Jersey 1.1.5.1 User Guide

Jersey 1.1.5.1 User Guide

Table of Contents

List of Examples

Preface

This doc is yet in a very early state. We want to create here sort of a user guide for jersey users, which will get versioned together with jersey code base.

More content to come soon...

Chapter 1. Getting Started

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 5, *Dependencies*

Maven developers require a dependency on

- the jersey-server [http://download.java.net/maven/2/com/sun/jersey/jersey-server/1.1.5.1/jersey-server-1.1.5.1.pom] module,
- the grizzly-servlet-webserver [http://download.java.net/maven/2/com/sun/grizzly/grizzly-servlet-webserver/1.9.8/grizzly-servlet-webserver-1.9.8.pom] module
- and optionally for WADL support if using Java SE 5 the jaxb-impl [http://download.java.net/maven/1/com.sun.xml.bind/poms/jaxb-impl-2.1.12.pom] module

The following dependencies need to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-server</artifactId>
    <version>1.1.5.1</version>
</dependency>
<dependency>
    <groupId>com.sun.grizzly</groupId>
    <artifactId>grizzly-servlet-webserver</artifactId>
    <version>1.9.8</version>
</dependency>
```

And the following repositories need to be added to the pom:

```
<repository>
    <id>maven2-repository.dev.java.net</id>
    <name>Java.net Repository for Maven</name>
    <url>http://download.java.net/maven/2/</url>
    <layout>default</layout>
</repository>
<repository>
    <id>maven-repository.dev.java.net</id>
    <name>Java.net Maven 1 Repository (legacy)</name>
    <url>http://download.java.net/maven/1</url>
    <layout>legacy</layout>
</repository>
```

Non-maven developers require:

- grizzly-servlet-webserver.jar [http://download.java.net/maven/2/com/sun/grizzly/grizzly-servlet-webserver/1.9.8/grizzly-servlet-webserver-1.9.8.jar],
- jersey-server.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-server/1.1.5.1/jersey-server-1.1.5.1.jar],
- jersey-core.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-core/1.1.5.1/jersey-core-1.1.5.1.jar],

- jsr311-api.jar [http://download.java.net/maven/2/javax/ws/rs/jsr311-api/1.1/jsr311-api-1.1.jar],
- asm.jar [http://repo1.maven.org/maven2/asm/asm/3.1/asm-3.1.jar]

and optionally for WADL support if using Java SE 5:

- jaxb-impl.jar [http://download.java.net/maven/1/com.sun.xml.bind/jars/jaxb-impl-2.1.12.jar],
- jaxb-api.jar [http://download.java.net/maven/1/javax.xml.bind/jars/jaxb-api-2.1.jar],
- activation.jar [http://download.java.net/maven/1/javax.activation/jars/activation-1.1.jar],
- stax-api.jar [http://download.java.net/maven/1/javax.xml.stream/jars/stax-api-1.0-2.jar]

For Ant developers the Ant Tasks for Maven [http://maven.apache.org/ant-tasks.html] 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="1.1.5.1"/>
  <dependency groupId="com.sun.grizzly"
              artifactId="grizzly-servlet-webserver"
              version="1.8.6.4"/>
  <artifact:remoteRepository id="maven2-repository.dev.java.net"
                             url="http://download.java.net/maven/2/" />
  <artifact:remoteRepository id="maven-repository.dev.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 the section called “Here's one Paul created earlier”

Creating a root resource

Create the following Java class in your project:

```
    // The Java class will be hosted at the URI path "/helloworld"
    @Path("/helloworld")
    public class HelloWorldResource {

        // The Java method will process HTTP GET requests
        @GET
        // The Java method will produce content identified by the MIME Media
        // type "text/plain"
        @Produces("text/plain")
        public String getClichedMessage() {
            // Return some cliched textual content
            return "Hello World";
        }
    }
```

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.

Deploying the root resource

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

Create the following Java class in your project:

```
public class Main {

     public static void main(String[] args) throws IOException {

         final String baseUri = "http://localhost:9998/";
         final Map<String, String> initParams =
                        new HashMap<String, String>();

         initParams.put("com.sun.jersey.config.property.packages",
                 "com.sun.jersey.samples.helloworld.resources");

        System.out.println("Starting grizzly...");
        SelectorThread threadSelector =
           GrizzlyWebContainerFactory.create(baseUri, initParams);
        System.out.println(String.format(
          "Jersey app started with WADL available at %sapplication.wadl\n" +
          "Try out %shelloworld\nHit enter to stop it...", baseUri, baseUri));
        System.in.read();
        threadSelector.stopEndpoint();
        System.exit(0);
    }
}
```

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

Lines 9 to 10 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).

Lines 13 to 14 deploys the root resource to the base URI "http://localhost:9998/" and returns a Grizzly SelectorThread. 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.

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
```

Here's one Paul created earlier

The example code presented above is shipped as the HelloWorld [http://download.java.net/maven/2/com/sun/jersey/samples/helloworld/1.1.5.1/helloworld-1.1.5.1-project.zip] 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 [http://download.java.net/maven/2/com/sun/jersey/samples/helloworld-webapp/1.1.5.1/helloworld-webapp-1.1.5.1-project.zip] sample.

Chapter 2. Overview of JAX-RS 1.1

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

The JAX-RS 1.1 API may be found online here [https://jsr311.dev.java.net/nonav/releases/1.1/index.html].

The JAX-RS 1.1 specification draft may be found online here [https://jsr311.dev.java.net/nonav/releases/1.1/spec/spec.html].

Root Resource Classes

Root resource classes are POJOs (Plain Old Java Objects) that are annotated with @Path [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Path.html] have at least one method annotated with @Path [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Path.html] or a resource method designator annotation such as @GET [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/GET.html], @PUT [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/PUT.html], @POST [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/POST.html], or @DELETE [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/DELETE.html]. Resource methods are methods of a resource class annotated with a resource method designator. This section shows how to use Jersey to annotate Java objects to create RESTful web services.

The following code example is a very simple example of a root resource class using JAX-RS annotations. The example code shown here is from one of the samples that ships with Jersey, the zip file of which can be found in the maven repository here [http://download.java.net/maven/2/com/sun/jersey/samples/helloworld/1.1.5.1/helloworld-1.1.5.1-project.zip].

Example 2.1. Simple hello world root resource class

```
package com.sun.ws.rest.samples.helloworld.resources;

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

// The Java class will be hosted at the URI path "/helloworld"
@Path("/helloworld")
public class HelloWorldResource {

    // The Java method will process HTTP GET requests
    @GET
    // The Java method will produce content identified by the MIME Media
    // type "text/plain"
    @Produces("text/plain")
    public String getClichedMessage() {
        // Return some cliched textual content
        return "Hello World";
    }
}
```

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

@Path

The @Path [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Path.html] annotation's value is a relative URI path. In the example above, the Java class will be hosted at the URI path `/helloworld`. This is an extremely simple use of the @Path [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Path.html] annotation. What makes JAX-RS so useful is that you can embed variables in the URIs.

URI path templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by curly braces. For example, look at the following @Path [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Path.html] annotation:

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

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

```
http://example.com/users/Galileo
```

To obtain the value of the username variable the @PathParam [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/PathParam.html] may be used on method parameter of a request method, for example:

Example 2.2. Specifying URI path parameter

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

    @GET
    @Produces("text/xml")
    public String getUser(@PathParam("username") String userName) {
        ...
    }
}
```

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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Path.html] value may or may not begin with a '/', it makes no difference. Likewise, by default, a @Path [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Path.html] value may or may not end in a '/', it makes no difference, and thus request URLs that end or do not end in a '/' will both be matched. 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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Path.html] does end in a '/'.

HTTP Methods

@GET [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/GET.html], @PUT [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/PUT.html], @POST [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/POST.html], @DELETE [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/DELETE.html], and @HEAD [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/HEAD.html] are *resource method designator* annotations defined by JAX-RS and which correspond to the similarly named HTTP methods. In the example above, the annotated Java method will process HTTP GET requests. The behavior of a resource is determined by which of the HTTP methods the resource is responding to.

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

Example 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 the Allow response header will be set to the set of HTTP methods support by the resource. In addition Jersey will return a WADL [https://wadl.dev.java.net/] document describing the resource.

@Produces

The @Produces [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Produces.html] annotation is used to specify the MIME media types of representations a resource can produce and send back to the client. In this example, the Java method will produce representations identified by the MIME media type "text/plain".

@Produces [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Produces.html] 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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Produces.html] annotation at the class level. The `doGetAsHtml` method's @Produces [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Produces.html] annotation overrides the class-level @Produces [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Produces.html] setting, and specifies that the method can produce HTML rather than plain text.

If a resource class is capable of producing more that one MIME media type then the resource method chosen will correspond to the most acceptable media type as declared by the client. More specifically the Accept header of the HTTP request 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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Produces.html] 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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/MediaType.html].

@Consumes

The @Consumes [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Consumes.html] 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

```
@POST
@Consumes("text/plain")
public void postClichedMessage(String message) {
    // Store the message
}
```

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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Consumes.html] can be applied at both the class and method levels and more than one media type may be declared in the same @Consumes [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Consumes.html] declaration.

Deploying a RESTful Web Service

JAX-RS provides the deployment agnostic abstract class Application [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Application.html] for declaring root resource and provider classes, and root resource and provider singleton instances. A Web service may extend this class to declare root resource and provider classes. For example,

Example 2.7. Deployment agnostic application model

```
public class MyApplicaton extends Application {
    public Set<Class<?>> getClasses() {
        Set<Class<?>> s = new HashSet<Class<?>>();
        s.add(HelloWorldResource.class);
        return s;
    }
}
```

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

```
public class MyApplication extends PackagesResourceConfig {
    public MyApplication() {
        super("org.foo.rest;org.bar.rest");
    }
}
```

For servlet deployments JAX-RS specifies that a class that implements Application [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Application.html] may be declared instead of a servlet class in `<servlet-class>` element of a `web.xml`, but as of writing this is not currently supported for Jersey. Instead it is necessary to declare the Jersey specific servlet and the Application [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Application.html] class as follows:

Example 2.9. 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</serv
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>MyApplication</param-value>
        </init-param>
    </servlet>
    ....
```

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

Example 2.10. 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</serv
        <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>
    ....
```

JAX-RS also provides the ability to obtain a container specific artifact from an Application [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Application.html] instance. For example, Jersey supports using Grizzly [https://grizzly.dev.java.net/] as follows:

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

Jersey also provides Grizzly [https://grizzly.dev.java.net/] 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.

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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/PathParam.html] to extract a path parameter from the path component of the request URL that matched the path declared in @Path [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Path.html].

@QueryParam [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/QueryParam.html] is used to extract query parameters from the Query component of the request URL. The following example is an extract from the sparklines sample:

Example 2.11. 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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/DefaultValue.html] annotation, will be assigned to the step method parameter. If the "step" value cannot be parsed as an 32 bit signed integer then a 400 (Client Error) response is returned. User defined Java types such as `ColorParam` may be used, which as implemented as follows:

Example 2.12. 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);
                return ((Color)f.get(null)).getRGB();
            } catch (Exception e) {
                throw new WebApplicationException(400);
            }
        }
    }
}
```

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` that accepts a single `String` argument (see, for example, `Integer.valueOf(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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/DefaultValue.html] is not used on conjuction with @QueryParam [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/QueryParam.html] and the query parameter is not present in the request then value will be an empty collection for `List`, `Set` or `SortedSet`, `null` for other object types, and the Java-defined default for primitive types.

The @PathParam [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/PathParam.html] and the other parameter-based annotations, @MatrixParam [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/MatrixParam.html], @HeaderParam [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/HeaderParam.html], @CookieParam [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/CookieParam.html] and @FormParam [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/FormParam.html] obey the same rules as @QueryParam [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/QueryParam.html]. @MatrixParam [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/MatrixParam.html] extracts information from URL path segments. @HeaderParam [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/HeaderParam.html] extracts information from the HTTP headers. @CookieParam [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/CookieParam.html] extracts information from the cookies declared in cookie related HTTP headers.

@FormParam [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/FormParam.html] is slightly special because it extracts information from a request representation that is of the MIME media type "application/x-www-form-urlencoded" and conforms to the encoding specified by HTML forms, as described here. This parameter is very useful for extracting information that is POSTed by HTML forms, for example the following extracts the form parameter named "name" from the POSTed form data:

Example 2.13. Processing POSTed HTML form

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

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

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

For header and cookie parameters the following:

Example 2.15. Obtaining general map of header parameters

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

In general @Context [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Context.html] 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 2.16. Obtaining general map of form parameters

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

Represetations and Java Types

Previous sections on @Produces [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Produces.html] and @Consumes [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Consumes.html] 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 [http://java.sun.com/javase/6/docs/api/javax/xml/bind/annotation/XmlRootElement.html] or @XmlType [http://java.sun.com/javase/6/docs/api/javax/xml/bind/annotation/XmlType.html]. 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 2.17. Using `File` with a specific MIME type to produce a response

```
@GET
@Path("/images/{image}")
@Produces("image/*")
public Response getImage(@PathParam("image") String image) {
    File f = new File(image);

    if (!f.exists()) {
        throw new WebApplicationException(404);
    }

    String mt = new MimetypesFileTypeMap().getContentType(f);
    return Response.ok(f, mt).build();
}
```

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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Produces.html] if the most acceptable media type is not a wild card (one that contains a *, for example "application/" or "/*"). Given the following method:

Example 2.18. The most acceptable MIME type is used when multiple output MIME types allowed

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

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

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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Response.html] and Response.ResponseBuilder [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Response.ResponseBuilder.html]. For example, a common RESTful pattern for the creation of a new resource is to support a POST request that returns a 201 (Created) status code and a `Location` header whose value is the URI to the newly created resource. This may be acheived as follows:

Example 2.19. Returning 201 status code and adding `Location` header in response to POST request

```
@POST
@Consumes("application/xml")
public Response post(String content) {
    URI createdUri = ...
    create(content);
    return Response.created(createdUri).build();
}
```

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

Example 2.20. Adding an entity body to a custom response

```
@POST
@Consumes("application/xml")
public Response post(String content) {
    URI createdUri = ...
    String createdContent = create(content);
    return Response.created(createdUri).entity(createdContent).build();
}
```

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

Sub-resources

@Path [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Path.html] may be used on classes and such classes are referred to as root resource classes. @Path [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Path.html] may also be used on methods of root resource classes. This enables common functionality for a number of resources to be grouped together and potentially reused.

The first way @Path [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Path.html] may be used is on resource methods and such methods are referred to as *sub-resource methods*. The following example shows the method signatures for a root resource class from the jmaki-backend sample:

Example 2.21. Sub-resource methods

```
@Singleton
@Path("/printers")
public class PrintersResource {

    @GET
    @Produces({"application/json", "application/xml"})
    public WebResourceList getMyResources() { ... }

    @GET @Path("/list")
    @Produces({"application/json", "application/xml"})
    public WebResourceList getListOfPrinters() { ... }

    @GET @Path("/jMakiTable")
    @Produces("application/json")
    public PrinterTableModel getTable() { ... }

    @GET @Path("/jMakiTree")
    @Produces("application/json")
    public TreeModel getTree() { ... }

    @GET @Path("/ids/{printerid}")
    @Produces({"application/json", "application/xml"})
    public Printer getPrinter(@PathParam("printerid") String printerId) { ... }

    @PUT @Path("/ids/{printerid}")
    @Consumes({"application/json", "application/xml"})
    public void putPrinter(@PathParam("printerid") String printerId,  Printer prin

    @DELETE @Path("/ids/{printerid}")
    public void deletePrinter(@PathParam("printerid") String printerId) { ... }
}
```

If the path of the request URL is "printers" then the resource methods not annotated with @Path [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Path.html] will be selected. If the request path of the request URL is "printers/list" then first the root resource class will be matched and then the sub-resource methods that match "list" will be selected, which in this case is the sub-resource method `getListOfPrinters`. So in this example hierarchical matching on the path of the request URL is performed.

The second way @Path [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/Path.html] may be used is on methods **not** annotated with resource method designators such as @GET [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/GET.html] or @POST [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/POST.html]. Such methods are referred to as *sub-resource locators*. The following example shows the method signatures for a root resource class and a resource class from the optimistic-concurrency sample:

Example 2.22. Sub-resource locators

```
@Path("/item")
public class ItemResource {
    @Context UriInfo uriInfo;

    @Path("content")
    public ItemContentResource getItemContentResource() {
        return new ItemContentResource();
    }

    @GET
    @Produces("application/xml")
    public Item get() { ... }
}

public class ItemContentResource {

    @GET
    public Response get() { ... }

    @PUT
    @Path("{version}")
    public void put(
            @PathParam("version") int version,
            @Context HttpHeaders headers,
            byte[] in) { ... }
}
```

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.

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 [http://java.sun.com/j2se/1.5.0/docs/api/java/net/URI.html], which is why JAX-RS has the UriBuilder [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/UriBuilder.html] class that makes it simple and easy to build URIs safely.

UriBuilder [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/UriBuilder.html] 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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/UriInfo.html] provides such information (in addition to further information, see next section).

The following example shows URI building with UriInfo and UriBuilder [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/UriBuilder.html] from the bookmark sample:

Example 2.23. URI building

```
@Path("/users/")
public class UsersResource {

    @Context UriInfo uriInfo;

    ...

    @GET
    @Produces("application/json")
    public JSONArray getUsersAsJsonArray() {
        JSONArray uriArray = new JSONArray();
        for (UserEntity userEntity : getUsers()) {
            UriBuilder ub = uriInfo.getAbsolutePathBuilder();
            URI userUri = ub.
                    path(userEntity.getUserid()).
                    build();
            uriArray.put(userUri.toASCIIString());
        }
        return uriArray;
    }
}
```

UriInfo [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/UriInfo.html] 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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/UriInfo.html] can be used to obtain URIs and associated UriBuilder [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/UriBuilder.html] 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() [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/UriInfo.html#getAbsolutePathBuilder()]. A new path segment is added, which is the user ID, and then the URI is built. Notice that it is not necessary to worry about the inclusion of '/' characters or that the user ID may contain characters that need to be percent encoded. UriBuilder takes care of such details.

UriBuilder [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/UriBuilder.html] 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 2.24. Building URIs using query parameters

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

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 2.25. 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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/WebApplicationException.html] and builds a HTTP response with the 404 status code and an optional message as the body of the response:

Example 2.26. 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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/WebApplicationException.html], or classes that extend WebApplicationException [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/WebApplicationException.html], and instead it may be preferable to map an existing exception to a response. For such cases it is possible to use the ExceptionMapper [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/ext/ExceptionMapper.html] interface. For example, the following maps the EntityNotFoundException [http://java.sun.com/javaee/5/docs/api/javax/persistence/EntityNotFoundException.html] to a 404 (Not Found) response:

Example 2.27. 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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/ext/Provider.html], this declares that the class is of interest to the JAX-RS runtime. Such a class may be added to the set of classes of the Application [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Application.html] instance that is configured. When an application throws an EntityNotFoundException [http://java.sun.com/javaee/5/docs/api/javax/persistence/EntityNotFoundException.html] the `toResponse` method of the `EntityNotFoundMapper` instance will be invoked.

Conditional GETs and Returning 304 (Not Modified) Responses

Conditional GETs are a great way to reduce bandwidth, and potentially server-side peformance, 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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Request.html].

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

Example 2.28. 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;

    if (!limits.contains(data))
        throw new WebApplicationException(400);

    this.tag = computeEntityTag(ui.getRequestUri());
    if (request.getMethod().equals("GET")) {
        Response.ResponseBuilder rb = request.evaluatePreconditions(tag);
        if (rb != null)
            throw new WebApplicationException(rb.build());
    }
}
```

The constructor of the `SparklinesResouce` root resource class computes an entity tag from the request URI and then calls the request.evaluatePreconditions [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Request.html#evaluatePreconditions(javax.ws.rs.core.EntityTag)] with that entity tag. If a client request contains an `If-None-Match` header with a value that contains the same entity tag that was calculated then the evaluatePreconditions [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Request.html#evaluatePreconditions(javax.ws.rs.core.EntityTag)] returns a pre-filled out response, with the 304 status code and entity tag set, that may be built and returned. Otherwise, evaluatePreconditions [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Request.html#evaluatePreconditions(javax.ws.rs.core.EntityTag)] returns `null` and the normal response can be returned.

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

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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Application.html].

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 applcation. If a root resource class is annotated with @PerSession then one instance is created per web session and stored as a session attribute.

Security

Security information is available by obtaining the SecurityContext [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/SecurityContext.html] using @Context [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Context.html], which is essentially the equivalent functionality available on the HttpServletRequest [http://java.sun.com/javaee/5/docs/api/javax/servlet/http/HttpServletRequest.html].

SecurityContext [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/SecurityContext.html] 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 2.29. Accessing `SecurityContext`

```
@Path("basket")
public ShoppingBasketResource get(@Context SecurityContext sc) {
    if (sc.isUserInRole("PreferredCustomer") {
       return new PreferredCustomerShoppingBaskestResource();
    } else {
       return new ShoppingBasketResource();
    }
}
```

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 2.30. Injection

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

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

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

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

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

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

There are some restrictions when injecting on to resource classes with a life-cycle 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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Context.html] annotation on fields, in such cases a thread local proxy will be injected.

The @FormParam [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/FormParam.html] annotation is special and may only be utilized on resource and sub-resource methods. This is because it extracts information from a request entity.

Use of @Context

Previous sections have introduced the use of @Context [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Context.html]. Chapter 5 [https://jsr311.dev.java.net/nonav/releases/1.1/spec/spec3.html#x3-520005] of the JAX-RS specification presents all the standard JAX-RS Java types that may be used with @Context [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Context.html].

When deploying a JAX-RS application using servlet then ServletConfig [http://java.sun.com/javaee/5/docs/api/javax/servlet/ServletConfig.html], ServletContext [http://java.sun.com/javaee/5/docs/api/javax/servlet/ServletContext.html], HttpServletRequest [http://java.sun.com/javaee/5/docs/api/javax/servlet/http/HttpServletRequest.html] and HttpServletResponse [http://java.sun.com/javaee/5/docs/api/javax/servlet/http/HttpServletResponse.html] are available using @Context [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/core/Context.html].

Annotations Defined By JAX-RS

For a list of the annotations specified by JAX-RS see Appendix A [https://jsr311.dev.java.net/nonav/releases/1.1/spec/spec3.html#x3-66000A] of the specification.

Chapter 3. Client API

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.

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 [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/WebResource.html], 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.

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 [https://jsr311.dev.java.net/nonav/releases/1.0/javax/ws/rs/core/UriBuilder.html] and UriTemplate [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/uri/UriTemplate.html] 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 [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/json/package-summary.html] support and MIME Multipart [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/contribs/jersey-multipart/index.html] support.

3. Using the builder pattern to make it easier to construct requests.

Some APIs, like the Apache HTTP client or java.net.HttpURLConnection [http://java.sun.com/j2se/1.5.0/docs/api/java/net/HttpURLConnection.html], 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 [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/representation/Form.html] is created with two parameters, a new WebResource [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/WebResource.html] instance is created from a Client [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/Client.html] 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.

Getting started with the Jersey client

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

Refer to the Java API documentation [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/package-summary.html] for details on the Jersey client API packages and classes.

Refer to the Java API Apache HTTP client documentation [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/contribs/jersey-apache-client/index.html] for details on how to use the Jersey client API with the Apache HTTP client.

Overview of the API

To utilize the client API it is first necessary to create an instance of a Client [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/Client.html], for example:

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

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 [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/config/ClientConfig.html] 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 [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/WebResource.html] 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.

Building a request

Requests to a Web resource are built using the builder pattern (see RequestBuilder [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/RequestBuilder.html]) where the terminating method corresponds to an HTTP method (see UniformInterface [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/UniformInterface.html]). 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);
```

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 [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/ClientResponse.html] 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 [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/UniformInterfaceException.html] 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();
    }
```

Creating new WebResources from a WebResource

A new WebResource [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/WebResource.html] 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”.

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 [https://jsr311.dev.java.net/nonav/releases/1.0/spec/index.html] 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 [http://download.java.net/maven/2/com/sun/jersey/samples/jaxb/1.1.5.1/jaxb-1.1.5.1-project.zip] to see how JAXB with XML and JSON can be utilized with the client API (more specifically, see the unit tests).

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 [https://jsr311.dev.java.net/nonav/javadoc/javax/ws/rs/ext/MessageBodyReader.html] and MessageBodyWriter [https://jsr311.dev.java.net/nonav/javadoc/javax/ws/rs/ext/MessageBodyWriter.html], respectively, for request and response entities (or inbound and outbound representations). Refer to the entity provider [http://download.java.net/maven/2/com/sun/jersey/samples/entity-provider/1.1.5.1/entity-provider-1.1.5.1-project.zip] 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);
```

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 [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/filter/Filterable.html] and that enables the addition of ClientFilter [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/filter/ClientFilter.html] 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 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 responses, filters are applied in reverse order, starting with the `WebResource` filters and then moving to the inherited filters. 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 `filter1`, `filter2` and `filter3` in that order. After the response has been received the response is filtered by `filter3`, `filter2` and `filter1` 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 [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/ClientRequest.html] 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.

Supported filters

The Jersey Client API currently supports two filters:

1. A GZIP content encoding filter, GZIPContentEncodingFilter [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/filter/GZIPContentEncodingFilter.html]. 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 [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/client/filter/LoggingFilter.html]. 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 [http://download.java.net/maven/2/com/sun/jersey/jersey-client/1.1.5.1/jersey-client-1.1.5.1-sources.jar] of the Jersey client for more details.

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 [http://download.java.net/maven/2/com/sun/jersey/samples/helloworld-webapp/1.1.5.1/helloworld-webapp-1.1.5.1-project.zip] 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`.

Security with Http(s)URLConnection

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 [http://java.sun.com/javase/6/docs/technotes/guides/net/http-auth.html] document for more details. For cookie management the `java.net.CookieHandler` can be extended and statically registered. Refer to the Cookie Management [http://java.sun.com/javase/6/docs/technotes/guides/net/http-cookie.html] 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 [https://jersey.dev.java.net/nonav/apidocs/latest/jersey/com/sun/jersey/client/urlconnection/HTTPSProperties.html] for how to set `javax.net.ssl.HostnameVerifier` and `javax.net.ssl.SSLContext`.

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 [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/contribs/jersey-apache-client/com/sun/jersey/client/apache/package-summary.html], ApacheHttpClientState [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/contribs/jersey-apache-client/com/sun/jersey/client/apache/config/ApacheHttpClientState.html] and ApacheHttpClientConfig [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/contribs/jersey-apache-client/com/sun/jersey/client/apache/config/ApacheHttpClientConfig.html] for more details.

Chapter 4. JSON Support

Jersey JSON support comes as a set of JAX-RS MessageBodyReader [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/ext/MessageBodyReader.html] and MessageBodyWriter [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/ext/MessageBodyWriter.html] providers distributed with *jersey-json* module. These providers enable using two basic approaches when working with JSON format:

- JAXB based JSON support
- Low-level, JSONObject/JSONArray based JSON support

Both approaches use the same principle. You need to make your resource methods consume and/or produce instances of certain Java types, known to provided MessageBodyReaders/Writers. Also, naturally, you need to make sure you use *jersey-json* module with your application.

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 4.1. 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 4.2. 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"}
```

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 4.3. 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 [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/json/JSONConfiguration.html] instance. The instance could be then further used to create a JSONConfigurated JSONJAXBContext [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/json/JSONJAXBContext.html], 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 [https://jsr311.dev.java.net/nonav/releases/1.1/javax/ws/rs/ext/ContextResolver.html]

Example 4.4. 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(
     JSONConfiguration.natural().build(), types);
    }

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

Creation of our specialized JAXBContext
Final JSON format is given by this JSONConfiguration instance

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 4.5. 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 4.6. 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 4.7. 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 https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/json/JSONConfiguration.html

Mapped notation

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

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

for usage in a `JAXBContext` resolver, Example 4.4, “An example of a JAXBContext resolver implementation”. Then a contact bean initialized with Example 4.7, “JAXB beans for JSON supported notations description, initialization”, will be serialized as

Example 4.8. 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 4.9. 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 4.10. 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 4.11. 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 4.12. 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 4.13. XML root tag equivalent kept in JSON using `mapped` notation

```
{"contact":{ "id":"2"
 ,"name":"Bob"
 ,"addresses":{"street":"Long Street 1"
                   ,"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 4.18, “XML namespace to JSON mapping configuration for Jettison based `mapped` notation”

Example 4.14. XML namespace to JSON mapping configuration for `mapped` notation

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

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 4.7, “JAXB beans for JSON supported notations description, initialization”:

Example 4.15. JSON expression produced using `natural` notation

```
{ "id":2
 ,"name":"Bob"
 ,"addresses":[{"street":"Long Street 1"
                     ,"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 4.12, “Keep XML root tag equivalent in JSON `mapped` JSON notation”).

Example 4.16. Keep XML root tag equivalent in JSON `natural` JSON notation

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

Jettison mapped notation

Next two notations are based on project Jettison [http://jettison.codehaus.org/User%27s+Guide]. 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 4.17. JSON expression produced using Jettison based `mapped` notation

```
{ "contact:{"id":2
              ,"name":"Bob"
              ,"addresses":{"street":"Long Street 1"
                                  ,"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 4.18. XML namespace to JSON mapping configuration for Jettison based `mapped` notation

```
      Map<String,String> ns2json = new HashMap<String, String>();
      ns2json.put("http://example.com", "example");
      context = new JSONJAXBContext(
     JSONConfiguration.mappedJettison()
          .xml2JsonNs(ns2json).build(), types);
```

Resulting JSON will look like in the example bellow.

Example 4.19. JSON expression with XML namespaces mapped into JSON

```
{ "contact:{"example.id":2
              ,"name":"Bob"
              ,"addresses":{"street":"Long Street 1"
                                   ,"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

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 4.7, “JAXB beans for JSON supported notations description, initialization” will be as follows.

Example 4.20. JSON expression produced using `badgerfish` notation

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

Examples

Download http://download.java.net/maven/2/com/sun/jersey/samples/json-from-jaxb/1.1.5.1/json-from-jaxb-1.1.5.1-project.zip or http://download.java.net/maven/2/com/sun/jersey/samples/jmaki-backend/1.1.5.1/jmaki-backend-1.1.5.1-project.zip to get a more complex example using JAXB based JSON support.

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 [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 4.21. 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 4.22. Constructing a JSONObject

```
JSONObject myObject = new JSONObject();
myObject.JSONObject myObject = new JSONObject();
try {
  myObject.put("name", "Agamemnon");
  myObject.put("age", 32);
} catch (JSONException ex) {
  LOGGER.log(Level.SEVERE, "Error ...", ex);
}
```

Examples

Download http://download.java.net/maven/2/com/sun/jersey/samples/bookmark/1.1.5.1/bookmark-1.1.5.1-project.zip to get a more complex example using low-level JSON support.

Chapter 5. Dependencies

Jersey is built, assembled and installed using Maven. Jersey is deployed to the Java.Net maven repository at the following location: http://download.java.net/maven/2/ [http://download.java.net/maven/2/com/sun/jersey]. The Jersey modules can be browsed at the following location: http://download.java.net/maven/2/com/sun/jersey/ [http://download.java.net/maven/2/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 [http://download.java.net/maven/2/com/sun/jersey/samples/jersey-samples/1.1.5.1/jersey-samples-1.1.5.1-project.zip]. Individual zip files for each sample may be found by browsing the samples [http://download.java.net/maven/2/com/sun/jersey/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.

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 [http://maven.apache.org/ant-tasks.html] very useful. For the convenience of non-maven developers the following are provided:

- A zip of Jersey [http://download.java.net/maven/2/com/sun/jersey/jersey-archive/1.1.5.1/jersey-archive-1.1.5.1.zip] containing the Jersey jars, core dependencies (it does not provide dependencies for third party jars beyond the those for JSON support) and JavaDoc.
- A jersey bundle jar [http://download.java.net/maven/2/com/sun/jersey/jersey-bundle/1.1.5.1/jersey-bundle-1.1.5.1.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

Core server

Maven developers require a dependency on the jersey-server [http://download.java.net/maven/2/com/sun/jersey/jersey-server/1.1.5.1/jersey-server-1.1.5.1.pom] module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-server</artifactId>
    <version>1.1.5.1</version>
</dependency>
```

And the following repositories need to be added to the pom:

```
<repository>
    <id>maven2-repository.dev.java.net</id>
    <name>Java.net Repository for Maven</name>
    <url>http://download.java.net/maven/2/</url>
    <layout>default</layout>
</repository>
<repository>
    <id>maven-repository.dev.java.net</id>
    <name>Java.net Maven 1 Repository (legacy)</name>
    <url>http://download.java.net/maven/1</url>
    <layout>legacy</layout>
</repository>
```

Non-maven developers require:

- jersey-server.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-server/1.1.5.1/jersey-server-1.1.5.1.jar],

- jersey-core.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-core/1.1.5.1/jersey-core-1.1.5.1.jar],

- jsr311-api.jar [http://download.java.net/maven/2/javax/ws/rs/jsr311-api/1.1/jsr311-api-1.1.jar],

- asm.jar [http://repo1.maven.org/maven2/asm/asm/3.1/asm-3.1.jar]

or, if using the jersey-bundle:

- jersey-bundle.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-bundle/1.1.5.1/jersey-bundle-1.1.5.1.jar],

- jsr311-api.jar [http://download.java.net/maven/2/javax/ws/rs/jsr311-api/1.1/jsr311-api-1.1.jar],

- asm.jar [http://repo1.maven.org/maven2/asm/asm/3.1/asm-3.1.jar]

For Ant developers the Ant Tasks for Maven [http://maven.apache.org/ant-tasks.html] 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="1.1.5.1"/>
  <artifact:remoteRepository id="maven2-repository.dev.java.net"
                             url="http://download.java.net/maven/2/" />
  <artifact:remoteRepository id="maven-repository.dev.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 [http://repo1.maven.org/maven2/asm/asm/3.1/asm-3.1.jar] dependency is required when either of the following com.sun.jersey.api.core.ResourceConfig [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/core/ResourceConfig.html] implementations are utilized:

- com.sun.jersey.api.core.ClasspathResourceConfig [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/core/ClasspathResourceConfig.html]; or
- com.sun.jersey.api.core.PackagesResourceConfig [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/core/PackagesResourceConfig.html]

By default Jersey will utilize the ClasspathResourceConfig [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/api/core/ClasspathResourceConfig.html] 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.

Core client

Maven developers require a dependency on the jersey-client [http://download.java.net/maven/2/com/sun/jersey/jersey-client/1.1.5.1/jersey-client-1.1.5.1.pom] module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-client</artifactId>
    <version>1.1.5.1</version>
</dependency>
```

Non-maven developers require:

- jersey-client.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-client/1.1.5.1/jersey-client-1.1.5.1.jar],
- jersey-core.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-core/1.1.5.1/jersey-core-1.1.5.1.jar],
- jsr311-api.jar [http://download.java.net/maven/2/javax/ws/rs/jsr311-api/1.1/jsr311-api-1.1.jar]

or, if using the jersey-bundle:

- jersey-bundle.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-bundle/1.1.5.1/jersey-bundle-1.1.5.1.jar],
- jsr311-api.jar [http://download.java.net/maven/2/javax/ws/rs/jsr311-api/1.1/jsr311-api-1.1.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 [http://download.java.net/maven/2/com/sun/jersey/contribs/jersey-apache-client/1.1.5.1/jersey-apache-client-1.1.5.1.pom] module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey.contribs</groupId>
    <artifactId>jersey-apache-client</artifactId>
    <version>1.1.5.1</version>
</dependency>
```

Container

Grizzly HTTP Web server

Maven developers, deploying an application using the Grizzly HTTP Web server, require a dependency on the grizzly-servlet-webserver [http://download.java.net/maven/2/com/sun/grizzly/grizzly-servlet-webserver/1.9.8/grizzly-servlet-webserver-1.9.8.pom] module.

Non-maven developers require: grizzly-servlet-webserver.jar [http://download.java.net/maven/2/com/sun/grizzly/grizzly-servlet-webserver/1.9.8/grizzly-servlet-webserver-1.9.8.jar]

Simple HTTP Web server

Maven developers, deploying an application using the Simple HTTP Web server, require a dependency on the jersey-simple [http://download.java.net/maven/2/com/sun/jersey/contribs/jersey-simple/1.1.5.1/jersey-simple-1.1.5.1.pom] module.

Light weight HTTP server

Maven developers, using Java SE 5 and deploying an application using the light weight HTTP server, require a dependency on the http [http://download.java.net/maven/2/com/sun/net/httpserver/http/20070405/http-20070405.pom] module.

Non-maven developers require: http.jar [http://download.java.net/maven/2/com/sun/net/httpserver/http/20070405/http-20070405.jar]

Deploying an application using the light weight HTTP server with Java SE 6 requires no additional dependences.

Servlet

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

See the Java documentation here [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey/com/sun/jersey/spi/container/servlet/package-summary.html] on how to configure the servlet container.

Using servlet: `com.sun.jersey.spi.container.servlet.ServletContainer` requires no additional dependences.

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 [http://download.java.net/maven/1/javax.persistence/poms/persistence-api-1.0.2.pom] module.

Non-Maven developers require: persistence-api.jar [http://download.java.net/maven/1/javax.persistence/jars/persistence-api-1.0.2.jar]

Entity

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 using JAXB with Java SE 6 requires no additional dependences, since Java SE 6 ships with JAXB 2.x support.

Maven developers, using Java SE 5, require a dependency on the jaxb-impl [http://download.java.net/maven/1/com.sun.xml.bind/poms/jaxb-impl-2.1.12.pom] module.

Non-maven developers require:

- jaxb-impl.jar [http://download.java.net/maven/1/com.sun.xml.bind/jars/jaxb-impl-2.1.12.jar],
- jaxb-api.jar [http://download.java.net/maven/1/javax.xml.bind/jars/jaxb-api-2.1.jar],
- activation.jar [http://download.java.net/maven/1/javax.activation/jars/activation-1.1.jar],
- stax-api.jar [http://download.java.net/maven/1/javax.xml.stream/jars/stax-api-1.0-2.jar]

Maven developers, using Java SE 5, that are consuming or producing `T[]`, `List<T>` or `Collection<T>` where `T` is a JAXB bean require a dependency on a StAX implementation, such as Woodstox version 3.2.1 or greater using the following dependency:

```
<dependency>
    <groupId>woodstox</groupId>
    <artifactId>wstx-asl</artifactId>
    <version>3.2.1</version>
</dependency>
```

Non-maven developers require: wstx-asl-3.2.1.jar [http://repo1.maven.org/maven2/woodstox/wstx-asl/3.2.1/wstx-asl-3.2.1.jar]

Maven developers, using JSON serialization support of JAXB beans when using the MIME media type `application/json` require a dependency on the jersey-json [http://download.java.net/maven/2/com/sun/jersey/jersey-json/1.1.5.1/jersey-json-1.1.5.1.pom] 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>1.1.5.1</version>
</dependency>
```

Non-maven developers require:

- jackson-core-asl.jar [http://repository.codehaus.org/org/codehaus/jackson/jackson-core-asl/1.1.1/jackson-core-asl-1.1.1.jar],
- jettison.jar [http://repo1.maven.org/maven2/org/codehaus/jettison/jettison/1.1/jettison-1.1.jar],
- jaxb-impl.jar [http://download.java.net/maven/1/com.sun.xml.bind/jars/jaxb-impl-2.1.12.jar],
- jaxb-api.jar [http://download.java.net/maven/1/javax.xml.bind/jars/jaxb-api-2.1.jar],
- activation.jar [http://download.java.net/maven/1/javax.activation/jars/activation-1.1.jar],
- stax-api.jar [http://download.java.net/maven/1/javax.xml.stream/jars/stax-api-1.0-2.jar]

and additionally, if not depending on the jersey-bundle.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-bundle/1.1.5.1/jersey-bundle-1.1.5.1.jar], non-maven developers require: jersey-json.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-json/1.1.5.1/jersey-json-1.1.5.1.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 [http://download.java.net/maven/2/com/sun/jersey/jersey-fastinfoset/1.1.5.1/jersey-fastinfoset-1.1.5.1.pom] 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>1.1.5.1</version>
</dependency>
```

Non-maven developers require:

- FastInfoset.jar [http://download.java.net/maven/1/com.sun.xml.fastinfoset/jars/FastInfoset-1.2.2.jar],
- jaxb-impl.jar [http://download.java.net/maven/1/com.sun.xml.bind/jars/jaxb-impl-2.1.12.jar],
- jaxb-api.jar [http://download.java.net/maven/1/javax.xml.bind/jars/jaxb-api-2.1.jar],
- activation.jar [http://download.java.net/maven/1/javax.activation/jars/activation-1.1.jar],
- stax-api.jar [http://download.java.net/maven/1/javax.xml.stream/jars/stax-api-1.0-2.jar]

and additionally, if not depending on the jersey-bundle.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-bundle/1.1.5.1/jersey-bundle-1.1.5.1.jar], non-maven developers require: jersey-fastinfoset.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-fastinfoset/1.1.5.1/jersey-fastinfoset-1.1.5.1.jar]

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 [http://download.java.net/maven/2/com/sun/jersey/contribs/jersey-atom-abdera/1.1.5.1/jersey-atom-abdera-1.1.5.1.pom] module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey.contribs</groupId>
    <artifactId>jersey-atom-abdera</artifactId>
    <version>1.1.5.1</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 [http://download.java.net/maven/2/com/sun/jersey/jersey-atom/1.1.5.1/jersey-atom-1.1.5.1.pom] module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-atom</artifactId>
    <version>1.1.5.1</version>
</dependency>
```

Non-maven developers require:

- rome.jar [http://download.java.net/maven/1/rome/jars/rome-0.9.jar],
- jdom.jar [http://repo1.maven.org/maven2/jdom/jdom/1.0/jdom-1.0.jar]

and additionally, if not depending on the jersey-bundle.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-bundle/1.1.5.1/jersey-bundle-1.1.5.1.jar], non-maven developers require: jersey-atom.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-atom/1.1.5.1/jersey-atom-1.1.5.1.jar]

JSON

The use of the Java types `org.codehaus.jettison.json.JSONObject` and `org.codehaus.jettison.json.JSONArray` requires Jettison version 1.0 or higher.

Maven developers require a dependency on the jersey-json [http://download.java.net/maven/2/com/sun/jersey/jersey-json/1.1.5.1/jersey-json-1.1.5.1.pom] module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-json</artifactId>
    <version>1.1.5.1</version>
</dependency>
```

Non-maven developers require: jettison.jar [http://repo1.maven.org/maven2/org/codehaus/jettison/jettison/1.1/jettison-1.1.jar] and additionally, if not depending on the jersey-bundle.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-bundle/1.1.5.1/jersey-bundle-1.1.5.1.jar], non-maven developers require: jersey-json.jar [http://download.java.net/maven/2/com/sun/jersey/jersey-json/1.1.5.1/jersey-json-1.1.5.1.jar]

Mail and MIME multipart

The use of the Java type `javax.mail.internet.MimeMultipart` with Java SE 5 or 6 requires Java Mail version 1.4 or higher.

Maven developers require a dependency on the java-mail [http://download.java.net/maven/1/javax.mail/poms/mail-1.4.pom] module.

Non-maven developers require:

- mail.jar [http://download.java.net/maven/1/javax.mail/jars/mail-1.4.jar],
- activation.jar [http://download.java.net/maven/1/javax.activation/jars/activation-1.1.jar]

The use of the Java type `javax.mail.internet.MimeMultipart` with Java EE 5 requires no additional dependencies.

Jersey ships with a high-level MIME multipart API. Maven developers requires a dependency on the jersey-multipart [http://download.java.net/maven/2/com/sun/jersey/contribs/jersey-multipart/1.1.5.1/jersey-multipart-1.1.5.1.pom] module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey.contribs</groupId>
    <artifactId>jersey-multipart</artifactId>
    <version>1.1.5.1</version>
</dependency>
```

Non-maven developers require:

- mimepull.jar [http://download.java.net/maven/2/org/jvnet/mimepull/1.3/mimepull-1.3.jar],
- jersey-multipart.jar [http://download.java.net/maven/2/com/sun/jersey/contribs/jersey-multipart/1.1.5.1/jersey-multipart-1.1.5.1.jar]

Activation

The use of the Java type `javax.activation.DataSource` with Java SE 5 requires Java Activation 1.1 or higher.

Maven developers require a dependency on the activation [http://download.java.net/maven/1/javax.activation/poms/activation-1.1.pom] module.

Non-maven developers require: activation.jar [http://download.java.net/maven/1/javax.activation/jars/activation-1.1.jar]

The use of the Java type `javax.activation.DataSource` with Java SE 6 and Java EE 5 requires no additional dependencies.

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 with Java SE 6 requires no additional dependences, since Java SE 6 ships with JAXB 2.x support.

Maven developers, using Java SE 5, require a dependency on the jaxb-impl [http://download.java.net/maven/1/com.sun.xml.bind/poms/jaxb-impl-2.1.12.pom] module.

Non-maven developers require:

- jaxb-impl.jar [http://download.java.net/maven/1/com.sun.xml.bind/jars/jaxb-impl-2.1.12.jar],
- jaxb-api.jar [http://download.java.net/maven/1/javax.xml.bind/jars/jaxb-api-2.1.jar],
- activation.jar [http://download.java.net/maven/1/javax.activation/jars/activation-1.1.jar],
- stax-api.jar [http://download.java.net/maven/1/javax.xml.stream/jars/stax-api-1.0-2.jar]

If the above dependencies are not present then WADL generation is disabled and a warning will be logged.

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

Spring

Maven developers, using Spring 2.0.x or Spring 2.5.x, require a dependency on the jersey-spring [http://download.java.net/maven/2/com/sun/jersey/contribs/jersey-spring/1.1.5.1/jersey-spring-1.1.5.1.pom] module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey.contribs</groupId>
    <artifactId>jersey-spring</artifactId>
    <version>1.1.5.1</version>
</dependency>
```

See the Java documentation here [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/contribs/jersey-spring/com/sun/jersey/spi/spring/container/servlet/package-summary.html] on how to integrate Jersey-based Web applications with Spring.

Guice

Maven developers, using Guice 2.0, require a dependency on the jersey-guice [http://download.java.net/maven/2/com/sun/jersey/contribs/jersey-guice/1.1.5.1/jersey-guice-1.1.5.1.pom] module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey.contribs</groupId>
    <artifactId>jersey-guice</artifactId>
    <version>1.1.5.1</version>
</dependency>
```

See the Java documentation here [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/contribs/jersey-guice/com/sun/jersey/guice/spi/container/servlet/package-summary.html] on how to integrate Jersey-based Web applications with Guice.

Guice support depends on the Guice artifacts distributed with GuiceyFruit [http://code.google.com/p/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 add the following dependency and repository to the pom:

```
<dependency>
    <groupId>org.guiceyfruit</groupId>
    <artifactId>guiceyfruit</artifactId>
    <version>2.0-beta-6</version>
</dependency>
...
<repository>
    <id>guice-maven</id>
    <name>guice maven</name>
    <url>http://guiceyfruit.googlecode.com/svn/repo/releases</url>
</repository>
```

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 [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey-test-framework/com/sun/jersey/test/framework/JerseyTest.html] class.

Maven developers require a dependency on the jersey-test-framework [http://download.java.net/maven/2/com/sun/jersey/jersey-test-framework/1.1.5.1/jersey-test-framework-1.1.5.1.pom] module. The following dependency needs to be added to the pom:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-test-framework</artifactId>
    <version>1.1.5.1</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.InMemoryTestContainerFactor` 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.GrizzlyWebTestContainerF` for testing with Web-based Grizzly.
- `com.sun.jersey.test.framework.spi.container.embedded.glassfish.EmbeddedGlassFish` for testing with embedded GlassFish v3
- `com.sun.jersey.test.framework.spi.container.external.ExternalTestContainerFactor` for testing with application deployed externally, for example to GlassFish or Tomcat.

The system property `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 \ -Dtest.containerFactory=com.sun.jersey.test.framework.spi.container.ext
```

`-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 [https://jersey.dev.java.net/nonav/apidocs/1.1.5.1/jersey-test-framework/com/sun/jersey/test/framework/package-summary.html] for an example.

See the blog entry on Jersey Test Framework [http://blogs.sun.com/naresh/entry/jersey_test_framework_makes_it] for detailed instructions on how to use 1.1.1-ea version of the framework in your application.