



## REST with Java (JAX-RS) using Jersey - Tutorial

Lars Vogel

Version 2.4

Copyright © 2009, 2010, 2011, 2012, 2013, 2014 vogella GmbH

20.08.2014

### RESTful web services with Java (Jersey / JAX-RS)

This tutorial explains how to develop RESTful web services in Java with the JAX-RS reference implementation Jersey.

In this tutorial Eclipse 4.4 (Luna), Java 1.6, Tomcat 6.0 and JAX-RS 2.0 (with Jersey 2.11) is used.

### Table of Contents

#### 1. REST - Representational State Transfer

- 1.1. What is REST?
- 1.2. HTTP methods
- 1.3. RESTful web services

#### 2. JAX-RS with Jersey

- 2.1. JAX-RS
- 2.2. Jersey
- 2.3. JAX-RS annotations

#### 3. Installation of Jersey

- 3.1. Manual setup of Jersey libraries in an Eclipse project
- 3.2. Required setup for Gradle with Eclipse web projects

#### 4. Web container

#### 5. Prerequisites

#### 6. Create your first RESTful Webservice

- 6.1. Create a new web project

#### 6.2. Run your test service

#### 7. Create a client

#### 8. RESTful web services and JAXB

- 8.1. Create necessary classes
- 8.2. Configure jersey usage
- 8.3. Create project
- 8.4. Create a client

#### 9. CRUD RESTful webservice

- 9.1. Project
- 9.2. Create a simple HTML form
- 9.3. Rest Service
- 9.4. Run
- 9.5. Create a client
- 9.6. Using the REST service via HTML page

#### 10. About this website

#### 11. Links and Literature

- 11.1. Rest Resources
- 11.2. vogella GmbH training and consulting support



Tutorials Services Products Books Com Search



Contact us

### QUICK LINKS

- [09 Nov - RCP Training](#)
- [23 Nov - Android Training](#)
- [vogella Training](#)
- [vogella Books](#)



## 1. REST - Representational State Transfer

### 1.1. What is REST?

REST is an architectural style which is based on web-standards and the HTTP protocol. REST was first described by Roy Fielding in 2000.

In a REST based architecture everything is a resource. A resource is accessed via a common interface based on the HTTP standard methods.

In a REST based architecture you typically have a REST server which provides access to the resources and a REST client which accesses and modifies the REST resources.

Every resource should support the HTTP common operations. Resources are identified by global IDs (which are typically URIs).

REST allows that resources have different representations, e.g., text, XML, JSON etc. The REST client can ask for a specific representation via the HTTP protocol (content negotiation).

## 1.2. HTTP methods

The *PUT*, *GET*, *POST* and *DELETE* methods are typical used in REST based architectures.

The following table gives an explanation of these operations.

- GET defines a reading access of the resource without side-effects. The resource is never changed via a GET request, e.g., the request has no side effects (idempotent).
- PUT creates a new resource. It must also be idempotent.
- DELETE removes the resources. The operations are idempotent. They can get repeated without leading to different results.
- POST updates an existing resource or creates a new resource.

## 1.3. RESTful web services

A RESTful web services are based on HTTP methods and the concept of REST. A RESTful web service typically defines the base URI for the services, the supported MIME-types (XML, text, JSON, user-defined, ...) and the set of operations (POST, GET, PUT, DELETE) which are supported.

# 2. JAX-RS with Jersey

## 2.1. JAX-RS

Java defines REST support via the *Java Specification Request (JSR) 311*. This specification is called JAX-RS (The Java API for RESTful Web Services). JAX-RS uses annotations to define the REST relevance of Java classes.

## 2.2. Jersey

*Jersey* is the reference implementation for the JSR 311 specification.

The Jersey implementation provides a library to implement Restful webservices in a Java servlet container.

On the server side Jersey provides a servlet implementation which scans predefined classes to identify RESTful resources. In your *web.xml* configuration file you register this servlet for your web application.

The Jersey implementation also provides a client library to communicate with a RESTful webservice.

The base URL of this servlet is:

```
http://your_domain:port/display-name/url-pattern/path_from_rest_class
```

This servlet analyzes the incoming HTTP request and selects the correct class and method to respond to this request. This selection is based on annotations in the class and methods.

A REST web application consists, therefore, out of data classes (resources) and services. These two types are typically maintained in different packages as the Jersey servlet will be instructed via the *web.xml* to scan certain packages for data classes.

JAX-RS supports the creation of XML and JSON via the Java Architecture for XML Binding (JAXB).

JAXB is described in the [JAXB Tutorial](#).

## 2.3. JAX-RS annotations

The most important annotations in JAX-RS are listed in the following table.

Table 1. JAX-RS annotations

Annotation	Description
@PATH(your_path)	Sets the path to base URL + /your_path. The base URL is based on your application name, the servlet and the URL pattern from the <i>web.xml</i> configuration file.
@POST	Indicates that the following method will answer to an HTTP POST request.
@GET	Indicates that the following method will answer to an HTTP GET request.
@PUT	Indicates that the following method will answer to an HTTP PUT request.
@DELETE	Indicates that the following method will answer to an HTTP DELETE request.
@Produces(MediaType.TEXT_PLAIN[, more-types])	@Produces defines which MIME type is delivered by a method annotated with @GET. In the example text ("text/plain") is produced. Other examples would be "application/xml" or "application/json".
@Consumes(type[, more-types])	@Consumes defines which MIME type is consumed by this method.
@PathParam	Used to inject values from the URL into a method parameter. This way you inject, for example, the ID of a resource into the method to get the correct object.

The complete path to a resource is based on the base URL and the @PATH annotation in your class.

```
http://your_domain:port/display-name/url-pattern/path_from_rest_class
```

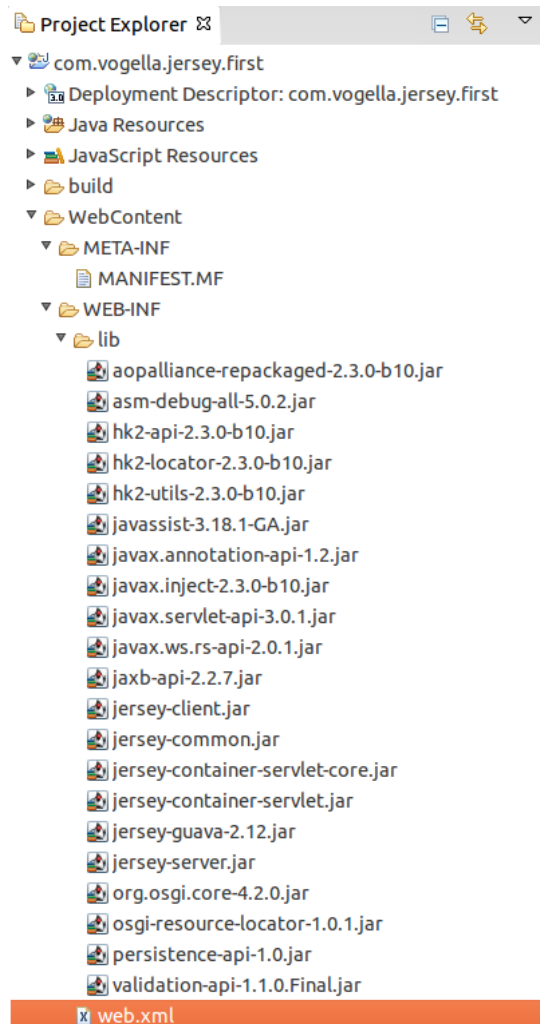
### 3. Installation of Jersey

#### 3.1. Manual setup of Jersey libraries in an Eclipse project

Download the Jersey distribution as zip file from the [Jersey download site](#).

The zip contains the Jersey implementation JAR and its core dependencies. It does not provide dependencies for third party JARs beyond those for JSON support and JavaDoc.

Copy all JARs from your Jersey download into the *WEB-INF/Lib* folder.



### 3.2. Required setup for Gradle with Eclipse web projects

TODO...

## 4. Web container

For this tutorial you can use any web container, for example Tomcat or the Google App Engine.

If you want to use Tomcat as servlet container please see [Eclipse WTP](#) and [Apache Tomcat](#) for instructions on how to install and use Eclipse WTP and Apache Tomcat.

Alternative you could also use the [Google App Engine](#) for running the server part of the following REST examples. If you use the Google App Engine, you do not have to install and configure Tomcat.

**Tip:** If you are using GAE/J, you have to create App Engine projects instead of *Dynamic Web Project*. The following description is based on Apache Tomcat.

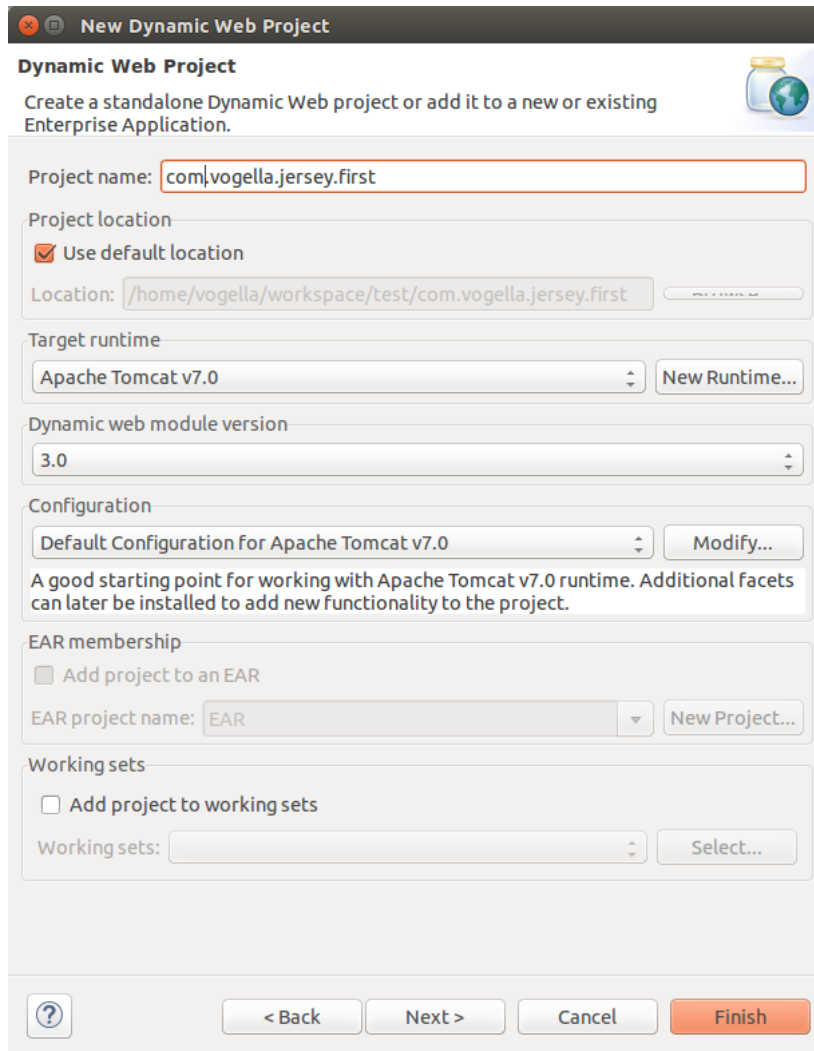
## 5. Prerequisites

The following description assumes that you are familiar with creating web applications in Eclipse. See [Eclipse WTP development](#) for an introduction into creating web applications with Eclipse.

## 6. Create your first RESTful Webservice

### 6.1. Create a new web project

Create a new *Dynamic Web Project* called *com.vogella.jersey.first*.



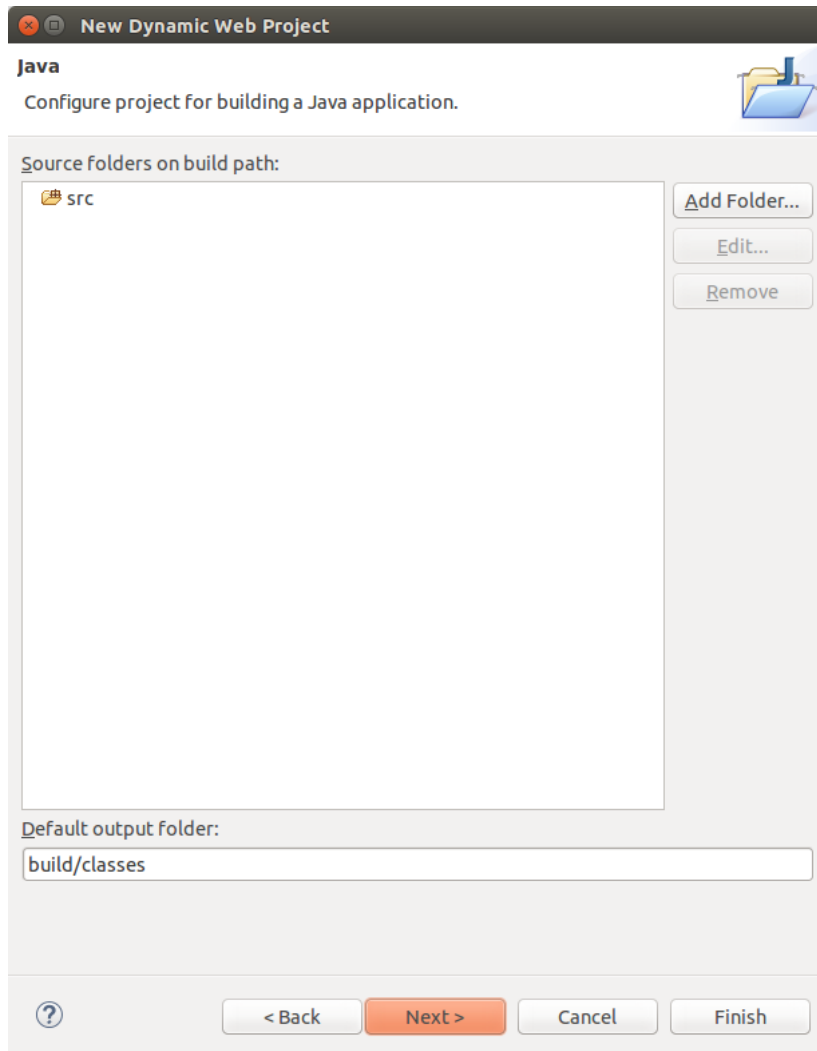
The screenshot shows the 'New Dynamic Web Project' dialog box in the Eclipse IDE. The dialog has a title bar 'New Dynamic Web Project' and a subtitle 'Dynamic Web Project'. Below the subtitle, it says 'Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.' and there is a small icon of a globe with a jar.

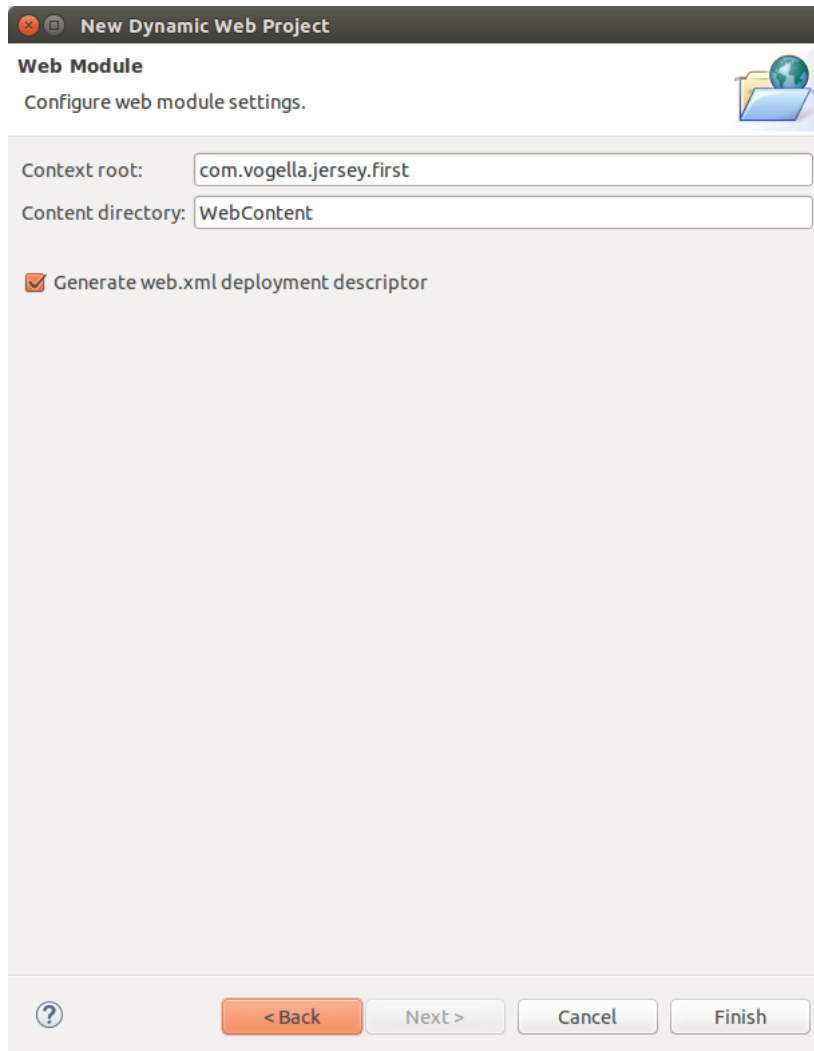
The dialog contains several sections:

- Project name:** A text field containing 'com.vogella.jersey.first'.
- Project location:** A section with a checked checkbox 'Use default location' and a text field 'Location:' containing '/home/vogella/workspace/test/com.vogella.jersey.first'.
- Target runtime:** A dropdown menu showing 'Apache Tomcat v7.0' and a 'New Runtime...' button.
- Dynamic web module version:** A dropdown menu showing '3.0'.
- Configuration:** A dropdown menu showing 'Default Configuration for Apache Tomcat v7.0' and a 'Modify...' button. Below this, a note states: 'A good starting point for working with Apache Tomcat v7.0 runtime. Additional facets can later be installed to add new functionality to the project.'
- EAR membership:** A section with an unchecked checkbox 'Add project to an EAR', a text field 'EAR project name:' containing 'EAR', and a 'New Project...' button.
- Working sets:** A section with an unchecked checkbox 'Add project to working sets', a text field 'Working sets:', and a 'Select...' button.

At the bottom of the dialog, there is a help icon (question mark in a circle) and four buttons: '< Back', 'Next >', 'Cancel', and 'Finish'.

Ensure that you create the `web.xml` deployment descriptor.





## 6.2. Configure jersey usage

See [Section 3. "Installation of Jersey"](#) for the setup.

## 6.3. Java Class

Create the following class.

```
package com.vogella.jersey.first;

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

// Plain old Java Object it does not extend as class or implements
// an interface

// The class registers its methods for the HTTP GET request using the @GET annotation.

// Using the @Produces annotation, it defines that it can deliver several MIME types,
// text, XML and HTML.

// The browser requests per default the HTML MIME type.

//Sets the path to base URL + /hello
@Path("/hello")
public class Hello {

    // This method is called if TEXT_PLAIN is request
    @GET
    @Produces(MediaType.TEXT_PLAIN)
```

```

public String sayPlainTextHello() {
    return "Hello Jersey";
}

// This method is called if XML is request
@GET
@Produces(MediaType.TEXT_XML)
public String sayXMLHello() {
    return "<?xml version='1.0'?'>" + "<hello> Hello Jersey" + "</hello>";
}

// This method is called if HTML is request
@GET
@Produces(MediaType.TEXT_HTML)
public String sayHtmlHello() {
    return "<html> " + "<title>" + "Hello Jersey" + "</title>"
        + "<body><h1>" + "Hello Jersey" + "</body></h1>" + "</html> ";
}
}

```

This class register itself as a get resource via the `@GET` annotation. Via the `@Produces` annotation it defines that it delivers the *text* and the *HTML* MIME types. It also defines via the `@Path` annotation that its service is available under the *hello* URL.

The browser will always request the HTML MIME type. To see the text version, you can use tool like [curl](#).

#### 6.4. Define Jersey Servlet dispatcher

You need to register Jersey as the servlet dispatcher for REST requests. Open the file *web.xml* and modify it to the following.

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>com.vogella.jersey.first</display-name>
  <servlet>
    <servlet-name>Jersey REST Service</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <!-- Register resources and providers under com.vogella.jersey.first package. -->
    <init-param>
      <param-name>jersey.config.server.provider.packages</param-name>
      <param-value>com.vogella.jersey.first</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Jersey REST Service</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>

```

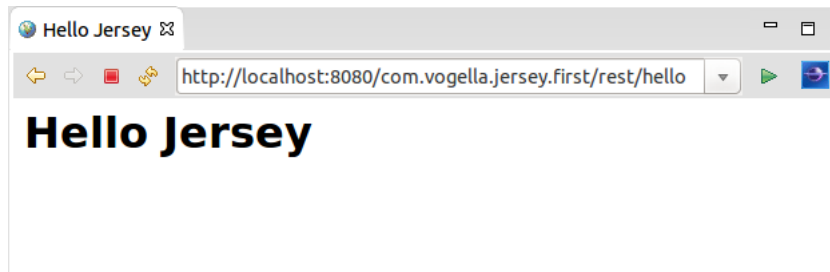
The parameter *jersey.config.server.provider.packages* defines in which package Jersey will look for the web service classes. This property must point to your resources classes. The URL pattern defines the part of the base URL your application will be placed.

#### 6.5. Run your rest service

Run you web application in Eclipse. See [Eclipse WTP](#) for details on how to run dynamic web applications.

You should be able to access your resources under the following URL:  
<http://localhost:8080/com.vogella.jersey.first/rest/hello>





This name is derived from the "display-name" defined in the *web.xml* file, augmented with the servlet-mapping URL-pattern and the *hello* `@Path` annotation from your class file. You should get the message "Hello Jersey".

The browser requests the HTML representation of your resource. In the next chapter we are going to write a client which will read the XML representation.

## 7. Create a client

Jersey contains a REST client library which can be used for testing or to build a real client in Java. The usage of this library is demonstrated in the following tutorial.

Create a new Java project *com.vogella.jersey.first.client* and add the Jersey JARs to the project and the project build path. Create the following test class.

```
package com.vogella.jersey.first;

import java.net.URI;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriBuilder;

import org.glassfish.jersey.client.ClientConfig;

public class Test {

    public static void main(String[] args) {
        ClientConfig config = new ClientConfig();

        Client client = ClientBuilder.newClient(config);

        WebTarget target = client.target(getBaseURI());

        String response = target.path("rest").
            path("hello").
            request().
            accept(MediaType.TEXT_PLAIN).
            get(Response.class)
            .toString();

        String plainAnswer = target.path("rest").path("hello").request().accept(MediaType.TEXT_PLAIN).get(String.class);
        String xmlAnswer = target.path("rest").path("hello").request().accept(MediaType.TEXT_XML).get(String.class);
        String htmlAnswer = target.path("rest").path("hello").request().accept(MediaType.TEXT_HTML).get(String.class);

        System.out.println(response);
        System.out.println(plainAnswer);
        System.out.println(xmlAnswer);
        System.out.println(htmlAnswer);
    }

    private static URI getBaseURI() {
        return UriBuilder.fromUri("http://localhost:8080/com.vogella.jersey.first").build();
    }
}
```

## 8. RESTful web services and JAXB

JAX-RS supports the automatic creation of XML and JSON via JAXB. For an introduction into XML please see [Java and XML - Tutorial](#). For an introduction into JAXB please see [JAXB](#). You can continue this tutorial without reading these tutorials, but they contain more background information.

### 8.1. Create necessary classes

Create a new *Dynamic Web Project* called *com.vogella.jersey.jaxb*. Ensure you create *web.xml* deployment descriptor.

### 8.2. Configure jersey usage

See [Section 3. "Installation of Jersey"](#) for the setup.

### 8.3. Create project

Create your domain class.

```
package com.vogella.jersey.jaxb.model;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
// JAX-RS supports an automatic mapping from JAXB annotated class to XML and JSON
// Isn't that cool?
public class Todo {
    private String summary;
    private String description;
    public String getSummary() {
        return summary;
    }
    public void setSummary(String summary) {
        this.summary = summary;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
}
```

Create the following resource class. This class simply returns an instance of the `Todo` class.

```
package com.vogella.jersey.jaxb.model;

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

@Path("/todo")
public class TodoResource {
    // This method is called if XML is request
    @GET
    @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
    public Todo getXML() {
        Todo todo = new Todo();
        todo.setSummary("This is my first todo");
        todo.setDescription("This is my first todo");
        return todo;
    }

    // This can be used to test the integration with the browser
    @GET
    @Produces({ MediaType.TEXT_XML })
    public Todo getHTML() {
        Todo todo = new Todo();
        todo.setSummary("This is my first todo");
        todo.setDescription("This is my first todo");
    }
}
```

```

    return todo;
}

}

```

Change *web.xml* to the following.

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.
com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.s
un.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>com.vogella.jersey.first</display-name>
  <servlet>
    <servlet-name>Jersey REST Service</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <!-- Register resources and providers under com.vogella.jersey.first package. -->
    <init-param>
      <param-name>jersey.config.server.provider.packages</param-name>
      <param-value>com.vogella.jersey.jaxb</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Jersey REST Service</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>

```

Run your web application in Eclipse and validate that you can access your service. Your application should be available under the following URL.

```
http://localhost:8080/com.vogella.jersey.jaxb/rest/todo
```

#### 8.4. Create a client

Create a new Java project *de.vogella.jersey.jaxb.client* and add the Jersey JARs to the project and the project build path. Create the following test class.

```

package com.vogella.jersey.jaxb.client;

import java.net.URI;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.UriBuilder;

import org.glassfish.jersey.client.ClientConfig;

public class TodoTest {

    public static void main(String[] args) {
        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);

        WebTarget target = client.target(getBaseURI());
        // Get XML
        String xmlResponse = target.path("rest").path("todo").request()
            .accept(MediaType.TEXT_XML).get(String.class);
        // Get XML for application
        String xmlAppResponse = target.path("rest").path("todo").request()
            .accept(MediaType.APPLICATION_XML).get(String.class);

        // For JSON response also add the Jackson libraries to your webapplication
        // In this case you would also change the client registration to
        // ClientConfig config = new ClientConfig().register(JacksonFeature.class);
        // Get JSON for application
        // System.out.println(target.path("rest").path("todo").request()
        //     .accept(MediaType.APPLICATION_JSON).get(String.class));

        System.out.println(xmlResponse);
        System.out.println(xmlAppResponse);
    }
}

```

```

    }

    private static URI getBaseURI() {
        return UriBuilder.fromUri("http://localhost:8080/_com.vogella.jersey.jaxb").build(
    );
    }
}

```

## 9. CRUD RESTful webservice

This section creates a CRUD (Create, Read, Update, Delete) restful web service. It will allow to maintain a list of TODOs in your web application via HTTP calls.

### 9.1. Project

Create a new dynamic project called *com.vogella.jersey.todo* and add the Jersey libs. Change the *web.xml* file to the following.

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.
com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.s
un.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>com.vogella.jersey.first</display-name>
  <servlet>
    <servlet-name>Jersey REST Service</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <!-- Register resources and providers under com.vogella.jersey.first package. -->
    <init-param>
      <param-name>jersey.config.server.provider.packages</param-name>
      <param-value>com.vogella.jersey.todo.resources</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Jersey REST Service</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>

```

Create the following data model and a Singleton which serves as the data provider for the model. We use the implementation based on an enumeration. Please see the link for details. The `Todo` class is annotated with a JAXB annotation. See [Java and XML](#) to learn about JAXB.

```

package com.vogella.jersey.todo.model;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Todo {
    private String id;
    private String summary;
    private String description;

    public Todo(){
    }

    public Todo (String id, String summary){
        this.id = id;
        this.summary = summary;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getSummary() {
        return summary;
    }

    public void setSummary(String summary) {
        this.summary = summary;
    }
}

```

```

    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
}

```

```

package com.vogella.jersey.todo.dao;

import java.util.HashMap;
import java.util.Map;

import com.vogella.jersey.todo.model.TODO;

public enum TODODao {
    instance;

    private Map<String, TODO> contentProvider = new HashMap<>();

    private TODODao() {

        TODO todo = new TODO("1", "Learn REST");
        todo.setDescription("Read http://www.vogella.com/tutorials/REST/article.html");
        contentProvider.put("1", todo);
        todo = new TODO("2", "Do something");
        todo.setDescription("Read complete http://www.vogella.com");
        contentProvider.put("2", todo);

    }
    public Map<String, TODO> getModel(){
        return contentProvider;
    }
}

```

## 9.2. Create a simple HTML form

The REST service can be used via HTML forms. The following HTML form will allow to post new data to the service. Create the following page called *create\_todo.html* in the *WebContent* folder.

```

<!DOCTYPE html>
<html>
<head>
    <title>Form to create a new resource</title>
</head>
<body>
    <form action="../com.vogella.jersey.todo/rest/todos" method="POST">
        <label for="id">ID</label>
        <input name="id" />
        <br/>
        <label for="summary">Summary</label>
        <input name="summary" />
        <br/>
        Description:
        <TEXTAREA NAME="description" COLS=40 ROWS=6></TEXTAREA>
        <br/>
        <input type="submit" value="Submit" />
    </form>
</body>
</html>

```

## 9.3. Rest Service

Create the following classes which will be used as REST resources.

```

package com.vogella.jersey.todo.resources;

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;

```

```

import javax.ws.rs.PUT;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import javax.xml.bind.JAXBElement;

import com.vogella.jersey.todo.dao.TodoDao;
import com.vogella.jersey.todo.model.Todo;

public class TodoResource {
    @Context
    UriInfo uriInfo;
    @Context
    Request request;
    String id;
    public TodoResource(UriInfo uriInfo, Request request, String id) {
        this.uriInfo = uriInfo;
        this.request = request;
        this.id = id;
    }

    //Application integration
    @GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Todo getTodo() {
        Todo todo = TodoDao.instance.getModel().get(id);
        if(todo==null)
            throw new RuntimeException("Get: Todo with " + id + " not found");
        return todo;
    }

    // for the browser
    @GET
    @Produces(MediaType.TEXT_XML)
    public Todo getTodoHTML() {
        Todo todo = TodoDao.instance.getModel().get(id);
        if(todo==null)
            throw new RuntimeException("Get: Todo with " + id + " not found");
        return todo;
    }

    @PUT
    @Consumes(MediaType.APPLICATION_XML)
    public Response putTodo(JAXBElement<Todo> todo) {
        Todo c = todo.getValue();
        return putAndGetResponse(c);
    }

    @DELETE
    public void deleteTodo() {
        Todo c = TodoDao.instance.getModel().remove(id);
        if(c==null)
            throw new RuntimeException("Delete: Todo with " + id + " not found");
    }

    private Response putAndGetResponse(Todo todo) {
        Response res;
        if(TodoDao.instance.getModel().containsKey(todo.getId())) {
            res = Response.noContent().build();
        } else {
            res = Response.created(uriInfo.getAbsolutePath()).build();
        }
        TodoDao.instance.getModel().put(todo.getId(), todo);
        return res;
    }
}

```

```

package com.vogella.jersey.todo.resources;

import java.io.IOException;

```

```

import java.util.ArrayList;
import java.util.List;

import javax.servlet.http.HttpServletResponse;
import javax.ws.rs.Consumes;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.UriInfo;

import com.vogella.jersey.todo.dao.TodoDao;
import com.vogella.jersey.todo.model.Todo;

// Will map the resource to the URL todos
@Path("/todos")
public class TodosResource {

    // Allows to insert contextual objects into the class,
    // e.g. ServletContext, Request, Response, UriInfo
    @Context
    UriInfo uriInfo;
    @Context
    Request request;

    // Return the list of todos to the user in the browser
    @GET
    @Produces(MediaType.TEXT_XML)
    public List<Todo> getTodosBrowser() {
        List<Todo> todos = new ArrayList<Todo>();
        todos.addAll(TodoDao.instance.getModel().values());
        return todos;
    }

    // Return the list of todos for applications
    @GET
    @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
    public List<Todo> getTodos() {
        List<Todo> todos = new ArrayList<Todo>();
        todos.addAll(TodoDao.instance.getModel().values());
        return todos;
    }

    // returns the number of todos
    // Use http://localhost:8080/com.vogella.jersey.todo/rest/todos/count
    // to get the total number of records
    @GET
    @Path("count")
    @Produces(MediaType.TEXT_PLAIN)
    public String getCount() {
        int count = TodoDao.instance.getModel().size();
        return String.valueOf(count);
    }

    @POST
    @Produces(MediaType.TEXT_HTML)
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public void newTodo(@FormParam("id") String id,
        @FormParam("summary") String summary,
        @FormParam("description") String description,
        @Context HttpServletResponse servletResponse) throws IOException {
        Todo todo = new Todo(id, summary);
        if (description != null) {
            todo.setDescription(description);
        }
        TodoDao.instance.getModel().put(id, todo);

        servletResponse.sendRedirect("../create_todo.html");
    }

    // Defines that the next path parameter after todos is
    // treated as a parameter and passed to the TodoResources

```

```
// Allows to type http://localhost:8080/com.vogella.jersey.todo/rest/todos/1
// 1 will be treaded as parameter todo and passed to TodoResource
@Path("/{todo}")
public TodoResource getTodo(@PathParam("todo") String id) {
    return new TodoResource(uriInfo, request, id);
}
}
```

This TodosResource uses the `@PathParam` annotation to define that the `id` is inserted as parameter.

## 9.4. Run

Run you web application in Eclipse and test the availability of your REST service under:

**<http://localhost:8080/com.vogella.jersey.todo/rest/todos>**. You should see the XML representation of your TODO items.



```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
- <todoes>
- <todo>
  <description>Read complete http://www.vogella.de</description>
  <id>2</id>
  <summary>Do something</summary>
</todo>
- <todo>
  <description>Read http://www.vogella.de/articles/REST/article.html</description>
  <id>1</id>
  <summary>Learn REST</summary>
</todo>
</todoes>
```

To see the count of TODO items use

**<http://localhost:8080/com.vogella.jersey.todo/rest/todos/count>** to see an exiting TODO use "http://localhost:8080/com.vogella.jersey.todo/rest/todos/{id}", e.g.,

**<http://localhost:8080/com.vogella.jersey.todo/rest/todos/1>** to see the TODO with ID 1. We currently have only TODOs with the ids 1 and 2, all other requests will result in an HTTP error code.

Please note that with the browser you can only issue HTTP GET requests. The next chapter will use the Jersey client libraries to issue get, post and delete.

## 9.5. Create a client

To test your service can you create new class in your server project. This project has already all required libs in the classpath, so this is faster than creating a new project.

Create the following class.

```
package com.vogella.jersey.todo.client;

import java.net.URI;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Form;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriBuilder;

import org.glassfish.jersey.client.ClientConfig;

import com.vogella.jersey.todo.model.Todo;

public class Tester {
    public static void main(String[] args) {

        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);
        WebTarget service = client.target(getBaseURI());

        // create one todo
        Todo todo = new Todo("3", "Blabla");
```



```

        Response response = service.path("rest").path("todos").path(todo.getId()).request(
        MediaType.APPLICATION_XML).put(Entity.entity(todo, MediaType.APPLICATION_XML), Response.
        class);

        // Return code should be 201 == created resource
        System.out.println(response.getStatus());

        // Get the Todos
        System.out.println(service.path("rest").path("todos").request().accept(MediaType.TEXT_XML).get(String.class));

        // // Get JSON for application
        // System.out.println(service.path("rest").path("todos").request().accept(MediaType.APPLICATION_JSON).get(String.class));

        // Get XML for application
        System.out.println(service.path("rest").path("todos").request().accept(MediaType.APPLICATION_XML).get(String.class));

        //Get Todo with id 1
        Response checkDelete = service.path("rest").path("todos/1").request().accept(MediaType.APPLICATION_XML).get();

        //Delete Todo with id 1
        service.path("rest").path("todos/1").request().delete();

        //Get get all Todos id 1 should be deleted
        System.out.println(service.path("rest").path("todos").request().accept(MediaType.APPLICATION_XML).get(String.class));

        //Create a Todo
        Form form =new Form();
        form.param("id", "4");
        form.param("summary", "Demonstration of the client lib for forms");
        response = service.path("rest").path("todos").request().post(Entity.entity(form, MediaType.APPLICATION_FORM_URLENCODED), Response.class);
        System.out.println("Form response " + response.getStatus());

        //Get all the todos, id 4 should have been created
        System.out.println(service.path("rest").path("todos").request().accept(MediaType.APPLICATION_XML).get(String.class));

    }

    private static URI getBaseURI() {
        return UriBuilder.fromUri("http://localhost:8080/com.vogella.jersey.todo").build();
    }
}

```

## 9.6. Using the REST service via HTML page

The above example contains a form which calls a post method of your rest service.

## 10. About this website



## 11. Links and Literature

### 11.1. Rest Resources

[Jersey Homepage](#)

[JSR 311](#)

[IBM Article about Rest with Tomcat and Jersey](#)

### 11.2. vogella GmbH training and consulting support