**Jersey hello world example**Jersey 2.23.2 User Guide  
https://jersey.java.net/documentation/latest/index.html  
  
[Jersey](http://jersey.java.net/), reference implementation to develope RESTful web service based on the [JAX-RS (JSR 311)](http://jsr311.java.net/nonav/releases/1.1/index.html) specification.

In this tutorial, we show you how to develop a simple hello world REST web application with **Jersey**.

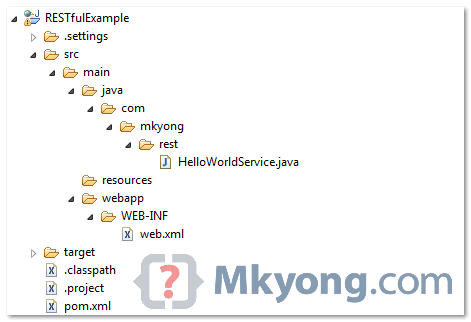
Technologies and Tools used in this article:

1. Jersey 1.8
2. JDK 1.6
3. Tomcat 6.0
4. Maven 3.0.3
5. Eclipse 3.6

**Note**  
If you want to know what and how REST works, just search on Google, ton of available resources.

**1. Directory Structure**

This is the final web project structure of this tutorial.



**2. Standard Web Project**

Create a standard Maven web project structure.

mvn archetype:generate -DgroupId=com.mkyong.rest -DartifactId=RESTfulExample

-DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false

**Note**  
To support Eclipse, use Maven command :

mvn eclipse:eclipse -Dwtpversion=2.0

**3. Project Dependencies**

Jersey is published in Java.net Maven repository. To develop Jersey REST application , just declares “**jersey-server**” in Maven pom.xml.

*File : pom.xml*

<project ...>

<repositories>

<repository>

<id>maven2-repository.java.net</id>

<name>Java.net Repository for Maven</name>

<url>http://download.java.net/maven/2/</url>

<layout>default</layout>

</repository>

</repositories>

<dependencies>

<dependency>

<groupId>com.sun.jersey</groupId>

<artifactId>jersey-server</artifactId>

<version>1.8</version>

</dependency>

</dependencies>

</project>

**4. REST Service**

Simple REST service with Jersey.

package com.mkyong.rest;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.PathParam;

import javax.ws.rs.core.Response;

@Path("/hello")

public class HelloWorldService {

@GET

@Path("/{param}")

public Response getMsg(@PathParam("param") String msg) {

String output = "Jersey say : " + msg;

return Response.status(200).entity(output).build();

}

}

**5. web.xml**

In web.xml, register “com.sun.jersey.spi.container.servlet.ServletContainer“, and puts your Jersey service folder under “**init-param**“, “com.sun.jersey.config.property.packages“.

*File : web.xml*

<web-app id="WebApp\_ID" version="2.4"

xmlns="http://java.sun.com/xml/ns/j2ee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<display-name>Restful Web Application</display-name>

<servlet>

<servlet-name>jersey-serlvet</servlet-name>

<servlet-class>

com.sun.jersey.spi.container.servlet.ServletContainer

</servlet-class>

<init-param>

<param-name>com.sun.jersey.config.property.packages</param-name>

<param-value>com.mkyong.rest</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>jersey-serlvet</servlet-name>

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

</servlet-mapping>

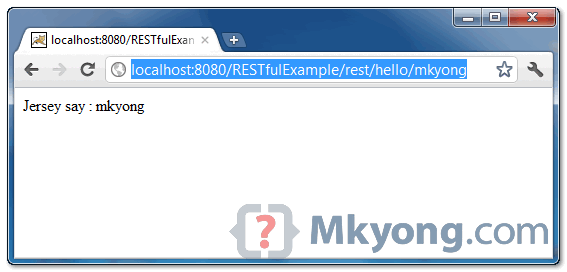
</web-app>

**6. Demo**

In this example, web request from “**projectURL/rest/hello/**” will match to “**HelloWorldService**“, via @Path("/hello").

And the “**{any values}**” from “**projectURL/rest/hello/{any values}**” will match to parameter annotated with @PathParam.

*URL : http://localhost:8080/RESTfulExample/rest/hello/mkyong*



**Now Consuming of Webservices In REST**[How to consume REST in Java](http://stackoverflow.com/questions/12916169/how-to-consume-rest-in-java)  
  
  
1: For Spring Application  
  
This is very complicated in java, which is why I would suggest using Spring's [RestTemplate](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/client/RestTemplate.html) abstraction:

String result =

restTemplate.getForObject(

"http://example.com/hotels/{hotel}/bookings/{booking}",

String.class,"42", "21"

);

You can use WSDL to generate the stub and Classes required to hit the SOAP Web Service.  
  
You can use HttpURLConnection. Below is an example of calling a RESTful service using the Java SE APIs including JAXB:

String uri =

"http://localhost:8080/CustomerService/rest/customers/1";

URL url = new URL(uri);

HttpURLConnection connection =

(HttpURLConnection) url.openConnection();

connection.setRequestMethod("GET");

connection.setRequestProperty("Accept", "application/xml");

JAXBContext jc = JAXBContext.newInstance(Customer.class);

InputStream xml = connection.getInputStream();

Customer customer =

(Customer) jc.createUnmarshaller().unmarshal(xml);

connection.disconnect();

Intretsing Comments must read:

|  |  |
| --- | --- |
|  | 1.If you are calling a RESTful service from a Service Provider (e.g Facebook, Twitter), you can do it with any *flavour* of your choice:  If you don't want to use external libraries, you can use java.net.HttpURLConnection or javax.net.ssl.HttpsURLConnection (for SSL), but that is call encapsulated in a Factory type pattern in java.net.URLConnection. To receive the result, you will have to connection.getInputStream() which returns you an InputStream. You will then have to convert your input stream to string and parse the string into it's representative object (e.g. XML, JSON, etc).  Alternatively, [Apache HttpClient](http://hc.apache.org/downloads.cgi) (version 4 is the latest). It's more stable and robust than java's default URLConnection and it supports most (if not all) HTTP protocol (as well as it can be set to Strict mode). Your response will still be in InputStream and you can use it as mentioned above  Example: If you just need to make a simple call to a REST service from java you use something along these line  /\*  \* Stolen from http://xml.nig.ac.jp/tutorial/rest/index.html  \* and http://www.dr-chuck.com/csev-blog/2007/09/calling-rest-web-services-from-java/  \*/  import java.io.\*;  import java.net.\*;  public class Rest {  public static void main(String[] args) throws IOException {  URL url = new URL(INSERT\_HERE\_YOUR\_URL);  String query = INSERT\_HERE\_YOUR\_URL\_PARAMETERS;  //make connection  URLConnection urlc = url.openConnection();  //use post mode  urlc.setDoOutput(true);  urlc.setAllowUserInteraction(false);  //send query  PrintStream ps = new PrintStream(urlc.getOutputStream());  ps.print(query);  ps.close();  //get result  BufferedReader br = new BufferedReader(new InputStreamReader(urlc  .getInputStream()));  String l = null;  while ((l=br.readLine())!=null) {  System.out.println(l);  }  br.close();  }  } |

There are several RESTful APIs around. I would recommend Jersey;

<https://jersey.java.net/>

Client API documentation is here;

<https://jersey.java.net/documentation/latest/index.html>

**Update**  
Location for the OAuth docs in the comment below is a dead link and has moved to <https://jersey.java.net/nonav/documentation/latest/security.html#d0e12334>

2.   
  
99% of the time when people use the term REST, they really mean HTTP; they could care less about “resources”, “representations”, “state transfers”, “uniform interfaces”, “hypermedia”, or any other [constraints or aspects of the REST architecture style identified by Fielding](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm). The abstractions provided by various REST frameworks are therefore confusing and unhelpful.

So: you want to send HTTP requests using Java in 2015. You want an API that is clear, expressive, intuitive, idiomatic, simple. What to use? I no longer use Java, but for the past few years the Java HTTP client library that has seemed the most promising and interesting is [OkHttp](http://square.github.io/okhttp/). Check it out.

You can definitely interact with RESTful web services by using URLConnection or [HTTPClient](http://hc.apache.org/httpcomponents-client-ga/index.html) to code HTTP requests.

However, it's generally more desirable to use a library or framework which provides a simpler and more semantic API specifically designed for this purpose. This makes the code easier to write, read, and debug, and reduces duplication of effort. These frameworks generally implement some great features which aren't necessarily present or easy to use in lower-level libraries, such as content negotiation, caching, and authentication.

Some of the most mature options are [Jersey](http://jersey.java.net/nonav/documentation/latest/user-guide.html#client-api), [RESTEasy](http://jboss.org/resteasy), and [Restlet](http://wiki.restlet.org/docs_2.0/13-restlet/21-restlet/318-restlet/320-restlet.html).

I'm most familiar with Restlet, and Jersey, let's look at how we'd make a POST request with both APIs.

**Jersey Example**

Form form = new Form();

form.add("x", "foo");

form.add("y", "bar");

Client client = ClientBuilder.newClient();

WebTarget resource = client.target("http://localhost:8080/someresource");

Builder request = resource.request();

request.accept(MediaType.APPLICATION\_JSON);

Response response = request.get();

if (response.getStatusInfo().getFamily() == Family.SUCCESSFUL) {

System.out.println("Success! " + response.getStatus());

System.out.println(response.getEntity());

} else {

System.out.println("ERROR! " + response.getStatus());

System.out.println(response.getEntity());

}

**Restlet Example**

Form form = new Form();

form.add("x", "foo");

form.add("y", "bar");

ClientResource resource = new ClientResource("http://localhost:8080/someresource");

Response response = resource.post(form.getWebRepresentation());

if (response.getStatus().isSuccess()) {

System.out.println("Success! " + response.getStatus());

System.out.println(response.getEntity().getText());

} else {

System.out.println("ERROR! " + response.getStatus());

System.out.println(response.getEntity().getText());

}

Of course, GET requests are even simpler, and you can also specify things like entity tags and Accept headers, but hopefully these examples are usefully non-trivial but not too complex.

As you can see, Restlet and Jersey have similar client APIs. I believe they were developed around the same time, and therefore influenced each other.

I find the Restlet API to be a little more semantic, and therefore a little clearer, but YMMV.

As I said, I'm most familiar with Restlet, I've used it in many apps for years, and I'm very happy with it. It's a very mature, robust, simple, effective, active, and well-supported framework. I can't speak to Jersey or RESTEasy, but my impression is that they're both also solid choices.

You can find the complete example here. By MKYong  
  
In this tutorial, we show you how to create a RESTful Java client with Java build-in **HTTP client library**. It’s simple to use and good enough to perform basic operations for REST service.

The RESTful services from last “[Jackson + JAX-RS](http://www.mkyong.com/webservices/jax-rs/integrate-jackson-with-resteasy/)” article will be reused, and we will use “java.net.URL” and “java.net.HttpURLConnection” to create a simple Java client to send “**GET**” and “**POST**” request.

**1. GET Request**

Review last REST service, return “json” data back to client.

@Path("/json/product")

public class JSONService {

@GET

@Path("/get")

@Produces("application/json")

public Product getProductInJSON() {

Product product = new Product();

product.setName("iPad 3");

product.setQty(999);

return product;

}

//...

Java client to send a “GET” request.

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.net.HttpURLConnection;

import java.net.MalformedURLException;

import java.net.URL;

public class NetClientGet {

// http://localhost:8080/RESTfulExample/json/product/get

public static void main(String[] args) {

try {

URL url = new URL("http://localhost:8080/RESTfulExample/json/product/get");

HttpURLConnection conn = (HttpURLConnection) url.openConnection();

conn.setRequestMethod("GET");

conn.setRequestProperty("Accept", "application/json");

if (conn.getResponseCode() != 200) {

throw new RuntimeException("Failed : HTTP error code : "

+ conn.getResponseCode());

}

BufferedReader br = new BufferedReader(new InputStreamReader(

(conn.getInputStream())));

String output;

System.out.println("Output from Server .... \n");

while ((output = br.readLine()) != null) {

System.out.println(output);

}

conn.disconnect();

} catch (MalformedURLException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

}

}

Output…

Output from Server ....

{"qty":999,"name":"iPad 3"}

**2. POST Request**

Review last REST service, accept “json” data and convert it into Product object, via Jackson provider automatically.

@Path("/json/product")

public class JSONService {

@POST

@Path("/post")

@Consumes("application/json")

public Response createProductInJSON(Product product) {

String result = "Product created : " + product;

return Response.status(201).entity(result).build();

}

//...

Java client to send a “POST” request, with json string.

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.OutputStream;

import java.net.HttpURLConnection;

import java.net.MalformedURLException;

import java.net.URL;

public class NetClientPost {

// http://localhost:8080/RESTfulExample/json/product/post

public static void main(String[] args) {

try {

URL url = new URL("http://localhost:8080/RESTfulExample/json/product/post");

HttpURLConnection conn = (HttpURLConnection) url.openConnection();

conn.setDoOutput(true);

conn.setRequestMethod("POST");

conn.setRequestProperty("Content-Type", "application/json");

String input = "{\"qty\":100,\"name\":\"iPad 4\"}";

OutputStream os = conn.getOutputStream();

os.write(input.getBytes());

os.flush();

if (conn.getResponseCode() != HttpURLConnection.HTTP\_CREATED) {

throw new RuntimeException("Failed : HTTP error code : "

+ conn.getResponseCode());

}

BufferedReader br = new BufferedReader(new InputStreamReader(

(conn.getInputStream())));

String output;

System.out.println("Output from Server .... \n");

while ((output = br.readLine()) != null) {

System.out.println(output);

}

conn.disconnect();

} catch (MalformedURLException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

}

}

Output…

Output from Server ....

Product created : Product [name=iPad 4, qty=100]  
  
  
  
  
  
**Publishing Using UDDI Registry:**  
  
A registry is of no use without some way to access it. The UDDI standard version 2.0 specifies two interfaces for service consumers and service providers to interact with the registry.

Service consumers use **Inquiry Interface** to find a service, and service providers use **Publisher Interface** to list a service.

The core of the UDDI interface is the UDDI XML Schema definitions. These define the fundamental UDDI data types through which all the information flows.

**Jersy API:**

**Main packages javax.ws.rs, javax.ws.rs.client  
javax.ws.rs.core and javax.ws.rs.container**

javax.ws.rs contains High-level interfaces and annotations used to create RESTful service resources

|  |  |
| --- | --- |
| [**ApplicationPath**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/ApplicationPath.html) | Identifies the application path that serves as the base URI for all resource URIs provided by [**Path**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/Path.html). |
| [**BeanParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/BeanParam.html) | The annotation that may be used to inject custom JAX-RS "parameter aggregator" value object into a resource class field, property or resource method parameter. |
| [**ConstrainedTo**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/ConstrainedTo.html) | Indicates the run-time context in which an annotated JAX-RS provider is applicable. |
| [**Consumes**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/Consumes.html) | Defines the media types that the methods of a resource class or [**MessageBodyReader**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/ext/MessageBodyReader.html) can accept. |
| [**CookieParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/CookieParam.html) | Binds the value of a HTTP cookie to a resource method parameter, resource class field, or resource class bean property. |
| [**DefaultValue**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/DefaultValue.html) | Defines the default value of request meta-data that is bound using one of the following annotations: [**PathParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/PathParam.html), [**QueryParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/QueryParam.html), [**MatrixParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/MatrixParam.html), [**CookieParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/CookieParam.html), [**FormParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/FormParam.html), or [**HeaderParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/HeaderParam.html). |
| [**DELETE**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/DELETE.html) | Indicates that the annotated method responds to HTTP DELETE requests. |
| [**Encoded**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/Encoded.html) | Disables automatic decoding of parameter values bound using [**QueryParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/QueryParam.html), [**PathParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/PathParam.html), [**FormParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/FormParam.html) or [**MatrixParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/MatrixParam.html). |
| [**FormParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/FormParam.html) | Binds the value(s) of a form parameter contained within a request entity body to a resource method parameter. |
| [**GET**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/GET.html) | Indicates that the annotated method responds to HTTP GET requests. |
| [**HEAD**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/HEAD.html) | Indicates that the annotated method responds to HTTP HEAD requests. |
| [**HeaderParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/HeaderParam.html) | Binds the value(s) of a HTTP header to a resource method parameter, resource class field, or resource class bean property. |
| [**HttpMethod**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/HttpMethod.html) | Associates the name of a HTTP method with an annotation. |
| [**MatrixParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/MatrixParam.html) | Binds the value(s) of a URI matrix parameter to a resource method parameter, resource class field, or resource class bean property. |
| [**NameBinding**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/NameBinding.html) | Meta-annotation used to create name binding annotations for filters and interceptors. |
| [**OPTIONS**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/OPTIONS.html) | Indicates that the annotated method responds to HTTP OPTIONS requests. |
| [**Path**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/Path.html) | Identifies the URI path that a resource class or class method will serve requests for. |
| [**PathParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/PathParam.html) | Binds the value of a URI template parameter or a path segment containing the template parameter to a resource method parameter, resource class field, or resource class bean property. |
| [**POST**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/POST.html) | Indicates that the annotated method responds to HTTP POST requests. |
| [**Produces**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/Produces.html) | Defines the media type(s) that the methods of a resource class or [**MessageBodyWriter**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/ext/MessageBodyWriter.html) can produce. |
| [**PUT**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/PUT.html) | Indicates that the annotated method responds to HTTP PUT requests. |
| [**QueryParam**](https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/QueryParam.html) | Binds the value(s) of a HTTP query parameter to a resource method parameter, resource class field, or resource class bean property. |