way in which URIs corresponding to action resources are serialized: in this case, using link headers. These link headers become part of the an order's representation. For instance, an order in the `RECEIVED` state, i.e. an order that can only be reviewed or canceled, will be represented as follows.

```
Link: <http://.../orders/1/review>;rel=review;op=POST
Link: <http://.../orders/1/cancel>;rel=cancel;op=POST
<order>
    <id>1</id>
    <customer>http://.../customers/21</customer>
    <shippingAddress>http://.../customers/21/address/1</shippingAddress>
    <orderItems>
        <product>http://.../products/3345</product>
        <quantity>1</quantity>
    </orderItems>
    <status>RECEIVED</status>
</order>
```

Without a method annotated with `@ContextualActionSet`, all actions are available at all times regardless of the state of an order. The following method can be provided to define a contextual contract for this resource.

```
1     @ContextualActionSet
2     public Set<String> getContextualActionSet() {
3         Set<String> result = new HashSet<String>();
4         switch (order.getStatus()) {
5             case RECEIVED:
6                 result.add("review");
7                 result.add("cancel");
8                 break;
9             case REVIEWED:
10                result.add("pay");
11                result.add("cancel");
```

```
                break;
            case PAYED:
                result.add("ship");
                break;
            case CANCELED:
            case SHIPPED:
                break;
        }
        return result;
    }
```

This method returns a set of action names based on the order's internal state; the values returned in this set correspond to the `@Action` annotations in the controller. For example, this contextual contract prevents shipping an order that has not been payed by only including the `ship` action in the `PAYED` state.

10.1.4. Client API

Although action resources can be accessed using traditional APIs for REST, including Jersey's client API, the use of client-side proxies and method invocations to trigger these actions seems quite natural. As we shall see, the use of client proxies also introduces a level of indirection that enables better support for server evolution, permitting the definition of contracts with various degrees of coupling.

Client proxies are created based on *hypermedia controller interfaces*. Hypermedia controller interfaces are Java interfaces annotated by `@HypermediaController` that, akin to the server side API, specify the name of a model class and the type of serialization to use for action resource URIs. The client-side model class should be based on the representation returned by the server; in particular, in our example the client-side model for an `Order` uses instances of `URI` to link an order to a customer, an address and a list of products.

```
    @HypermediaController(
        model=Order.class,
        linkType=LinkType.LINK_HEADERS)
    public interface OrderController {

        public Order getModel();

        @Action("review")
        public void review(@Name("notes") String notes);

        @POST @Action("pay")
        public void pay(@QueryParam("newCardNumber") String newCardNumber);

        @Action("ship")
        public Order ship(Address newShippingAddress);

        @Action("cancel")
        public void cancel(@Name("notes") String notes);
    }
```

The `@Action` annotation associates an interface method with a link relation and hence an action resource on the server. Thus, invoking a method on the generated proxy results in an interaction with the corresponding action resource. The way in which the method invocation is mapped to an HTTP request depends on the additional annotations specified in the interface. For instance, the pay action in the example above indicates that it must use a `POST` and that the `String` parameter `newCardNumber` must be passed as a query parameter. This is an example of a *static* contract in which the client has built-in knowledge of the way in which an action is defined by the server.

In contrast, the `review` action only provides a name for its parameter using the `@Name` annotation. This is an example of a *dynamic* contract in which the client is only coupled to the `review` link relation and the knowledge that this relation requires `notes` to be supplied. The exact interaction with the `review` action must therefore be discovered dynamically and the `notes` parameter mapped accordingly. The Jersey client runtime uses WADL fragments that describe action resources to map these method calls into HTTP requests.

The following sample shows how to use `OrderController` to generate a proxy to review, pay and ship an order. For the sake of the example, we assume the customer that submitted the order has been suspended and needs to be activated before the order is reviewed. For that purpose, the client code retrieves the customer URI from the order's model and obtains an instance of `CustomerController`.[3]

```
    // Instantiate Jersey's Client class
    Client client = new Client();

    // Create proxy for order and retrieve model
    OrderController orderCtrl = client.view(
        "http://.../orders/1", OrderController.class);

    // Create proxy for customer in order 1
    CustomerController customerCtrl = client.view(
        orderCtrl.getModel().getCustomer(),
        CustomerController.class);

    // Activate customer in order 1
    customerCtrl.activate();

    // Review and pay order
    orderCtrl.review("approve");
    orderCtrl.pay("123456789");

    // Ship order
    Address newAddress = getNewAddress();
    orderCtrl.ship(newAddress);
```

The client runtime will automatically update the action set throughout a conversation: for example, even though the `review` action does not produce a result, the HTTP response to that action still includes a list of link headers defining the contextual action set, which in this case will consist of the pay and `cancel` actions but not the `ship` action. An attempt to interact with any action not in the contextual action set will result in a client-side exception if using proxies (as shown above), or a server-side exception if using some other API.

The following annotations are allowed in hypermedia controller interfaces: `@Name`, `@Consumes`, `@Produces`, `@CookieParam`, `@FormParam`, `@HeaderParam`, `@QueryParam`, as well as any HTTP method annotation like `@POST`, `@PUT`, etc. All other annotations not listed here are unsupported at this time (of note, `@MatrixParam` is not supported). The `@Name` annotation used in Jersey is defined in `com.sun.jersey.core.hypermedia`.

10.1.5. Server Evolution

In the last section, we have seen how the use of client proxies based on *partially* annotated interfaces facilitates server evolution. An interface method annotated with `@Action` and `@Name` represents a *loosely-coupled* contract with a server. Changes to action resource URIs, HTTP methods and parameter types on the server will not require a client re-spin. Naturally, as in all client-server architectures, it is always possible to break backward compatibility, but the ability to support more dynamic contracts --usually at the cost of additional processing time-- is still an area of investigation.

We see this form of hypermedia support in Jersey as a small step in this direction, showing the potential of using dynamic meta-data for the definition of these type of contracts. The use of dynamic meta-data requires the server to return WADL fragments when using `OPTIONS` on an action resource URI. When dynamic meta-data is not available (for example if using an older version of Jersey on the server) client interfaces should be fully annotated and a *tighly-coupled* contract should be used instead.

10.1.6. Configuring Hypermedia in Jersey

There are a few configuration options in hypermedia-action-sample that are worth highlighting. First, a special filter factory `HypermediaFilterFactory` must be enabled. This can be done programmatically by adding

```
com.sun.jersey.server.hypermedia.filter.HypermediaFilterFactory
```

to the list defined by

```
com.sun.jersey.spi.container.ResourceFilters
```

or declaratively in the application's `web.xml` file. Refer to the Java class `com.sun.jersey.samples.hypermedia.Main` in the hypermedia sample for an example of how to do this using Grizzly (which can be executed with `mvn install exec:java`). Second, hypermedia applications must included the following Maven runtime dependencies:

```
        <dependency>
            <groupId>com.sun.jersey.experimental.hypermedia-action</groupId>
            <artifactId>hypermedia-action-server</artifactId>
            <version>${jersey-version}</version>
        </dependency>
        <dependency>
            <groupId>com.sun.jersey.experimental.hypermedia-action</groupId>
            <artifactId>hypermedia-action-client</artifactId>
            <version>${jersey-version}</version>
        </dependency>
```

Last, even though the annotation `@HypermediaController` has a `linkType` element, only the `LinkType.LINK_HEADERS` option is currently supported and must be used by both clients and servers.

---

[2] Several details about this class are omitted for clarity. The reader is referred to the hypermedia sample for more details.

[3] `CustomerController` is a hypermedia controller interface akin to `OrderController` which is omitted since it does not highlight any additional feature.

Chapter 11. Dependencies

Table of Contents

Jersey is built, assembled and installed using Maven. Jersey is deployed to the Java.Net maven repository at the following location: http://maven.java.net/. The Jersey modules can be browsed at the following location: https://maven.java.net/content/repositories/releases/com/sun/jersey. Jars, Jar sources, Jar JavaDoc and samples are all available on the java.net maven repository.

A zip file containing all maven-based samples can be obtained here. Individual zip files for each sample may be found by browsing the samples directory.

An application depending on Jersey requires that it in turn includes the set of jars that Jersey depends on. Jersey has a pluggable component architecture so the set of jars required to be include in the class path can be different for each application.

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

Developers using maven are likely to find it easier to include and manage dependencies of their applications than developers using ant or other build technologies. This document will explain to both maven and non-maven developers how to depend on Jersey for their application. Ant developers are likely to find the Ant Tasks for Maven very useful. For the convenience of non-maven developers the following are provided:

- A zip of Jersey containing the Jersey jars, core dependencies (it does not provide dependencies for third party jars beyond those for JSON support) and JavaDoc.
- A jersey bundle jar to avoid the dependency management of multiple jersey-based jars.

Jersey's runtime dependences are categorized into the following:

- Core server. The minimum set of dependences that Jersey requires for the server.
- Core client. The minimum set of dependences that Jersey requires for the client.
- Container. The set of container dependences. Each container provider has it's own set of dependences.
- Entity. The set of entity dependencies. Each entity provider has it's own set of dependences.
- Tools. The set of dependencies required for runtime tooling.
- Spring. The set of dependencies required for Spring.
- Guice. The set of dependencies required for Guice.

All dependences in this documented are referenced by hyper-links

11.1. Core server

Maven developers require a dependency on the jersey-server module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-server</artifactId>
    <version>1.12</version>
</dependency>
```

If you want to depend on Jersey snapshot versions the following repository needs to be added to the 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>
```

Non-maven developers require:

- jersey-server.jar ,
- jersey-core.jar ,
- asm.jar

or, if using the jersey-bundle:

- jersey-bundle.jar ,
- asm.jar

For Ant developers the Ant Tasks for Maven may be used to add the following to the ant document such that the dependencies do not need to be downloaded explicitly:

```
<artifact:dependencies pathId="dependency.classpath">
  <dependency groupId="com.sun.jersey"
              artifactId="jersey-server"
              version="1.12"/>
  <artifact:remoteRepository id="maven-repository.java.net"
                             url="http://maven.java.net/" />
  <artifact:remoteRepository id="maven1-repository.java.net"
                             url="http://download.java.net/maven/1"
                             layout="legacy" />
</artifact:dependencies>
```

The path id “dependency.classpath” may then be referenced as the classpath to be used for compiling or executing. Specifically the asm.jar dependency is required when either of the following com.sun.jersey.api.core.ResourceConfig implementations are utilized:

- com.sun.jersey.api.core.ClasspathResourceConfig ; or
- com.sun.jersey.api.core.PackagesResourceConfig

By default Jersey will utilize the ClasspathResourceConfig if an alternative is not specified. If an alternative is specified that does not depend on the asm.jar then it is no longer necessary to include the asm.jar in the minimum set of required jars.

11.2. Core client

Maven developers require a dependency on the jersey-client module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-client</artifactId>
    <version>1.12</version>
</dependency>
```

Non-maven developers require:

- jersey-client.jar ,
- jersey-core.jar

or, if using the jersey-bundle:

- jersey-bundle.jar

The use of client with the Apache HTTP client to make HTTP request and receive HTTP responses requires a dependency on the jersey-apache-client module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey.contribs</groupId>
    <artifactId>jersey-apache-client</artifactId>
    <version>1.12</version>
</dependency>
```

11.3. Container

11.3.1. Grizzly HTTP Web server

Maven developers, deploying an application using the Grizzly2 HTTP Web server, require a dependency on the jersey-grizzly2 module.

Non-maven developers require: jersey-grizzly2.jar, grizzly-http-server.jar, and grizzly-http.jar

11.3.2. Grizzly Servlet container

Maven developers, deploying an application using the Grizzly Servlet container, require a dependency on the jersey-grizzly2-servlet module.

Non-maven developers require: jersey-grizzly2-servlet.jar, grizzly-http-servlet.jar, jersey-grizzly2.jar, javax.servlet-3.1.jar

11.3.3. Simple HTTP Web server

Maven developers, deploying an application using the Simple HTTP Web server, require a dependency on the jersey-simple-server module.

11.3.4. Light weight HTTP server

Deploying an application using the light weight HTTP server requires no additional dependences, as Java SE 6 already contains everything needed.

11.3.5. Servlet

Deploying an application on a servlet container requires a deployment dependency with that container.

See the Java documentation here on how to configure the servlet container.

Using servlet: `com.sun.jersey.spi.container.servlet.ServletContainer` requires a dependency on the jersey-servlet module.

Maven developers using servlet: `com.sun.jersey.server.impl.container.servlet.ServletAdaptor` in a non-EE 5 servlet require a dependency on the persistence-api module in addition.

Non-Maven developers require: persistence-api.jar

11.4. Entity

11.4.1. JAXB

XML serialization support of Java types that are JAXB beans requires a dependency on the JAXB reference implementation version 2.x or higher (see later for specific version constraints with respect to JSON support). Deploying an application for XML serialization support requires no additional dependences, since Java SE 6 ships with JAXB 2.x support.

Maven developers, using JSON serialization support of JAXB beans when using the MIME media type `application/json` require a dependency on the jersey-json module (no explicit dependency on jaxb-impl is required). This module depends on the JAXB reference implementation version 2.1.12 or greater, and such a version is required when enabling support for the JAXB natural JSON convention. For all other supported JSON conventions any JAXB 2.x version may be utilized. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-json</artifactId>
    <version>1.12</version>
</dependency>
```

Non-maven developers require:

- jackson-core-asl.jar ,
- jackson-mapper-asl.jar ,
- jackson-jaxrs.jar ,
- jettison.jar ,
- jaxb-impl.jar ,
- jaxb-api.jar ,
- activation.jar ,
- stax-api.jar

and additionally, if not depending on the jersey-bundle.jar, non-maven developers require: jersey-json.jar

Maven developers, using Fast Infoset serialization support of JAXB beans with using the MIME media type `application/fastinfoset` require a dependency on the jersey-fastinfoset module (no dependency on jaxb-impl is required). The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-fastinfoset</artifactId>
    <version>1.12</version>
</dependency>
```

Non-maven developers require:

- FastInfoset.jar ,
- jaxb-impl.jar ,
- jaxb-api.jar ,
- activation.jar ,
- stax-api.jar

and additionally, if not depending on the jersey-bundle.jar, non-maven developers require: jersey-fastinfoset.jar

11.4.2. Atom

The use of the Java types `org.apache.abdera.model.{Categories, Entry, Feed, Service}` requires a dependency on Apache Abdera.

Maven developers require a dependency on the jersey-atom-abdera module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey.contribs</groupId>
    <artifactId>jersey-atom-abdera</artifactId>
    <version>1.12</version>
</dependency>
```

The use of the Java types `com.sun.syndication.feed.atom.Entry` and `com.sun.syndication.feed.atom.Feed` requires a dependency on ROME version 0.9 or higher.

Maven developers require a dependency on the jersey-atom module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-atom</artifactId>
    <version>1.12</version>
</dependency>
```

Non-maven developers require:

- rome.jar ,
- jdom.jar

and additionally, if not depending on the jersey-bundle.jar, non-maven developers require: jersey-atom.jar

11.4.3. JSON

The use of the Java types `org.codehaus.jettison.json.JSONObject` and `org.codehaus.jettison.json.JSONArray` requires Jettison version 1.1 or higher.

Maven developers require a dependency on the jersey-json module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-json</artifactId>
    <version>1.12</version>
</dependency>
```

Non-maven developers require: jettison.jar and additionally, if not depending on the jersey-bundle.jar non-maven developers require: jersey-json.jar

11.4.4. Mail and MIME multipart

The use of the Java type `javax.mail.internet.MimeMultipart` requires Java Mail version 1.4 or higher.

Maven developers require a dependency on the java-mail module.

Non-maven developers require:

- mail.jar ,
- activation.jar

Jersey ships with a high-level MIME multipart API. Maven developers requires a dependency on the jersey-multipart module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey.contribs</groupId>
    <artifactId>jersey-multipart</artifactId>
    <version>1.12</version>
</dependency>
```

Non-maven developers require:

- mimepull.jar ,
- jersey-multipart.jar

11.4.5. Activation

The use of the Java type `javax.activation.DataSource` requires no additional dependencies, as Java SE 6 ships everything needed.

11.5. Tools

By default WADL for resource classes is generated dynamically at runtime. WADL support requires a dependency on the JAXB reference implementation version 2.x or higher. Deploying an application for WADL support requires no additional dependences, since Java SE 6 ships with JAXB 2.x support.

The WADL ant task requires the same set of dependences as those for runtime WADL support.

11.6. Spring

Maven developers, using Spring 2.0.x or Spring 2.5.x, require a dependency on the jersey-spring module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey.contribs</groupId>
    <artifactId>jersey-spring</artifactId>
    <version>1.12</version>
</dependency>
```

See the Java documentation here on how to integrate Jersey-based Web applications with Spring.

11.7. Guice

Maven developers, using Guice 2.0, require a dependency on the jersey-guice module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey.contribs</groupId>
    <artifactId>jersey-guice</artifactId>
    <version>1.12</version>
</dependency>
```

See the Java documentation here on how to integrate Jersey-based Web applications with Guice.

The Jersey Guice support may also be used with GuiceyFruit, a set of extensions on top of Guice 2.0, such as support for Java EE artifacts like `@PostConstruct`/`@PreDestroy`, `@Resource` and `@PersistenceContext`. To avail of GuiceyFruit features exclude the guice dependency from the maven central repo and add the following:

```
<dependency>
    <groupId>org.guiceyfruit</groupId>
    <artifactId>guiceyfruit</artifactId>
    <version>2.0</version>
</dependency>
...
<repository>
    <id>guice-maven</id>
    <name>guice maven</name>
    <url>http://guiceyfruit.googlecode.com/svn/repo/releases/</url>
</repository>
```

11.8. Jersey Test Framework

NOTE that breaking changes have occurred between 1.1.1-ea and 1.1.2-ea. See the end of this section for details.

Jersey Test Framework allows you to test your RESTful Web Services on a wide range of containers. It supports light-weight containers such as Grizzly, Embedded GlassFish, and the Light Weight HTTP Server in addition to regular web containers such as GlassFish and Tomcat. Developers may plug in their own containers.

A developer may write tests using the Junit 4.x framework can extend from the abstract JerseyTest class.

Maven developers require a dependency on the jersey-test-framework-grizzly module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey.jersey-test-framework</groupId>
```

```
    <artifactId>jersey-test-framework-grizzly</artifactId>
    <version>1.12</version>
    <scope>test</scope>
</dependency>
```

When utilizing an embedded container this framework can manage deployment and testing of your web services. However, the framework currently doesn't support instantiating and deploying on external containers.

The test framework provides the following test container factories:

- `com.sun.jersey.test.framework.spi.container.http.HTTPContainerFactory` for testing with the Light Weight HTTP server.
- `com.sun.jersey.test.framework.spi.container.inmemory.InMemoryTestContainerFactory` for testing in memory without using HTTP.
- `com.sun.jersey.test.framework.spi.container.grizzly.GrizzlyTestContainerFactory` for testing with low-level Grizzly.
- `com.sun.jersey.test.framework.spi.container.grizzly.web.GrizzlyWebTestContainerFactory` for testing with Web-based Grizzly.
- `com.sun.jersey.test.framework.spi.container.grizzly2.GrizzlyTestContainerFactory` for testing with low-level Grizzly2.
- `com.sun.jersey.test.framework.spi.container.grizzly2.web.GrizzlyWebTestContainerFactory` for testing with Web-based Grizzly2.
- `com.sun.jersey.test.framework.spi.container.embedded.glassfish.EmbeddedGlassFishTestContainerFactory` for testing with embedded GlassFish v3
- `com.sun.jersey.test.framework.spi.container.external.ExternalTestContainerFactory` for testing with application deployed externally, for example to GlassFish or Tomcat.

The system property `jersey.test.containerFactory` is utilized to declare the default test container factory that shall be used for testing, the value of which is the fully qualified class name of a test container factory class. If the property is not declared then the GrizzlyWebTestContainerFactory is utilized as default test container factory.

To test a maven-based web project with an external container such as GlassFish, create the war file then deploy as follows (assuming that the pom file is set up for deployment):

```
mvn clean package -Dmaven.test.skip=true
```

Then execute the tests as follows:

```
mvn test \ -Djersey.test.containerFactory=com.sun.jersey.test.framework.spi.container.embedded.glassfish.external.ExternalTestContainerFactory \
-DJERSEY_HTTP_PORT=<HTTP_PORT>
```

Breaking changes from 1.1.1-ea to 1.1.2-ea

The maven project groupId has changed from `com.sun.jersey.test.framework` to `com.sun.jersey`

The extending of Jersey unit test and configuration has changed. See here for an example.

See the blog entry on Jersey Test Framework for detailed instructions on how to use 1.1.1-ea version of the framework in your application.

Chapter 12. Jersey with GlassFish

Table of Contents

This chapter will present instructions on how to use Jersey with Glassfish when Jersey is distributed in the war and to manually upgrade the Jersey version bundled with a GlassFish installation.

12.1. Overriding Jersey with war files

To override the version of Jersey distributed in GlassFish with a version of Jersey distributed in a war file ensure that class loader delegation is set to false in WEB-INF/sun-web.xml or WEB-INF/glassfish-web.xml. For example:

```
<sun-web-app error-url="">
    <context-root>/context</context-root>
    <class-loader delegate="false"/>
</sun-web-app>
```

In the GlassFish admin console, go to Configuration->JVM Settings, switch to the JVM Options tab and add the following option:

```
-Dcom.sun.enterprise.overrideablejavaxpackages=javax.ws.rs,javax.ws.rs.core,javax.ws.rs.ext
```

Restart GlassFish for new JVM settings to take effect.

12.2. Upgrading Jersey in GlassFish

12.2.1. GlassFish v3.0 and 3.0.1

GlassFish v3.0 comes up with Jersey version 1.1.4.1. GlassFish v3.0.1 with Jersey version 1.1.5. To upgrade these to 1.12 you will need to replace certain files manually.

GlassFish uses Jersey internally in it's REST administration API, and the Update Center client would not allow you to upgrade in order to prevent this functionality. The workaround described in this section is known to work, but it is not currently a tested and supported scenario. Please keep in mind things could break. It is recommended to back up all all impacted files. The actual replace steps follow.

Remove the existing Jersey files: Stop all running server instances. Then remove the following files from the GlassFish installation directory:

- glassfish/modules/jsr311-api.jar
- glassfish/modules/jersey-gf-bundle.jar
- glassfish/modules/jersey-gf-statsproviders.jar
- glassfish/modules/jersey-multipart.jar
- glassfish/modules/jackson-core-asl.jar
- glassfish/modules/jettison.jar
- glassfish/modules/mimepull.jar

Download the new Jersey version archive from https://maven.java.net/service/local/artifact/maven/redirect?r=releases&g=com.sun.jersey.glassfish.v3&a=jersey-gfv3-core&v=1.12&c=project&e=zip and unzip it's `glassfish/modules` content into the GlassFish installation directory (to the `glassfish/modules` subdirectory there).

To install also Jersey examples in addition, download https://maven.java.net/service/local/artifact/maven/redirect?r=releases&g=com.sun.jersey.glassfish.v3&a=jersey-gfv3-docs-and-samples&v=1.12&c=project&e=zip and unzip it's content into the GlassFish installation directory.

To be 100 % sure the changes take effect, you might also want to remove the felix cache, which is located in `glassfish/domains/domain1/osgi-cache` directory. This directory gets created upon the first start of the GlassFish instance.

12.2.2. GlassFish 3.1

GlassFish 3.1 is released. GlassFish 3.1 builds can be downloaded from http://glassfish.java.net/downloads/3.1-final.html

If you want to update the Jersey bits in GlassFish with the latest Jersey snapshot version, or if you want to install Jersey documentation and examples, you would need to do the following steps:

Remove the existing Jersey files: Stop all running server instances. Then remove the following files from the GlassFish installation directory:

- glassfish/modules/jersey-core.jar
- glassfish/modules/jersey-client.jar
- glassfish/modules/jersey-gf-server.jar
- glassfish/modules/jersey-json.jar
- glassfish/modules/jersey-multipart.jar
- glassfish/modules/jersey-gf-statsproviders.jar
- glassfish/modules/jackson-core-asl.jar
- glassfish/modules/jackson-mapper-asl.jar
- glassfish/modules/jackson-jaxrs.jar
- glassfish/modules/jettison.jar

Download the new Jersey version archive from https://maven.java.net/service/local/artifact/maven/redirect?r=releases&g=com.sun.jersey.glassfish.v3&a=jersey-gfv3-core&v=1.12&c=project&e=zip and unzip it's `glassfish/modules` content into the GlassFish installation directory (to the `glassfish/modules` subdirectory there).

To install also Jersey examples in addition, download (you can change the version in the link to get another non-snapshot version of Jersey) https://maven.java.net/service/local/artifact/maven/redirect?r=releases&g=com.sun.jersey.glassfish.v3&a=jersey-gfv3-docs-and-samples&v=1.12&c=project&e=zip and unzip it's content into the GlassFish installation directory.

To be 100 % sure the changes take effect, you might also want to remove the felix cache, which is located in `glassfish/domains/domain1/osgi-cache` directory. This directory gets created upon the first start of the GlassFish instance.

Chapter 13. Building and testing Jersey

Table of Contents

13.1. Checking out the source

The Jersey source code is available from the Subversion repository located at http://java.net/projects/jersey/sources/svn/show. You can also read more about how the top level of the `jersey' Subversion repository is structured at that same link.

To check out the trunk where active development on the next release occurs use the following command:

```
svn checkout https://svn.java.net/svn/jersey~svn/trunk/jersey jersey --username <username>
```

If you are new to Subversion, you may want to visit the Subversion Project website and read Version Control with Subversion.

Stable releases of Jersey are tagged in the location http://java.net/projects/jersey/sources/svn/show/tags.

The source code may be browsed using FishEye.

13.2. Building using Maven

Java SE 6 or greater is required. Maven 2.2.1 or greater is recommended.

It is recommended to build the whole of Jersey after you have initially checked out the source code. To build all of Jersey use the following command from the checked out `jersey` directory:

```
mvn clean install
```

To skip all the tests do:

```
mvn -Dmaven.test.skip=true clean install
```

The following maven options are recommended:

```
-Xmx1048m -XX:PermSize=64M -XX:MaxPermSize=128M
```

Building the whole Jersey project including tests could take about an hour, depending on your system performance of course. Even if you have a pretty fast performant machine, this could be quite annoying. Especially if you just want to experiment with a limited amount of code. To avoid building the whole Jersey project tree, you can easily utilize the maven reactor plugin.

To build only the modules needed for the helloworld example, you can launch:

```
mvn reactor:make -Dmake.goals=clean,install -Dmake.folders=samples/helloworld
```

which takes less then 2 minutes on my machine. To switch testing off, when building the same set of modules, you will use:

```
mvn reactor:make -Dmake.goals=-Dmaven.test.skip,clean,install -Dmake.folders=samples/helloworld
```

13.3. Testing

Jersey contains many unit tests. Most of these are not really unit tests per-say and are functional tests using the JUnit test framework because it is very convientient for execution and reporting.

Some modules have specific tests but most tests associated with testing the jersey-core, jersey-client and jersey-server modules are located in the jersey-test module. This module can take some time to execute all the tests so it is recommended that you pick the appropriate tests to run related to the particular area that is being investigated. For example, using:

```
mvn -Dtest=<pattern> test
```

where `pattern` may be a comma separated set of names matching tests.

13.4. Continuous building and testing with Hudson

Jersey is built, tested and deployed on Solaris, Windows and Linux using an interal Hudson server. The Jersey Hudson jobs are available publically at http://hudson.glassfish.org/job/Jersey-trunk-

multiplatform/.

13.5. Using NetBeans

NetBeans 6.8 or greater has excellent maven support. The Jersey maven modules can be loaded, built and tested in NetBeans without any additional project-specific requirements.