# Web Service

## Network Protocols

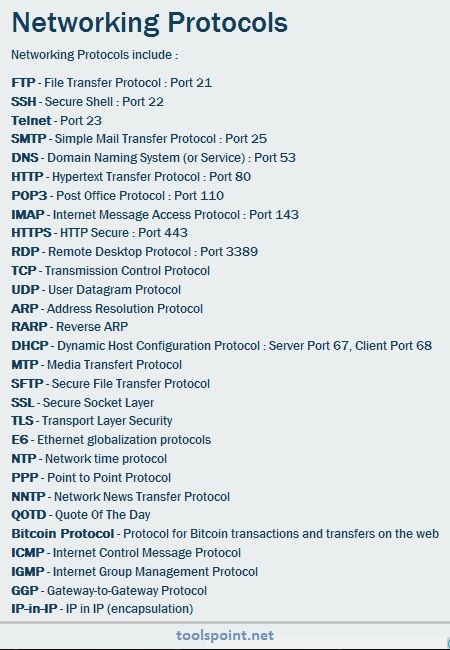
Network protocols are formal standards and policies comprised of rules, procedures and formats that define communication between two or more devices over a network.

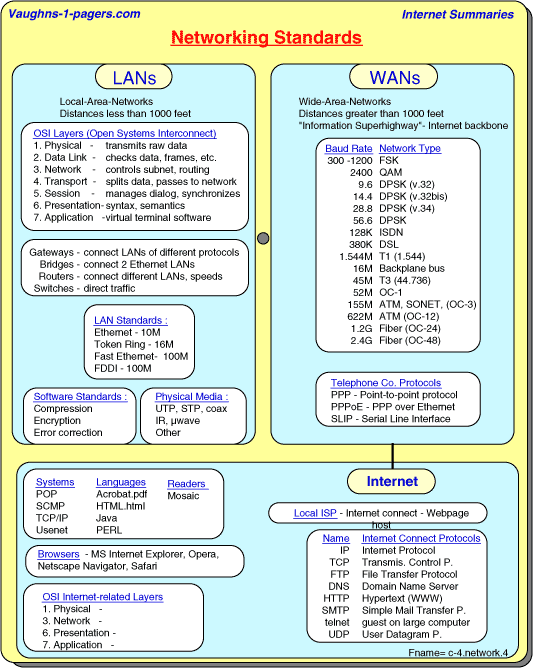
Network protocols govern the end-to-end processes of timely, secure and managed data or network communication.

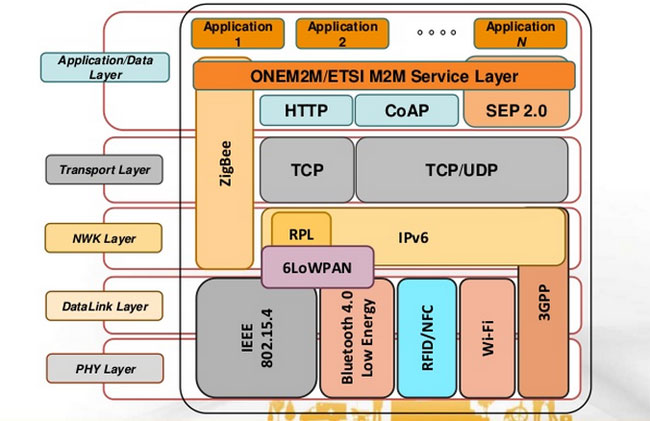
Network protocols incorporate all the processes, requirements and constraints of initiating and accomplishing communication between computers, servers, routers and other network enabled devices.

Network protocols must be confirmed and installed by the sender and receiver to ensure network/data communication and apply to software and hardware nodes that communicate on a network.

* Ethernet (Physical/Data Link Layers) ...
* IP and IPX (Network Layer) ...
* TCP and SPX (Transport Layer) ...
* HTTP, FTP, SMTP and DNS (Session/Presentation/Application Layers)







## What is Web Service

A **Web Service** is can be defined by following ways:

* is a client server application or application component for communication.
* method of communication between two devices over network.
* is a software system for interoperable machine to machine communication.
* is a collection of standards or protocols for exchanging information between two devices or application.

Let's understand it by the figure given below:

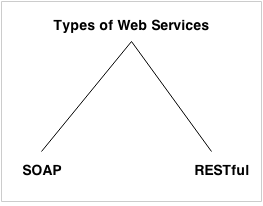


As you can see in the figure, java, .net or PHP applications can communicate with other applications through web service over the network. For example, java application can interact with Java, .Net and PHP applications. So web service is a language independent way of communication.

## Types of Web Services

There are mainly two types of web services.

1. SOAP web services.
2. RESTful web services.



## SOAP

SOAP is an acronym for Simple Object Access Protocol.

SOAP is a XML-based protocol for accessing web services.

SOAP is a W3C recommendation for communication between applications.

SOAP is XML based, so it is platform independent and language independent. In other words, it can be used with Java, .Net or PHP language on any platform.

## WSDL

WSDL is an acronym for Web Services Description Language.

WSDL is a xml document containing information about web services such as method name, method parameter and how to access it.

WSDL is a part of UDDI. It acts as a interface between web service applications.

WSDL is pronounced as wiz-dull.

## UDDI

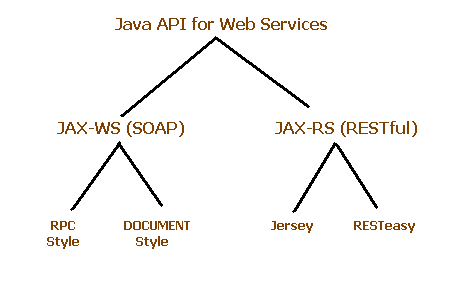
UDDI is an acronym for Universal Description, Discovery and Integration.

UDDI is a XML based framework for describing, discovering and integrating web services.

UDDI is a directory of web service interfaces described by WSDL, containing information about web services.

## Understanding SOAP and REST

Web Services are mainly of 2 types, **SOAP** [Simple Object Access Protocol] and **REST** [Representational state transfer] based services. We have different type of specifications to implement SOAP and REST services. I believe so far you might be in confusion with these kind keywords like, JAX-RS, JAX-WS, RESTful, SOAP, Apache Axis2, Apache CXF bla bla…  Let me try to bring you out of them.



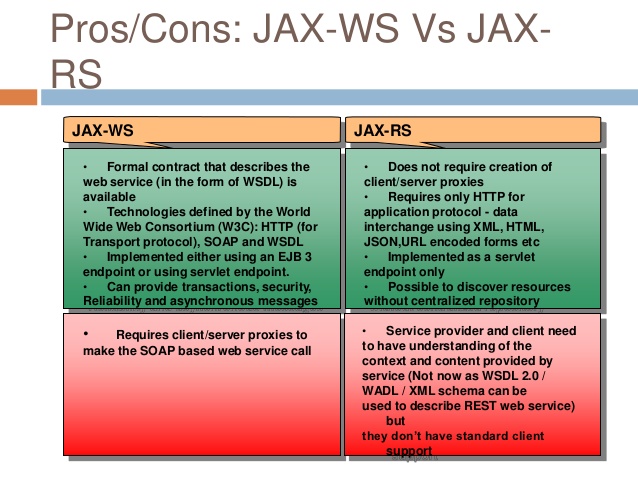
**jax**-**ws**.java.net. The Java API for XML Web Services (**JAX**-**WS**) is a Java programming language API for creating web services, particularly SOAP services.

* JAX-RS provides the implementation of RESTful web services,  JAX-RS is a specification for RESTful Web Services with Java and it is given by Sun.  Since it is a specification, other frameworks can be written to implement these specifications, and that includes Jersey from Oracle, Resteasy from Jboss, CXF from Apache bla bla.

**Jersey** RESTful Web Services framework is an open source framework for developing RESTful Web Services in Java. It provides support for **JAX**-**RS** APIs and serves as a **JAX**-**RS** (JSR 311 & JSR 339) Reference Implementation.

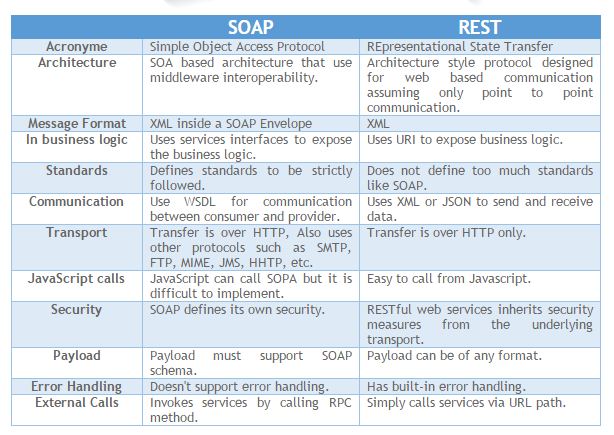
* JAX-WS, Apache Axis2 provides the implementation for SOAP
* Apache CXF provides implementation for SOAP and RESTful services both.

# Differences between JAX-WS Vs JAX-RS

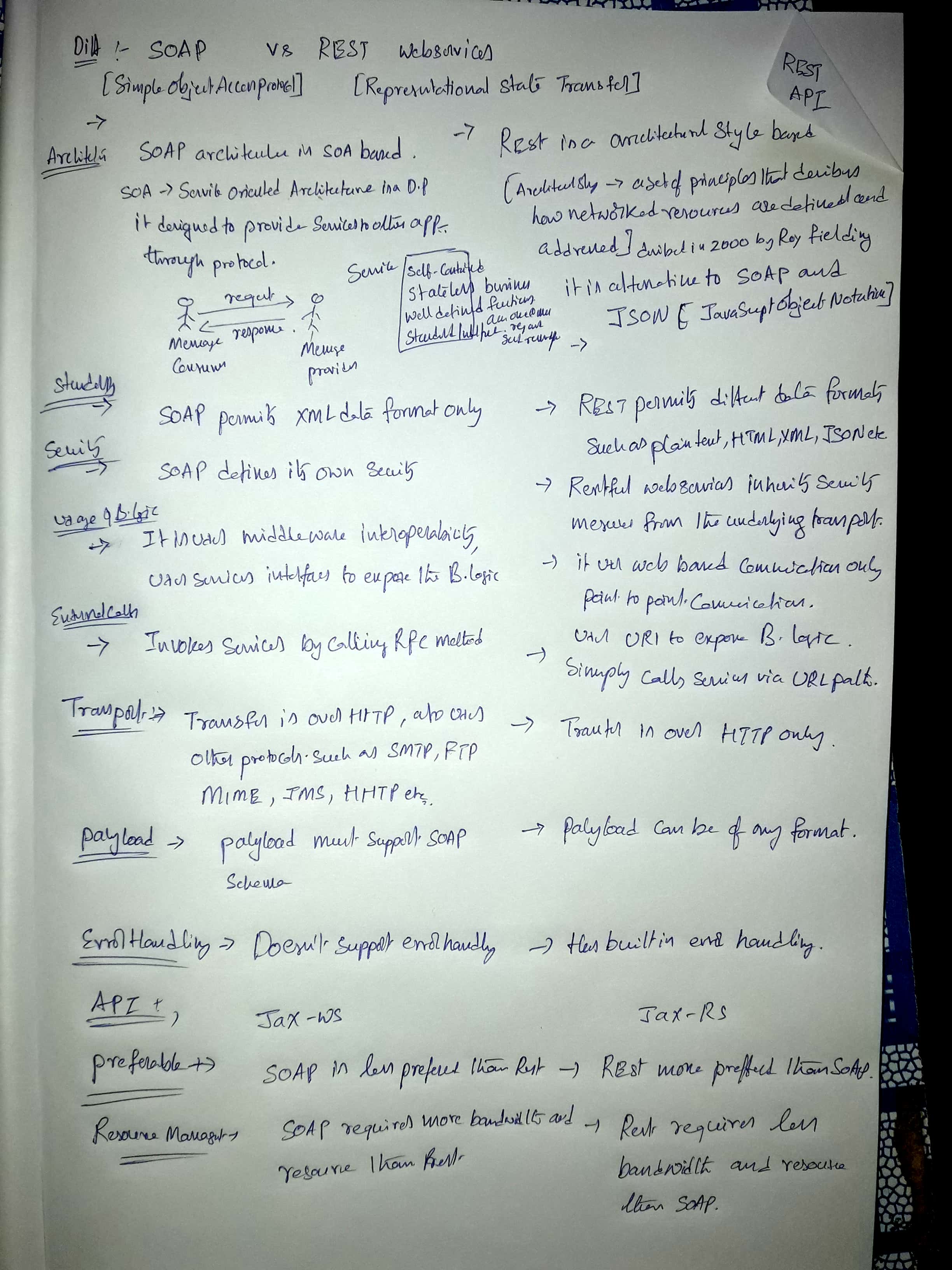




SOAP vs REST Web Services



|  |  |  |
| --- | --- | --- |
| **No.** | **SOAP** | **REST** |
| Acronym | SOAP stands for **Simple Object Access Protocol**. | REST stands for **REpresentational State Transfer**. |
| architecture | SOAP is a **protocol**. | REST is an **architectural style**. |
| usage | SOAP **can't use REST** because it is a protocol. | REST **can use SOAP** web services because it is a concept and can use any protocol like HTTP, SOAP. |
| In Business logic | SOAP **uses services interfaces to expose the business logic**. | REST **uses URI to expose business logic**. |
| external calls | **Invokes services by calling RPC methods** | **Simply call services via URL path** |
| Transport | **Transport over HTTP, Also uses other protocols such as FTP,MIME,JMS, etc...** | **Transport is over HTTP only.** |
| Api | **JAX-WS** is the java API for SOAP web services. | **JAX-RS** is the java API for RESTful web services. |
| standards | SOAP **defines standards**to be strictly followed. | REST does not define too much standards like SOAP. |
| Resource requirement | SOAP **requires more bandwidth** and resource than REST. | REST **requires less bandwidth** and resource than SOAP. |
| Security | SOAP **defines its own security**. | RESTful web services **inherits security measures** from the underlying transport. |
| dataformat | SOAP **permits XML** data format only. | REST **permits different** data format such as Plain text, HTML, XML, JSON etc. |
| Preferable | SOAP is **less preferred** than REST. | REST **more preferred** than SOAP. |



## RESTful

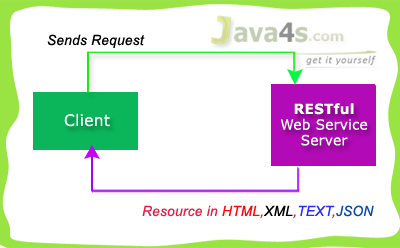
What ever the data/response we will get from the server is known as ***Resource*** [remember this point], Each resource can be accessed by its URI’s.  We can get the resource from RESTful service in differentformats like, HTML,XML,JSON,TEXT,PDF and in the Image formats as well, but in real time we mainly we will prefer JSON.  REST guidelines always talks about stateless communication between client and the Server.  Stateless means, every single request from client to server will be considered as a fresh request. Because of this reason REST always prefers to choose HTTP as it a stateless protocol.

In stateless protocol there is no record of the state is saved at server end.Client send request to the server and server response back according to current state. A stateless server does not keeps state between connections.So,When you send a request to a stateless server, it does not create any objects that track information regarding your requests. ex.UDP , HTTP etc.

On the other hand in statefull protocol there is tight dependency between client and server . If client send a request to the server then it expects some kind of response, if it does not get any responce then resend request.A stateful server keeps state of connections. ex.FTP , Talnet.

Even you can force to make stateless behave like stateful if need by keep data using session , cookies.

A bit of information may be helpfull !



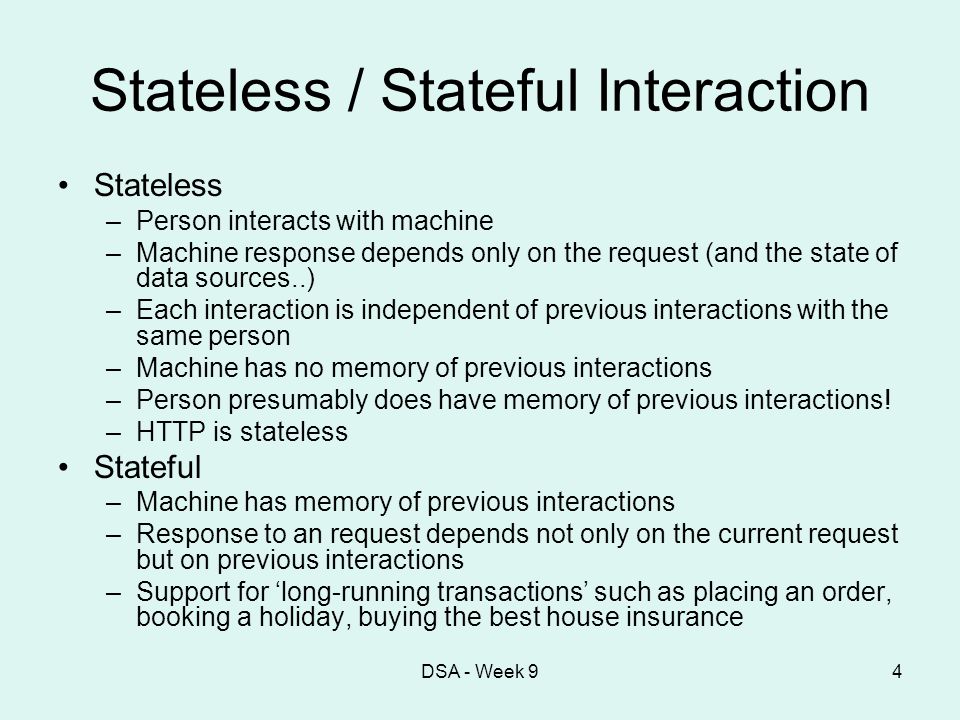
RESTful used 4 main HTTP methods…

* **GET** - Retrieve Data
* **POST**- Create/Insert Data
* **PUT**- Update Data
* **DELETE**- Delete Data

Generally we will prefer RESTful Services in these scenarios…

* If clients require caching, means if you have limited bandwidth
* If you want every thing to be stateless [ I have already explained about stateless ]

But SOAP gives the output only in XML format.   Hope you are good now :-) by the way we are going to use Jersey to implement JAX-RS specifications.

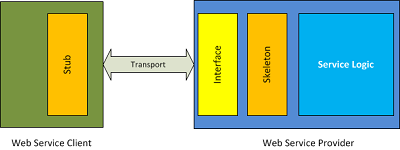


In stateless protocol there is no record of the state is saved at server end.Client send request to the server and server response back according to current state. A stateless server does not keeps state between connections.So,When you send a request to a stateless server, it does not create any objects that track information regarding your requests. ex.UDP , HTTP etc.

On the other hand in statefull protocol there is tight dependency between client and server . If client send a request to the server then it expects some kind of response, if it does not get any responce then resend request.A stateful server keeps state of connections. ex.FTP , Talnet.

Even you can force to make stateless behave like stateful if need by keep data using session , cookies.

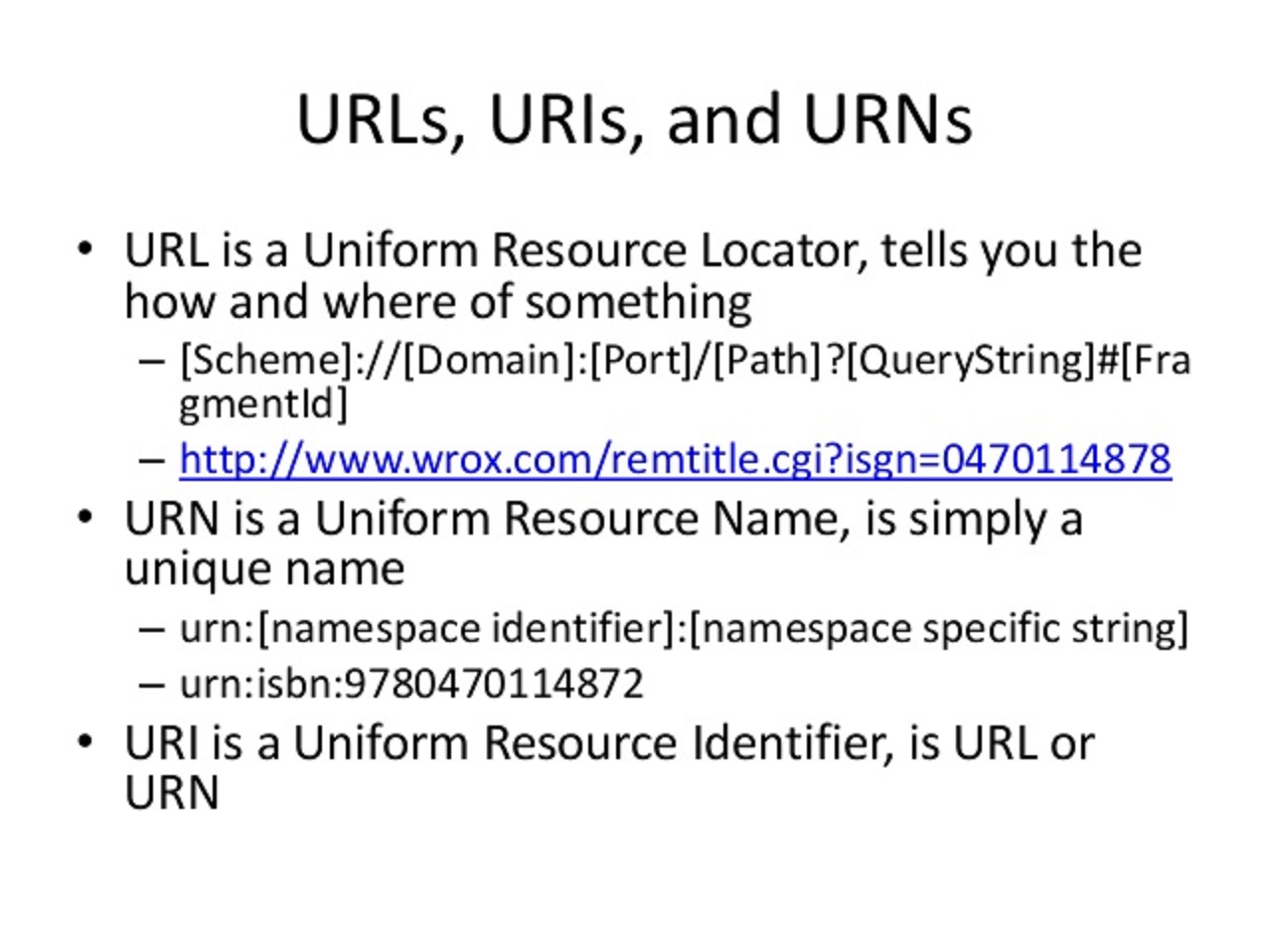
A bit of information may be helpfull !

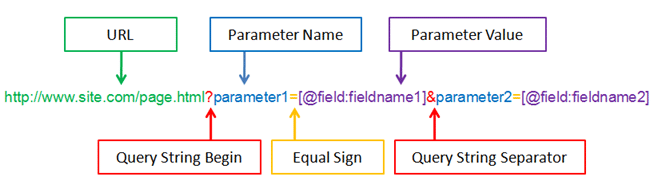


Generally,

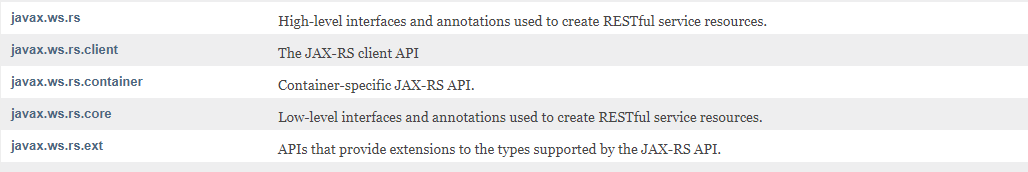
* Application to application communication.
* Interoperability between disparate systems.
* Communication over network.
* Exposed interface is platform independent and internal implementation is abstracted.
* Enables loosely coupled design.
* Open protocol is used for establishing communication.
* Web services are self contained.

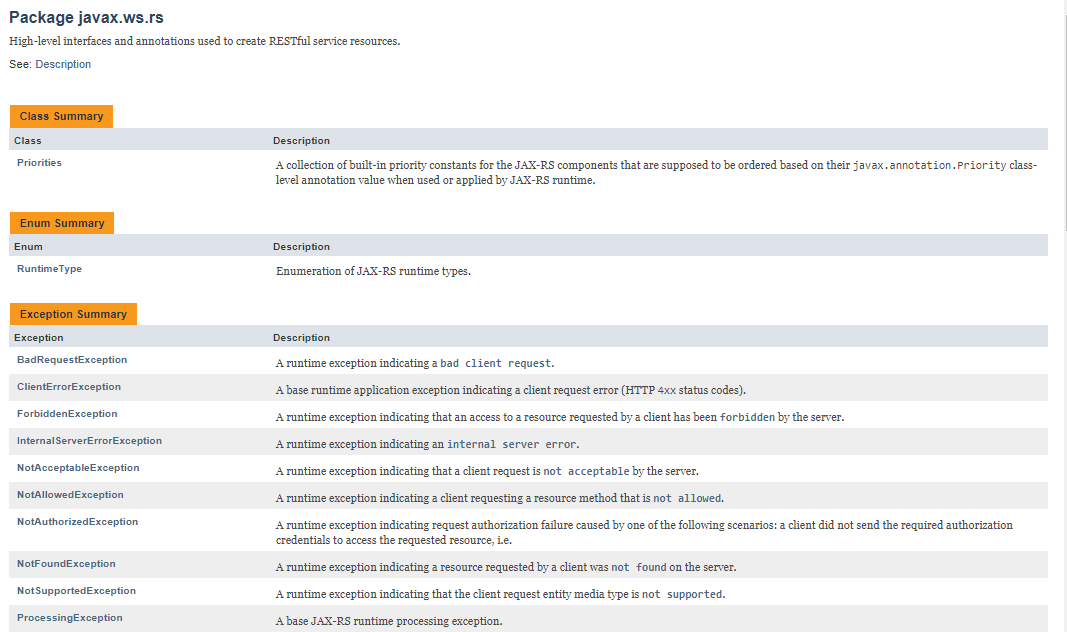
### URI & URL

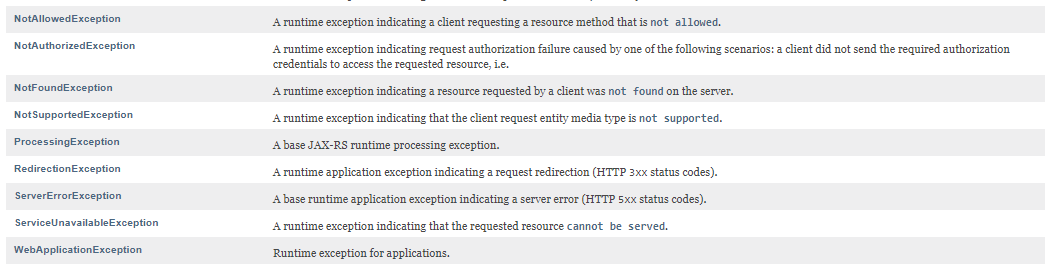
1

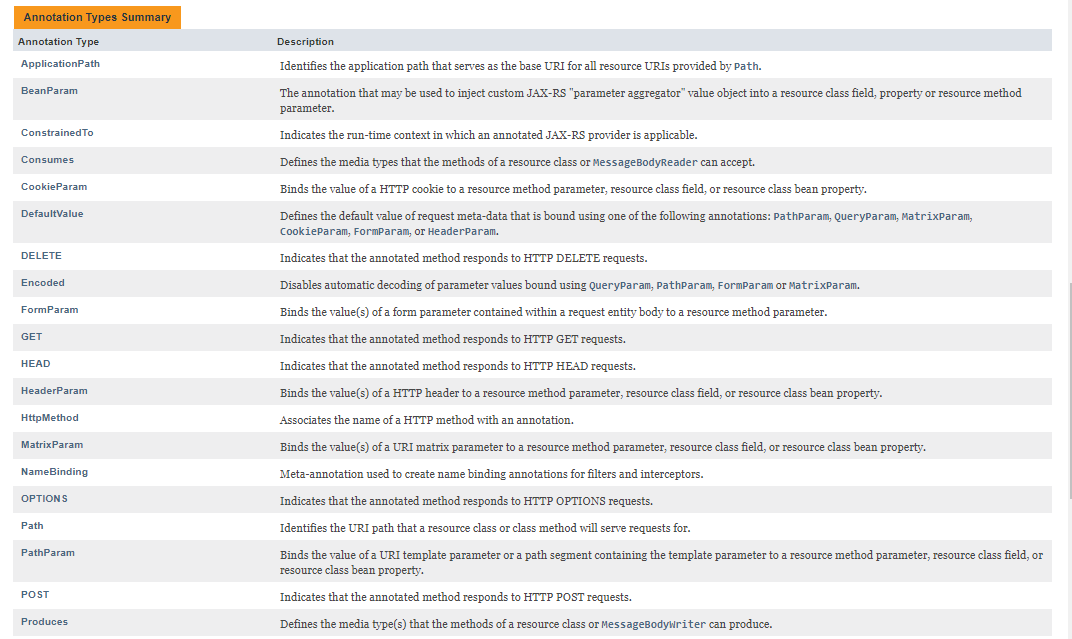


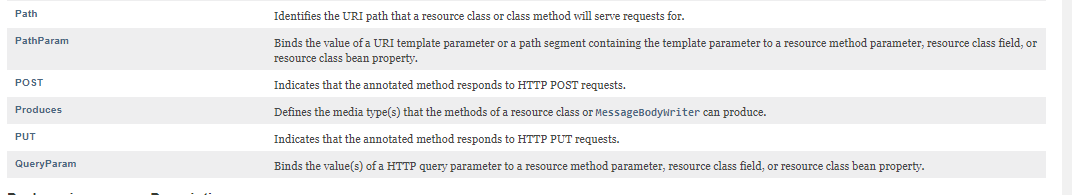
RestFul API

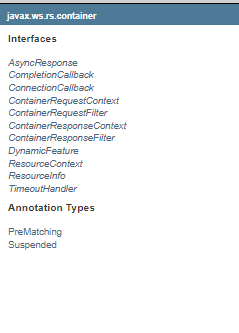
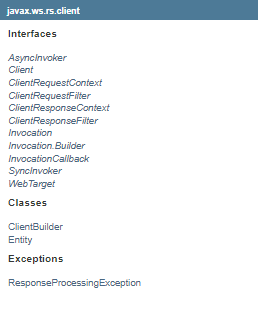


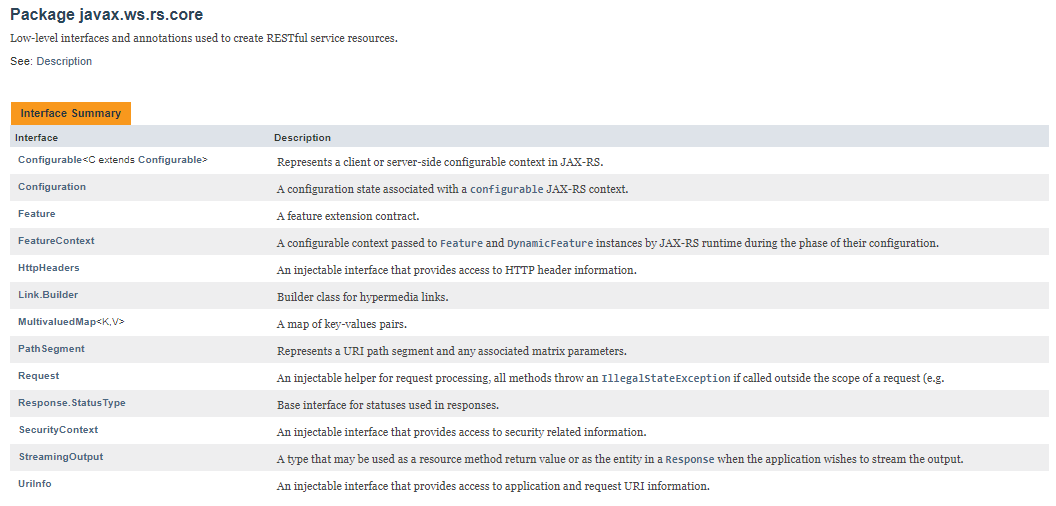
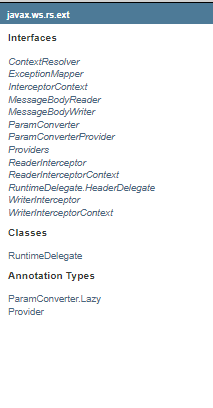


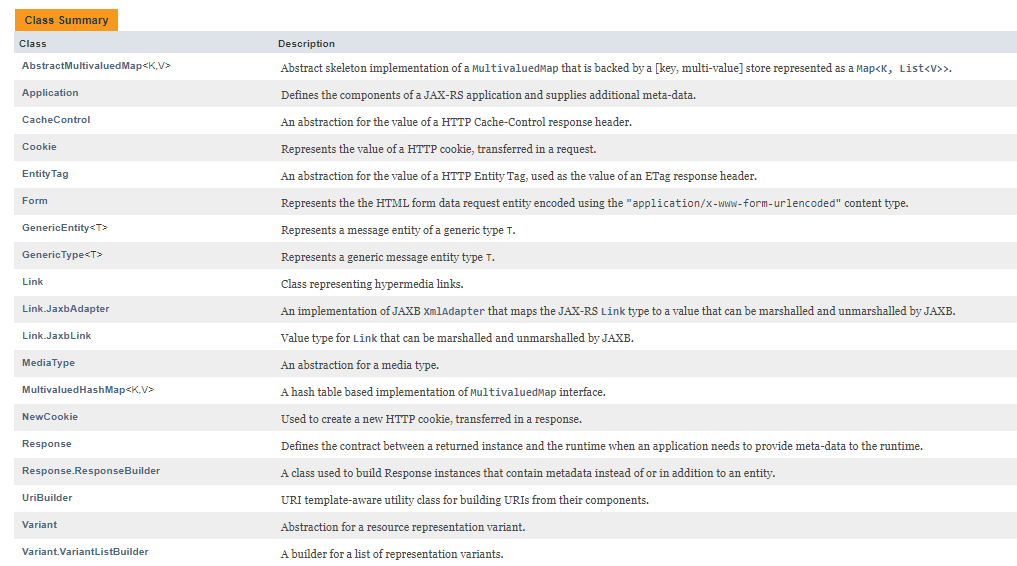


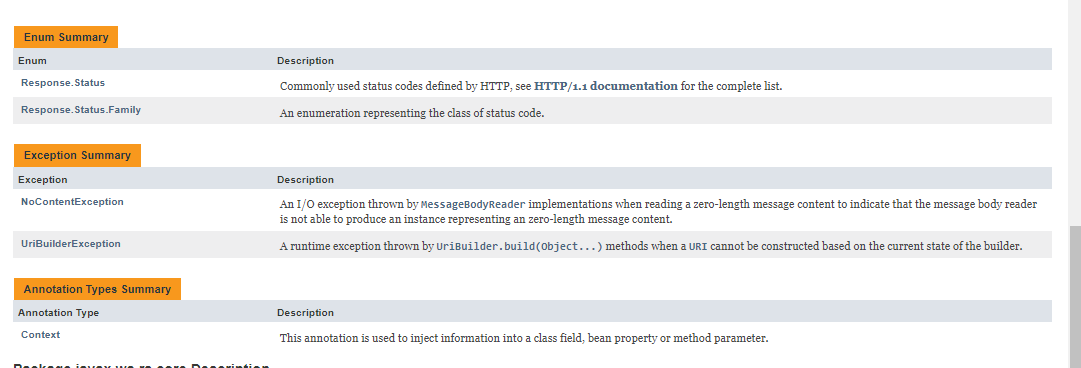












RESTful Web Services (JAX-RS) Annotations

## JAX-RS Annotations

* @Path(‘Path‘)
* @GET
* @POST
* @PUT
* @DELETE
* @Produces(MediaType.TEXT\_PLAIN [, more-types])
* @Consumes(type[, more-types])
* @PathParam()
* @QueryParam()
* @MatrixParam()
* @FormParam()

## @Path() Annotation

* Its a Class & Method level of annotation
* This will check the path next to the base URL

**Syntax** :  
Base URL :  
http://localhost:(port)/<YourApplicationName>/<UrlPattern In Web.xml>/<path>  
Here <path> is the part of URI, and this will be identified by @path annotation at class/method level, you will be able to understand in the next RESTful  hello world tutorial.

## @GET

Its a method level of annotation, this annotation indicates that the following method should respond to the HTTP GET request only,  i mean if we annotate our method with @GET, the execution flow will enter that following method if we send GET request from the client

## @POST

Its a method level of annotation, this annotation indicates that the following method should respond to the HTTP POST request only.

## @PUT

Its a method level of annotation, this annotation indicates that the following method should respond to the HTTP PUT request only.

## @DELETE

Its a method level of annotation, this annotation indicates that the following method should respond to the HTTP DELETE request only.

## @Produces

Its a method or field level annotation, This tells which MIME type is delivered by the method annotated with @GET.  I mean when ever we send a HTTP GET request to our RESTful service, it will invokes particular method and produces the output in different formats.  There you can specifies in what are all formats (MIME) your method can produce the output, by using @produces annotation.  
**Remember**: We will use @Produces annotation for GET requests only.

## @Consumes

This is a class and method level annotation, this will define which MIME type is consumed by the particular method. I mean in which format the method can accept the input from the client.

Will discuss later regarding @PathParam, @QueryParam, @MatrixParam, @FormParam annotations :-) , i will talk more about these annotations in the next examples.

How RESTful Web Services Extract Input Parameters

[Web Services](http://www.java4s.com/web-services/) »on Jul 6, 2014 [**{ 8 Comments }**](http://www.java4s.com/web-services/how-restful-web-services-extract-input-parameters/#comments) By [Sivateja](https://plus.google.com/118054670710951892925?rel=author)

In this article i will show you how a RESTful web service will  extract input parameters from the clientrequest.  We have different ways of sending input values to the rest services, and RESTful web service extract those details based upon the client URL pattern. In JAX-RS we can use the following annotations to extract the input values sent by the client.

* @PathParam
* @QueryParam
* @MatrixParam
* @FormParam

@PathParam,@QueryParam,@MatrixParam are parameter annotations which allows us to map variable URI path fragments into your method call. Confused ? :-) In simple words, these three annotations will come into picture in case if we are passing the input values to the restful service through the URL. After that Rest service will extract those values by using these annotations. Regarding @FormParam, restful web service will use this annotation to retrieve the values sent by the client through some HTML/JSP form.

## @PathParam URL Syntax

http://localhost:7001/<Rest Service Name>/rest/customers/100/Java4s

Did you observe the two parameters appear in the end of the above URL [100 & Java4s], which are separated by forward slash(/) are called as path parameters, as of now just remember the syntax, going forward i will give you an example on each annotation.

## @QueryParam URL Syntax

http://localhost:7001/…/rest/customers?custNo=100&custName=Java4s

If the client sends an input in the form of query string in the URL, then those parameters are called as Query Parameters.  If you observe the above syntax, client passing 2 parameters 100 and Java4s  started after question mark (?) symbol and each parameter is separated by & symbol,  those parameters are called as query parameters.

## @MatrixParam URL Syntax

http://localhost:7001/…/rest/customers;custNo=100;custName=Java4s

Matrix parameters are  another way defining the parameters to be added to URL.  If you observe the above syntax, client is passing two parameters each are separated by semicolon, these parameters are called as matrix parameters.  Remember these parameters may appear any where in the path.

## @FormParam URL Syntax

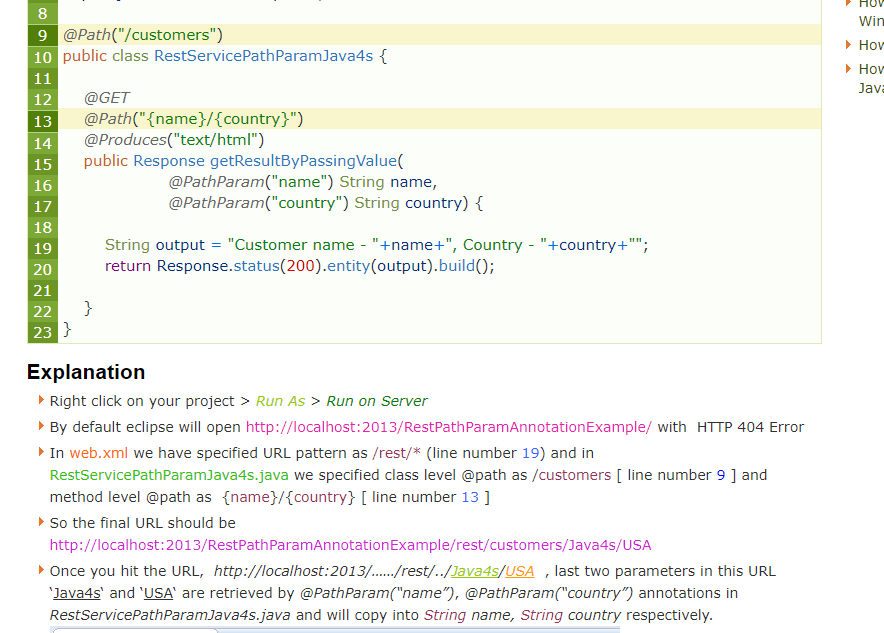
Finally form parameters,  if we have a HTML form having two input fields and submit button. Lets client enter those details and submit to the RESTful web service. Then the rest service will extract those details by using this @FormParam annotation.  
For now just remember these consents, going forward i will give you an example on each annotation.

Jersey Hello World Example Using JAX-RS Specification





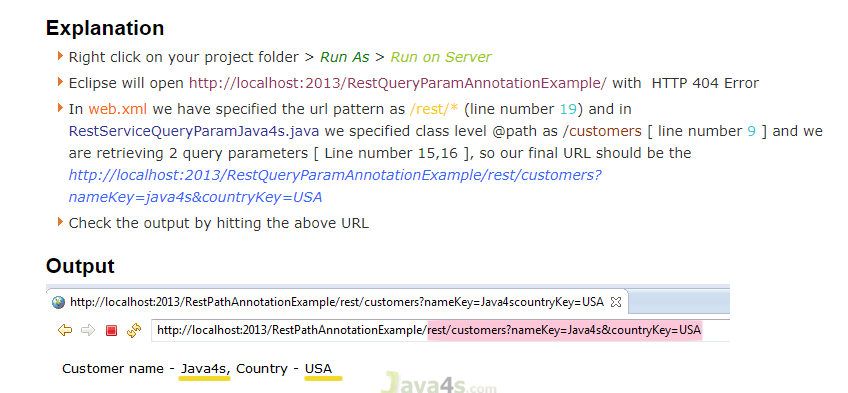
RESTful Web Services (JAX-RS) @PathParam Example



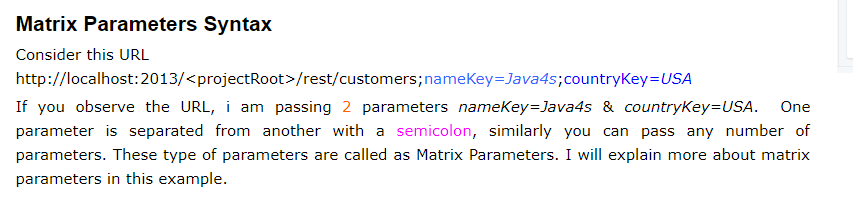


RESTful Web Services (JAX-RS) @QueryParam Example

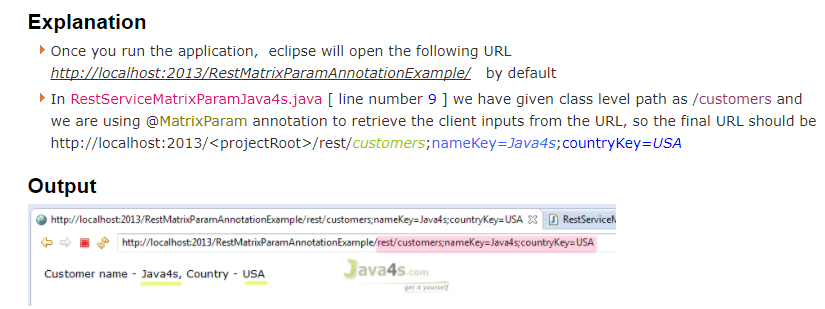




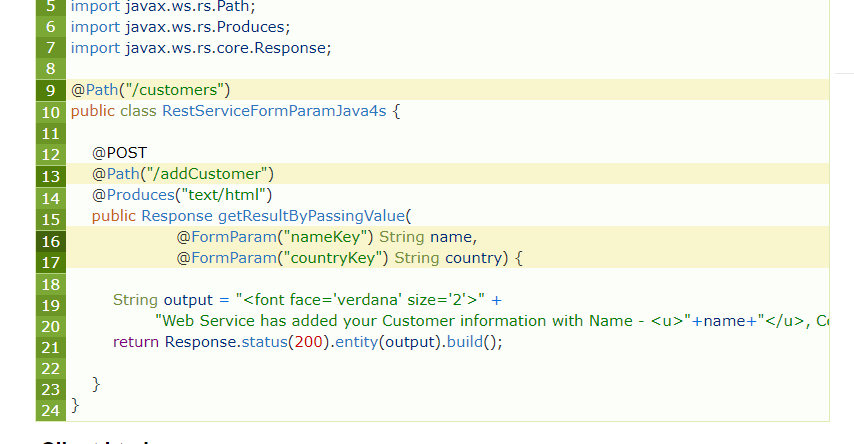
RESTful Web Services (JAX-RS) @MatrixParam Example







RESTful Web Services (JAX-RS) @FormParam Example



<form action="http://localhost:2013/RestFormParamAnnotationExample/rest/customers/addCustomer" method="post">

<table>

<tr>

<td><font face="verdana" size="2px">Customer Name : </font></td>

<td><input type="text" name="nameKey" /> </td>

</tr>

<tr>

<td><font face="verdana" size="2px">Country</font></td>

<td> <input type="text" name="countryKey" /> </td>

</tr>

<tr>

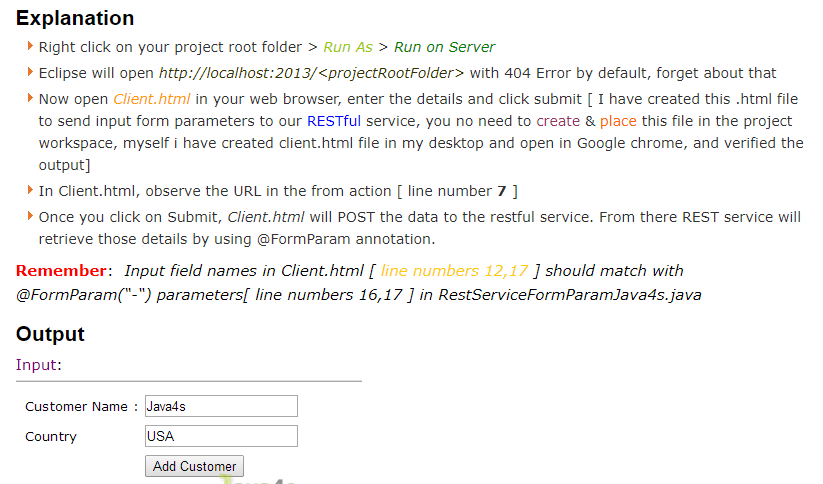
<td></td>

<td><input type="submit" value="Add Customer" /> </td>

</tr>

</table>

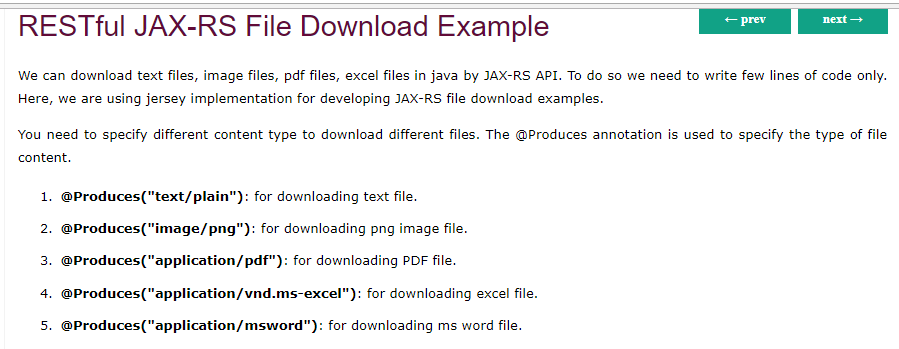
</form>



# RestFul Jax-RS File Download Example:

# Restful MiMe/Mediatypes

@Produces(MediaType.APPLICATION\_JSON)



# What is a Resource?

REST architecture treats every content as a resource. These resources can be Text Files, Html Pages, Images, Videos or Dynamic Business Data. REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ Global IDs. REST uses various representations to represent a resource where Text, JSON, XML. The most popular representations of resources are XML and JSON.

**Representation of Resources**

A resource in REST is a similar Object in Object Oriented Programming or is like an Entity in a Database. Once a resource is identified then its representation is to be decided using a standard format so that the server can send the resource in the above said format and client can understand the same format.

For example, in [RESTful Web Services - First Application](https://www.tutorialspoint.com/restful/restful_first_application.htm) chapter, a user is a resource which is represented using the following XML format −

<user>

<id>1</id>

<name>Mahesh</name>

<profession>Teacher</profession>

</user>

The same resource can be represented in JSON format as follows −

{

"id":1,

"name":"Mahesh",

"profession":"Teacher"

}

**Good Resources Representation**

REST does not impose any restriction on the format of a resource representation. A client can ask for JSON representation whereas another client may ask for XML representation of the same resource to the server and so on. It is the responsibility of the REST server to pass the client the resource in the format that the client understands.

Following are some important points to be considered while designing a representation format of a resource in RESTful Web Services.

**Understandability** − Both the Server and the Client should be able to understand and utilize the representation format of the resource.

**Completeness** − Format should be able to represent a resource completely. For example, a resource can contain another resource. Format should be able to represent simple as well as complex structures of resources.

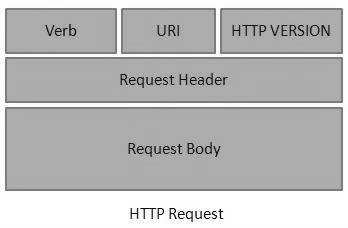
**Linkablity** − A resource can have a linkage to another resource, a format should be able to handle such situations.

However, at present most of the web services are representing resources using either XML or JSON format. There are plenty of libraries and tools available to understand, parse, and modify XML and JSON data.

RESTful Web Services - Messages

RESTful Web Services make use of HTTP protocols as a medium of communication between client and server. A client sends a message in form of a HTTP Request and the server responds in the form of an HTTP Response. This technique is termed as Messaging. These messages contain message data and metadata i.e. information about message itself. Let us have a look on the HTTP Request and HTTP Response messages for HTTP 1.1.

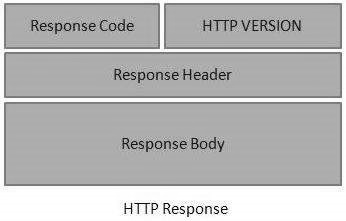
## HTTP Request



An HTTP Request has five major parts −

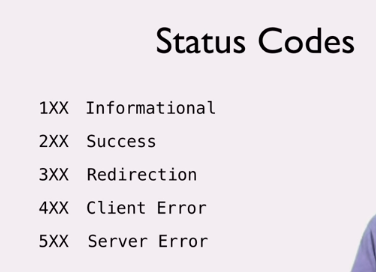
* **Verb** − Indicates the HTTP methods such as GET, POST, DELETE, PUT, etc.
* **URI** − Uniform Resource Identifier (URI) to identify the resource on the server.
* **HTTP Version** − Indicates the HTTP version. For example, HTTP v1.1.
* **Request Header** − Contains metadata for the HTTP Request message as key-value pairs. For example, client (or browser) type, format supported by the client, format of the message body, cache settings, etc.
* **Request Body** − Message content or Resource representation.

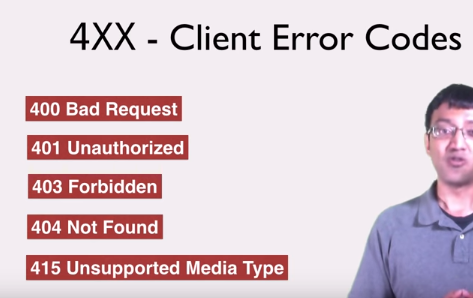
## HTTP Response

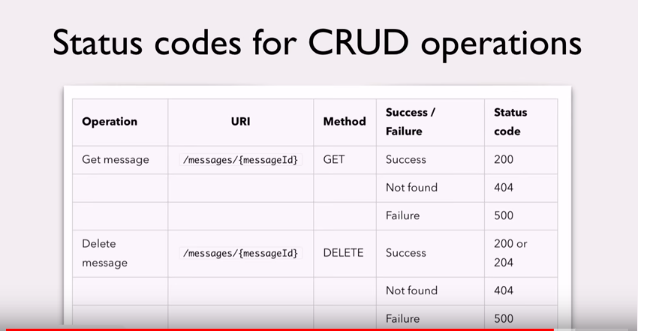


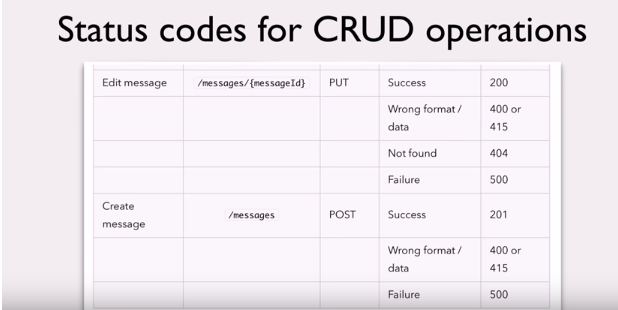
An HTTP Response has four major parts −

* **Status/Response Code** − Indicates the Server status for the requested resource. For example, 404 means resource not found and 200 means response is ok.





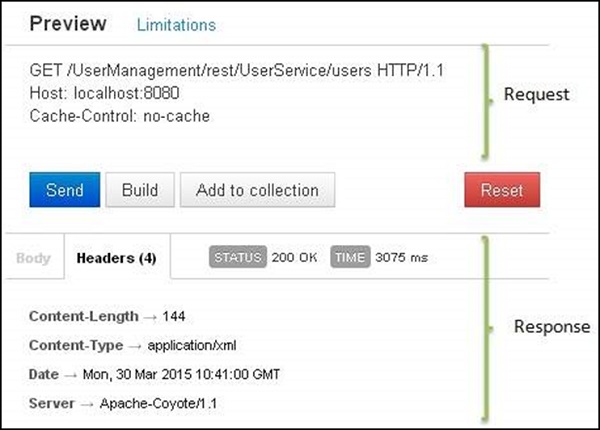




* **HTTP Version** − Indicates the HTTP version. For example HTTP v1.1.
* **Response Header** − Contains metadata for the HTTP Response message as keyvalue pairs. For example, content length, content type, response date, server type, etc.
* **Response Body** − Response message content or Resource representation.

Example

As we have explained in the [RESTful Web Services - First Application chapter](https://www.tutorialspoint.com/restful/restful_first_application.htm), let us put http://localhost:8080/UserManagement/rest/UserService/users in the POSTMAN with a GET request. If you click on the Preview button which is near the send button of Postman and then click on the Send button, you may see the following output.



# RESTful Web Service (JAX-RS) JSON Example Using Jersey

These steps are mandatory in order to make Jersey to support with JSON mappings.

* Apart from existing dependencies, add ‘jersey-json.jar‘ to your Maven pom.xml which includes all Jackson and other JSON supporting libraries



|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>                  <groupId>com.sun.jersey</groupId>                  <artifactId>jersey-json</artifactId>                  <version>1.8</version>  </dependency> |

* In web.xml add “com.sun.jersey.api.json.POJOMappingFeature” as “init-param”
* In the web service class, we need to annotate the method with @Produces(MediaType.APPLICATION\_JSON). By doing so we are instructiong the service method that we are expecting the JSON output, thats it jersey will take care rest of the things

**Note**: In previous examples i used Tomcat 6 and JDK 1.6 but for this [JAX-RS JSON Example] example i have used JDK 1.7.

Required Files

* pom.xml
* web.xml
* Customer.java
* JsonFromRestful.java

pom.xml



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">    <modelVersion>4.0.0</modelVersion>    <groupId>JsonFromRestfulWebServices</groupId>    <artifactId>JsonFromRestfulWebServices</artifactId>    <version>0.0.1-SNAPSHOT</version>    <packaging>war</packaging>        <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>junit</groupId>                  <artifactId>junit</artifactId>                  <version>4.8.2</version>                  <scope>test</scope>             </dependency>               <dependency>                  <groupId>com.sun.jersey</groupId>                  <artifactId>jersey-server</artifactId>                  <version>1.8</version>             </dependency>               <dependency>                  <groupId>com.sun.jersey</groupId>                  <artifactId>jersey-json</artifactId>                  <version>1.8</version>            </dependency>       </dependencies>        <build>         <finalName>JsonFromRestfulWebServices</finalName>         <plugins>            <plugin>                <artifactId>maven-compiler-plugin</artifactId>                   <configuration>                      <compilerVersion>1.5</compilerVersion>                      <source>1.5</source>                      <target>1.5</target>                   </configuration>             </plugin>         </plugins>      </build>  </project> |

 web.xml



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | <?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/j2ee" xmlns:javaee="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd" id="WebApp\_ID" version="2.4">    <display-name>JsonFromRestfulWebServices</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>java4s</param-value>          </init-param>          <init-param>            <param-name>com.sun.jersey.api.json.POJOMappingFeature</param-name>            <param-value>true</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> |

 Customer.java



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | package java4s;    public class Customer {        private int custNo;      private String custName;      private String custCountry;        public int getCustNo() {          return custNo;      }      public void setCustNo(int custNo) {          this.custNo = custNo;      }      public String getCustName() {          return custName;      }      public void setCustName(String custName) {          this.custName = custName;      }      public String getCustCountry() {          return custCountry;      }      public void setCustCountry(String custCountry) {          this.custCountry = custCountry;      }  } |

JsonFromRestful.java



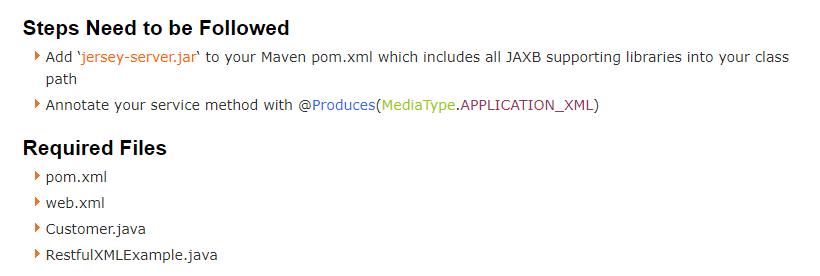
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | package java4s;    import javax.ws.rs.GET;  import javax.ws.rs.Path;  import javax.ws.rs.PathParam;  import javax.ws.rs.Produces;  import javax.ws.rs.core.MediaType;    @Path("/customers")  public class JsonFromRestful {        @GET      @Path("/{cusNo}")      @Produces(MediaType.APPLICATION\_JSON)      public Customer produceCustomerDetailsinJSON(@PathParam("cusNo") int no) {            /\*           \* I AM PASSING CUST.NO AS AN INPUT, SO WRITE SOME BACKEND RELATED STUFF AND           \* FIND THE CUSTOMER DETAILS WITH THAT ID. AND FINALLY SET THOSE RETRIEVED VALUES TO           \* THE CUSTOMER OBJECT AND RETURN IT, HOWEVER IT WILL RETURN IN JSON FORMAT :-)           \* \*/            Customer cust = new Customer();              cust .setCustNo(no);              cust .setCustName("Java4s");              cust .setCustCountry("India");          return cust;      }  } |

Output



JAX-RS XML Example With JAXB Using Jersey

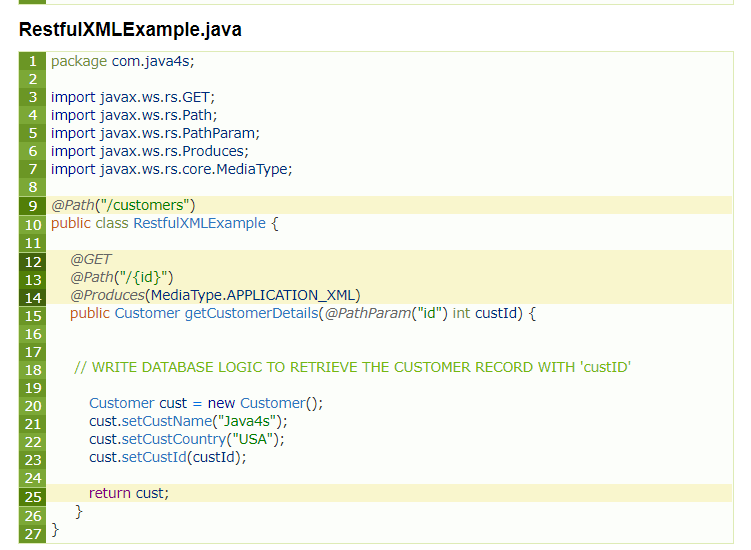
n this article i will give you an example on how a RESTful web service produces XML response using Jersey. Basically JAX-RS supports conversion of java objects into XML with the help of JAXB. As Jersey it self contains JAXB libraries we no need to worry about JAXB-Jersey integration stuff.













# Exception Handling in Restful webservies:

Errors can be reported to a client either by creating and returning the appropriate **Response** object or by throwing an exception.

Application code is allowed to throw any checked (classes extending **java.lang.Exception**) or unchecked (classes extending **java.lang.RuntimeException**) exceptions they want

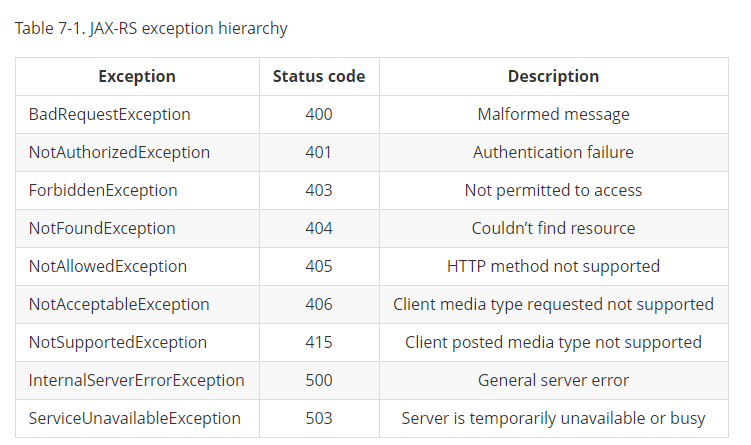
Thrown exceptions are handled by the JAX-RS runtime if you have registered an exception mapper

Exception mappers can convert an exception to an HTTP response. If the thrown exception is not handled by a mapper, it is propagated and handled by the container (i.e., servlet) JAX-RS is running within.

 JAX-RS also provides the **javax.ws.rs.WebApplicationException**

 implement and register instances of **javax.ws.rs.ext.ExceptionMapper**.

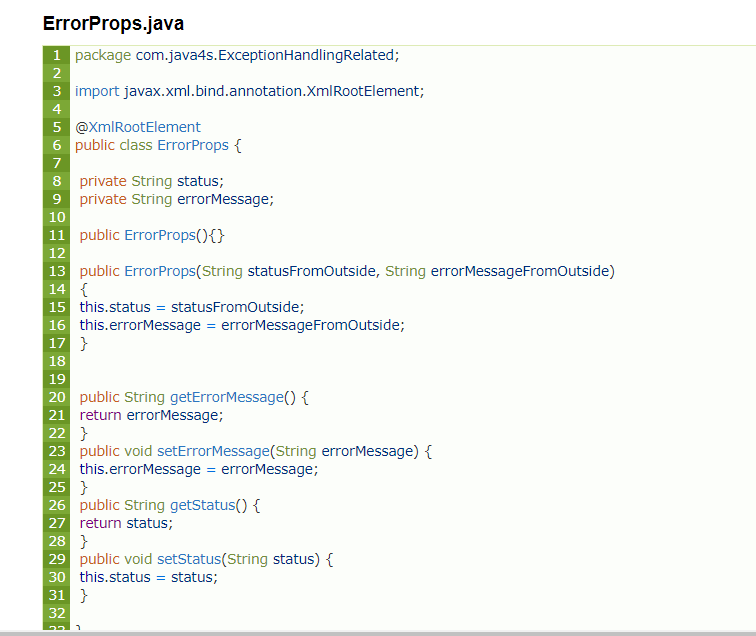
These objects know how to map a thrown application exception to a **Response** object:



**javax.ws.rs.ext.ExceptionMapper**.







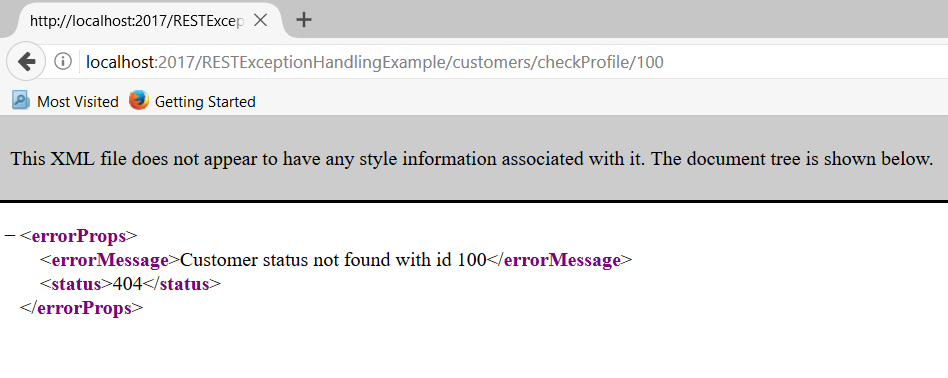
## Explanation

* Created an exception mapper CustomerDataNotFoundExceptionMapper for CustomerDataNotFoundException
* All exception mappers should implement ExceptionMapper interface of type generic, for now I am going to use this exception mapper only for our exception, so I have implemented ExceptionMapperof type CustomerDataNotFoundException [check at line number 9]
* We need to override the toResponse method of ExceptionMapper interface, which takes exception as an argument, in this case CustomerDataNotFoundException
* I want to display my exception details as an XML, so created a simple java model ErrorProps.java and annotated with @XmlRootElement
* Now come back to mapper class toResponse method, there I am returning Response object

Response.status( – ) :- setting the current status  
.entity( – ) :- passing ErrorProps class object by setting required values, here I am setting status as 404, and our custom exception message

* Finally annotate our mapper class with @Provider annotation, so that JAX-RS will register this mapper to intercept the response when particular exception was thrown

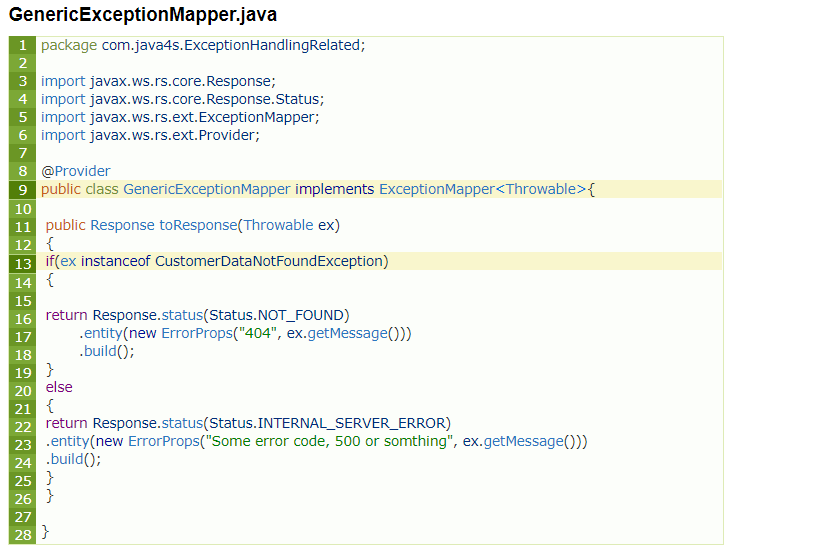
Go ahead and run the application and see…

[](http://www.java4s.com/wp-content/uploads/2017/05/jax-rs-exceptions-in-xml.PNG)

We did it :-)

But this is only for NullPointerException, but how about the other exceptions ? for that we need to modify the mapper. Let me do it by creating new mapper class.

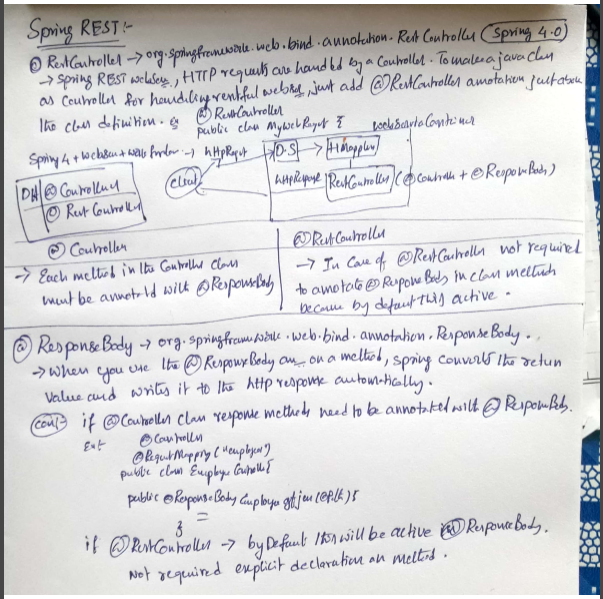
## GenericExceptionMapper.java

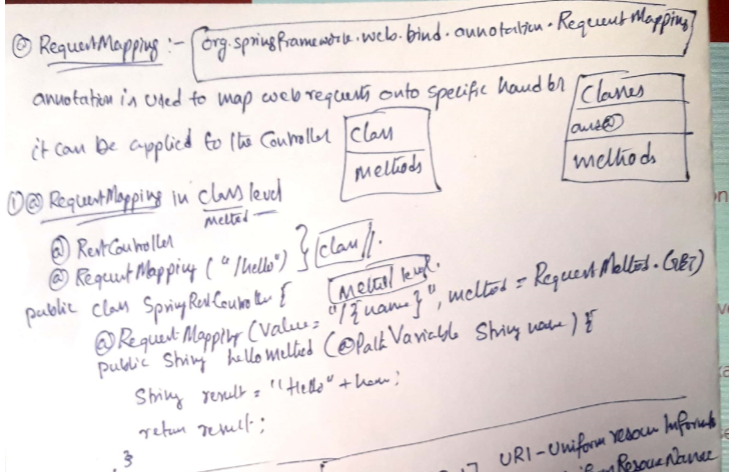


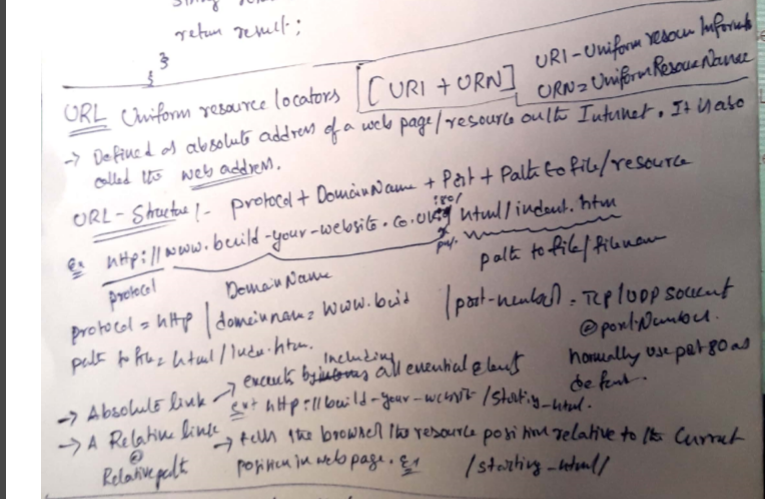
## What are the changes ?

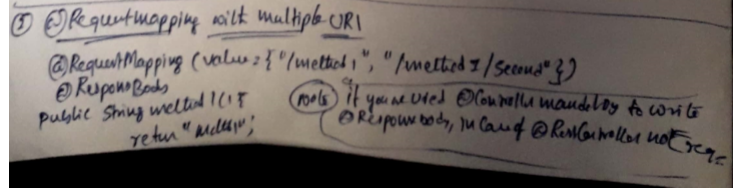
Implement ExceptionMapper of type Throwable, instead of our own exception. If you check the above class, at line number 13, I am checking whether the Throwable is the instance of CustomerDataNotFoundException, if its true, I will build my Response accordingly, like this you can handle any number of exceptions in a single class, you can download and play with it.

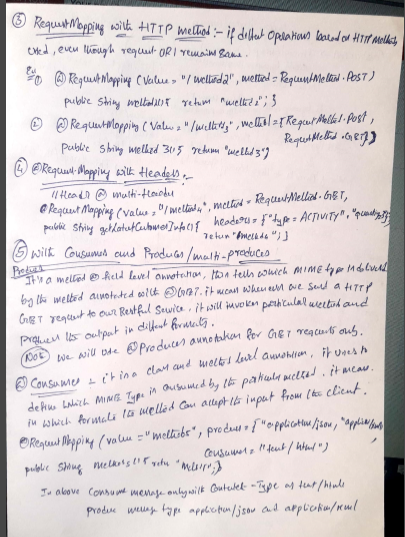
# SPRING RESTFUL WebServices

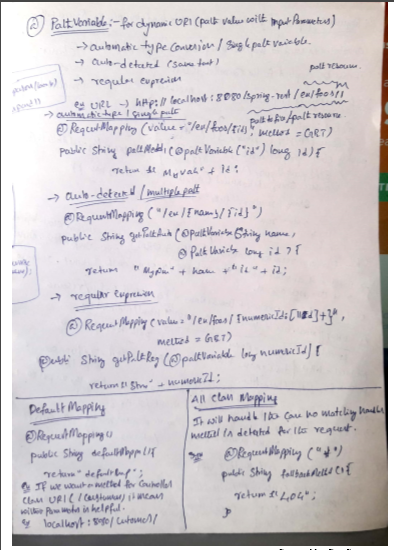


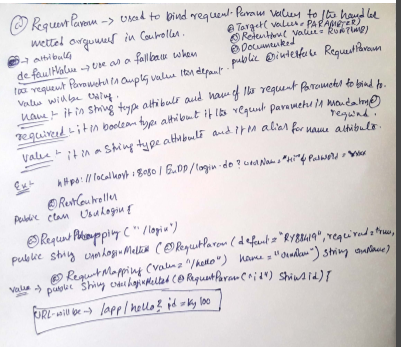












### Spring Restful web service End Points

We will have following rest web services end points.

|  |  |  |  |
| --- | --- | --- | --- |
| SL. NO | URI | HTTP METHOD | DETAILS |
| 1 | /rest/emp/dummy | GET | Health Check service, to insert a dummy data in the Employees data storage |
| 2 | /rest/emp/{id} | GET | To get the Employee object based on the id |
| 3 | /rest/emps | GET | To get the list of all the Employees in the data store |
| 4 | /rest/emp/create | POST | To create the Employee object and store it |
| 5 | /rest/emp/delete/{id} | PUT | To delete the Employee object from the data storage based on the id |

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.ResponseBody;

import com.journaldev.spring.model.Employee;

/\*\*

\* Handles requests for the Employee service.

\*/

@Controller

public class EmployeeController {

private static final Logger logger = LoggerFactory.getLogger(EmployeeController.class);

//Map to store employees, ideally we should use database

Map<Integer, Employee> empData = new HashMap<Integer, Employee>();

@RequestMapping(value = EmpRestURIConstants.DUMMY\_EMP, method = RequestMethod.GET)

public @ResponseBody Employee getDummyEmployee() {

logger.info("Start getDummyEmployee");

Employee emp = new Employee();

emp.setId(9999);

emp.setName("Dummy");

emp.setCreatedDate(new Date());

empData.put(9999, emp);

return emp;

}

@RequestMapping(value = EmpRestURIConstants.GET\_EMP, method = RequestMethod.GET)

public @ResponseBody Employee getEmployee(@PathVariable("id") int empId) {

logger.info("Start getEmployee. ID="+empId);

return empData.get(empId);

}

@RequestMapping(value = EmpRestURIConstants.GET\_ALL\_EMP, method = RequestMethod.GET)

public @ResponseBody List<Employee> getAllEmployees() {

logger.info("Start getAllEmployees.");

List<Employee> emps = new ArrayList<Employee>();

Set<Integer> empIdKeys = empData.keySet();

for(Integer i : empIdKeys){

emps.add(empData.get(i));

}

return emps;

}

@RequestMapping(value = EmpRestURIConstants.CREATE\_EMP, method = RequestMethod.POST)

public @ResponseBody Employee createEmployee(@RequestBody Employee emp) {

logger.info("Start createEmployee.");

emp.setCreatedDate(new Date());

empData.put(emp.getId(), emp);

return emp;

}

@RequestMapping(value = EmpRestURIConstants.DELETE\_EMP, method = RequestMethod.PUT)

public @ResponseBody Employee deleteEmployee(@PathVariable("id") int empId) {

logger.info("Start deleteEmployee.");

Employee emp = empData.get(empId);

empData.remove(empId);

return emp;

}

}

import com.journaldev.spring.controller.EmpRestURIConstants;

import com.journaldev.spring.model.Employee;

public class TestSpringRestExample {

public static final String SERVER\_URI = "http://localhost:9090/SpringRestExample";

public static void main(String args[]){

testGetDummyEmployee();

System.out.println("\*\*\*\*\*");

testCreateEmployee();

System.out.println("\*\*\*\*\*");

testGetEmployee();

System.out.println("\*\*\*\*\*");

testGetAllEmployee();

}

private static void testGetAllEmployee() {

RestTemplate restTemplate = new RestTemplate();

//we can't get List<Employee> because JSON convertor doesn't know the type of

//object in the list and hence convert it to default JSON object type LinkedHashMap

List<LinkedHashMap> emps = restTemplate.getForObject(SERVER\_URI+EmpRestURIConstants.GET\_ALL\_EMP, List.class);

System.out.println(emps.size());

for(LinkedHashMap map : emps){

System.out.println("ID="+map.get("id")+",Name="+map.get("name")+",CreatedDate="+map.get("createdDate"));;

}

}

private static void testCreateEmployee() {

RestTemplate restTemplate = new RestTemplate();

Employee emp = new Employee();

emp.setId(1);emp.setName("Pankaj Kumar");

Employee response = restTemplate.postForObject(SERVER\_URI+EmpRestURIConstants.CREATE\_EMP, emp, Employee.class);

printEmpData(response);

}

private static void testGetEmployee() {

RestTemplate restTemplate = new RestTemplate();

Employee emp = restTemplate.getForObject(SERVER\_URI+"/rest/emp/1", Employee.class);

printEmpData(emp);

}

private static void testGetDummyEmployee() {

RestTemplate restTemplate = new RestTemplate();

Employee emp = restTemplate.getForObject(SERVER\_URI+EmpRestURIConstants.DUMMY\_EMP, Employee.class);

printEmpData(emp);

}

public static void printEmpData(Employee emp){

System.out.println("ID="+emp.getId()+",Name="+emp.getName()+",CreatedDate="+emp.getCreatedDate());

}

}

# Spring MVC 4 RESTFul Web Services CRUD Example+RestTemplate

**Created on:**August 11, 2015 | **Last updated on:**September 30, 2017  [websystiqueadmin](http://websystique.com/author/blogadminuser/)

In this post we will write a CRUD Restful WebService using Spring MVC 4, and write a REST client with RestTemplate to consume those services. We will also test those services using external clients. Let’s get going.

### Short & Quick introduction to REST

[REST](http://stackoverflow.com/questions/671118/what-exactly-is-restful-programming) stands for Representational State Transfer.It’s an is an architectural style which can be used to design web services, that can be consumed from a variety of clients. The core idea is that, rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls among them.

In Rest based design, resources are being manipulated using a common set of verbs.

* To Create a resource : HTTP POST should be used
* To Retrieve a resource : HTTP GET should be used
* To Update a resource : HTTP PUT should be used
* To Delete a resource : HTTP DELETE should be used

That means, you as a REST service developer or Client, should comply to above criteria, in order to be REST complained.

Often Rest based Web services return JSON or XML as response, although it is not limited to these types only. Clients can specify (using HTTP **Accept header**) the resource type they are interested in, and server may return the resource , specifying **Content-Type** of the resource it is serving. This [StackOverflow link](http://stackoverflow.com/questions/671118/what-exactly-is-restful-programming) is a must read to understand REST in detail.

**Other interesting posts you may like**

* [Spring Boot+AngularJS+Spring Data+Hibernate+MySQL CRUD App](http://websystique.com/spring-boot/spring-boot-angularjs-spring-data-jpa-crud-app-example/)
* [Spring Boot REST API Tutorial](http://websystique.com/spring-boot/spring-boot-rest-api-example/)
* [Spring Boot WAR deployment example](http://websystique.com/spring-boot/spring-boot-war-deployment-example/)
* [Secure Spring REST API using OAuth2](http://websystique.com/spring-security/secure-spring-rest-api-using-oauth2/)
* [Spring Boot Introduction + Hello World Example](http://websystique.com/spring-boot/spring-boot-introduction-hello-world-example/)
* [Spring 4 MVC+JPA2+Hibernate Many-to-many Example](http://websystique.com/springmvc/spring-4-mvc-jpa2-hibernate-many-to-many-example/)
* [AngularJS+Spring Security using Basic Authentication](http://websystique.com/spring-security/angularjs-basic-authentication-using-spring-security/)
* [Secure Spring REST API using Basic Authentication](http://websystique.com/spring-security/secure-spring-rest-api-using-basic-authentication/)
* [Spring 4 Email Template Library Example](http://websystique.com/spring/spring-4-email-using-velocity-freemaker-template-library/)
* [Spring 4 Cache Tutorial with EhCache](http://websystique.com/spring/spring-4-cache-tutorial-with-ehcache/)
* [Spring 4 Email With Attachment Tutorial](http://websystique.com/spring/spring-4-email-with-attachment-tutorial/)
* [Spring 4 Email Integration Tutorial](http://websystique.com/spring/spring-4-email-integration-tutorial/)
* [Spring MVC 4+JMS+ActiveMQ Integration Example](http://websystique.com/springmvc/spring-4-mvc-jms-activemq-annotation-based-example/)
* [Spring 4+JMS+ActiveMQ @JmsLister @EnableJms Example](http://websystique.com/spring/spring-4-jms-activemq-example-with-jmslistener-enablejms/)
* [Spring 4+JMS+ActiveMQ Integration Example](http://websystique.com/spring/spring-4-jms-activemq-example-with-annotations/)
* [Spring MVC 4+Apache Tiles 3 Integration Example](http://websystique.com/springmvc/spring-4-mvc-apache-tiles-3-annotation-based-example/)
* [Spring MVC 4+Spring Security 4 + Hibernate Integration Example](http://websystique.com/springmvc/spring-mvc-4-and-spring-security-4-integration-example/)
* [Spring MVC 4+AngularJS Example](http://websystique.com/springmvc/spring-mvc-4-angularjs-example/)
* [Spring MVC 4+AngularJS Routing with ngRoute Example](http://websystique.com/springmvc/spring-4-mvc-angularjs-routing-example-using-ngroute/)
* [Spring MVC 4+AngularJS Routing with UI-Router Example](http://websystique.com/springmvc/spring-4-mvc-angularjs-routing-example-using-ui-router/)
* [Spring MVC 4+Hibernate 4 Many-to-many JSP Example](http://websystique.com/springmvc/springmvc-hibernate-many-to-many-example-annotation-using-join-table/)
* [Spring MVC 4+Hibernate 4+MySQL+Maven integration example using annotations](http://websystique.com/springmvc/spring-4-mvc-and-hibernate4-integration-example-using-annotations/)
* [Spring MVC4 FileUpload-Download Hibernate+MySQL Example](http://websystique.com/springmvc/spring-mvc-4-fileupload-download-hibernate-example/)
* [TestNG Mockito Integration Example Stubbing Void Methods](http://websystique.com/java/testing/testng-mockito-example-stubbing-void-methods/)
* [Maven surefire plugin and TestNG Example](http://websystique.com/java/testing/maven-surefire-plugin-testng/)
* [Spring MVC 4 Form Validation and Resource Handling](http://websystique.com/springmvc/spring-4-mvc-form-validation-with-hibernate-jsr-validator-resource-handling-using-annotations/)

## ****Rest Based Controller****

Following is one possible Rest based controller, implementing REST API. I said possible, means Other’s may implement it in another way, still (or even more pure way) conforming to REST style.

**This is what our REST API does:**

* **GET** request to /api/user/ returns a list of users
* **GET** request to /api/user/1 returns the user with ID 1
* **POST** request to /api/user/ with a user object as JSON creates a new user
* **PUT** request to /api/user/3 with a user object as JSON updates the user with ID 3
* **DELETE** request to /api/user/4 deletes the user with ID 4
* **DELETE** request to /api/user/ deletes all the users

|  |
| --- |
| package com.websystique.springmvc.controller;    import java.util.List;    import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.http.HttpHeaders;  import org.springframework.http.HttpStatus;  import org.springframework.http.MediaType;  import org.springframework.http.ResponseEntity;  import org.springframework.web.bind.annotation.PathVariable;  import org.springframework.web.bind.annotation.RequestBody;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.RestController;  import org.springframework.web.util.UriComponentsBuilder;    import com.websystique.springmvc.model.User;  import com.websystique.springmvc.service.UserService;    @RestController  public class HelloWorldRestController {        @Autowired      UserService userService;  //Service which will do all data retrieval/manipulation work          //-------------------Retrieve All Users--------------------------------------------------------        @RequestMapping(value = "/user/", method = RequestMethod.GET)      public ResponseEntity<List<User>> listAllUsers() {          List<User> users = userService.findAllUsers();          if(users.isEmpty()){              return new ResponseEntity<List<User>>(HttpStatus.NO\_CONTENT);//You many decide to return HttpStatus.NOT\_FOUND          }          return new ResponseEntity<List<User>>(users, HttpStatus.OK);      }          //-------------------Retrieve Single User--------------------------------------------------------        @RequestMapping(value = "/user/{id}", method = RequestMethod.GET, produces = MediaType.APPLICATION\_JSON\_VALUE)      public ResponseEntity<User> getUser(@PathVariable("id") long id) {          System.out.println("Fetching User with id " + id);          User user = userService.findById(id);          if (user == null) {              System.out.println("User with id " + id + " not found");              return new ResponseEntity<User>(HttpStatus.NOT\_FOUND);          }          return new ResponseEntity<User>(user, HttpStatus.OK);      }            //-------------------Create a User--------------------------------------------------------        @RequestMapping(value = "/user/", method = RequestMethod.POST)      public ResponseEntity<Void> createUser(@RequestBody User user,    UriComponentsBuilder ucBuilder) {          System.out.println("Creating User " + user.getName());            if (userService.isUserExist(user)) {              System.out.println("A User with name " + user.getName() + " already exist");              return new ResponseEntity<Void>(HttpStatus.CONFLICT);          }            userService.saveUser(user);            HttpHeaders headers = new HttpHeaders();          headers.setLocation(ucBuilder.path("/user/{id}").buildAndExpand(user.getId()).toUri());          return new ResponseEntity<Void>(headers, HttpStatus.CREATED);      }          //------------------- Update a User --------------------------------------------------------        @RequestMapping(value = "/user/{id}", method = RequestMethod.PUT)      public ResponseEntity<User> updateUser(@PathVariable("id") long id, @RequestBody User user) {          System.out.println("Updating User " + id);            User currentUser = userService.findById(id);            if (currentUser==null) {              System.out.println("User with id " + id + " not found");              return new ResponseEntity<User>(HttpStatus.NOT\_FOUND);          }            currentUser.setName(user.getName());          currentUser.setAge(user.getAge());          currentUser.setSalary(user.getSalary());            userService.updateUser(currentUser);          return new ResponseEntity<User>(currentUser, HttpStatus.OK);      }        //------------------- Delete a User --------------------------------------------------------        @RequestMapping(value = "/user/{id}", method = RequestMethod.DELETE)      public ResponseEntity<User> deleteUser(@PathVariable("id") long id) {          System.out.println("Fetching & Deleting User with id " + id);            User user = userService.findById(id);          if (user == null) {              System.out.println("Unable to delete. User with id " + id + " not found");              return new ResponseEntity<User>(HttpStatus.NOT\_FOUND);          }            userService.deleteUserById(id);          return new ResponseEntity<User>(HttpStatus.NO\_CONTENT);      }          //------------------- Delete All Users --------------------------------------------------------        @RequestMapping(value = "/user/", method = RequestMethod.DELETE)      public ResponseEntity<User> deleteAllUsers() {          System.out.println("Deleting All Users");            userService.deleteAllUsers();          return new ResponseEntity<User>(HttpStatus.NO\_CONTENT);      }    } |

**Detailed Explanation :**

**@RestController** : First of all, we are using Spring 4′s new @RestController annotation. This annotation eliminates the need of annotating each method with @ResponseBody. Under the hood, @RestController is itself annotated with @ResponseBody, and can be considered as combination of @Controller and @ResponseBody.

**@RequestBody** : If a method parameter is annotated with @RequestBody, Spring will bind the incoming HTTP request body(for the URL mentioned in @RequestMapping for that method) to that parameter. While doing that, Spring will [behind the scenes] use [HTTP Message converters](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/" \l "rest-message-conversion) to convert the HTTP request body into domain object [deserialize request body to domain object], based on ACCEPT or Content-Type header present in request.

**@ResponseBody** : If a method is annotated with @ResponseBody, Spring will bind the return value to outgoing HTTP response body. While doing that, Spring will [behind the scenes] use [HTTP Message converters](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/" \l "rest-message-conversion) to convert the return value to HTTP response body [serialize the object to response body], based on Content-Type present in request HTTP header. As already mentioned, in Spring 4, you may stop using this annotation.

**ResponseEntity** is a real deal. It represents the entire HTTP response. Good thing about it is that you can control anything that goes into it. You can specify status code, headers, and body. It comes with several constructors to carry the information you want to sent in HTTP Response.

**@PathVariable** This annotation indicates that a method parameter should be bound to a URI template variable [the one in '{}'].

Basically, @RestController , @RequestBody, ResponseEntity & @PathVariable are all you need to know to implement a REST API in Spring 4. Additionally, spring provides several support classes to help you implement something customized.

**MediaType :** With @RequestMapping annotation, you can additionally, specify the MediaType to be produced or consumed (using **produces** or **consumes** attributes) by that particular controller method, to further narrow down the mapping.

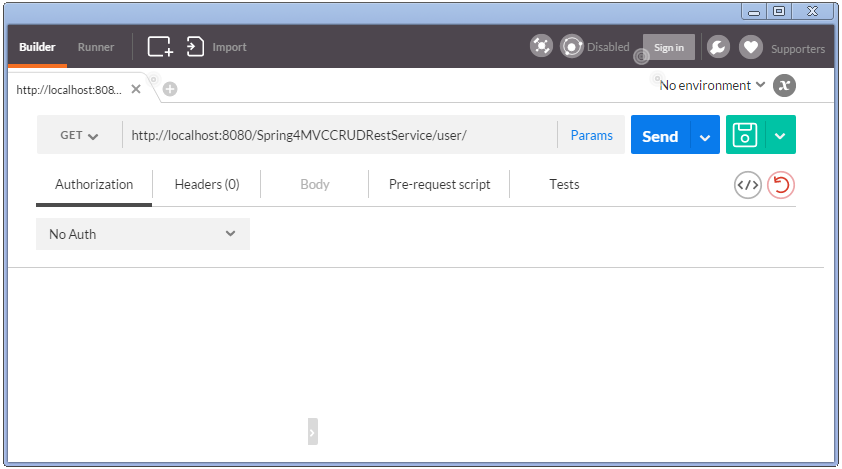
### ****Deploy and Test this API, let’s dig deeper into how this thing works****

At the at end of day, it’s just a plain controller class, part of a deploy-able application.[Complete downloadable application code is shown further down in post which you can deploy straight-away in your container]. I am going to deploy it, in order to see things live and discuss each operation in detail. Deployed Application is available at <http://localhost:8080/Spring4MVCCRUDRestService>.

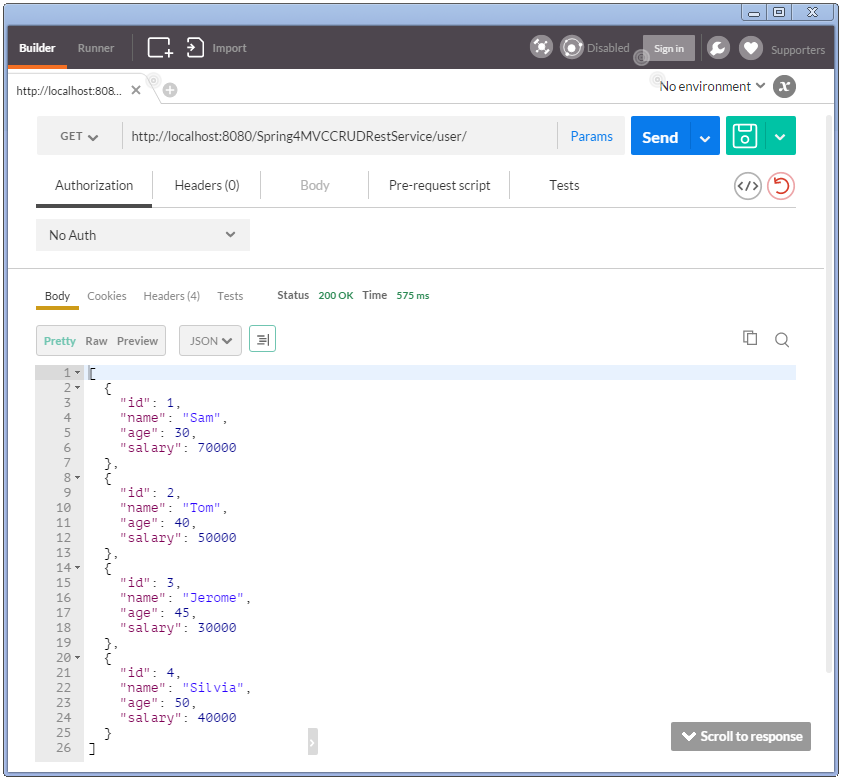
To test this API, i will use an external client POSTMAN (An extension from CHROME). We will write our own client in just few minutes.

**1. Retrieve all users**

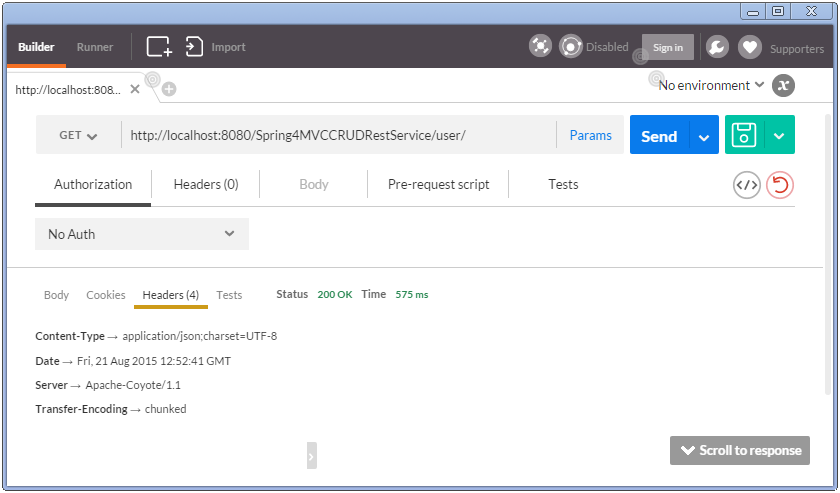
Open POSTMAN tool, select request type [GET for this usecase], specify the operation uri.



Notice that we did not specify any HTTP header here. Click on Send, you will receive list of all users.



Also notice the HTTP 200 response. Additionally check headers.



You might be wondering how the response is sent as JSON string, and the Content-Type header in response confirms that. Glad you asked. This is due to the fact that we have included Jackson library in our project.

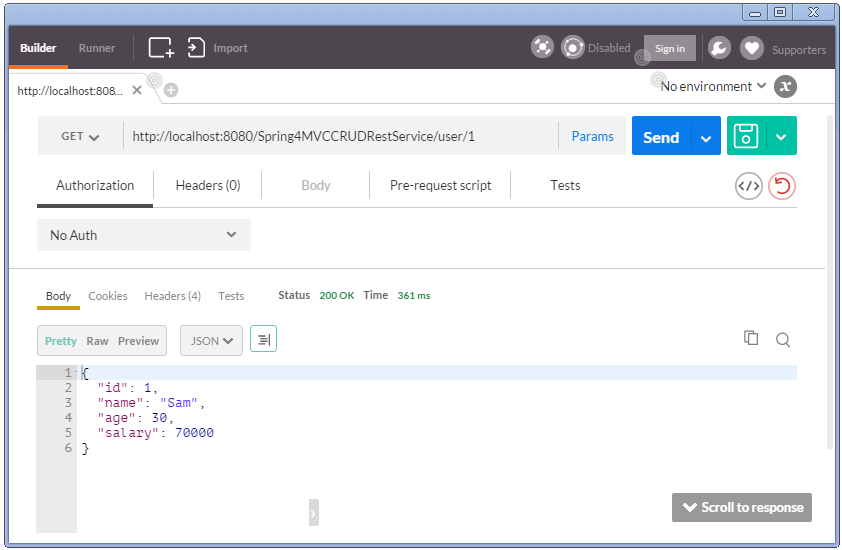
|  |
| --- |
| <dependency>      <groupId>com.fasterxml.jackson.core</groupId>      <artifactId>jackson-databind</artifactId>      <version>2.5.3</version>  </dependency> |

Since spring finds this library in class path, it invokes inbuilt [MappingJackson2HttpMessageConverter](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/" \l "rest-message-conversion) converter to convert the response (List of objects) into JSON.

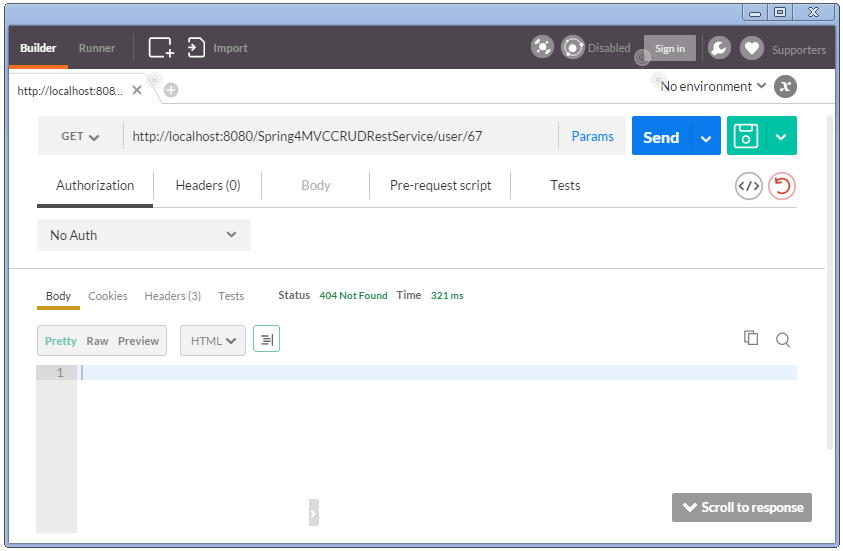
Good thing about Spring inbuilt converters are that most of the time they just need certain library in classpath in order to perform conversion. Of course sometime we do need to adapt our API/application as well. For instance, if we want to serve XML as well, we should annotate User class with proper JAXB annoations.

**2. Retrieve Single User**

Specify a GET with /user/1 , click on send.

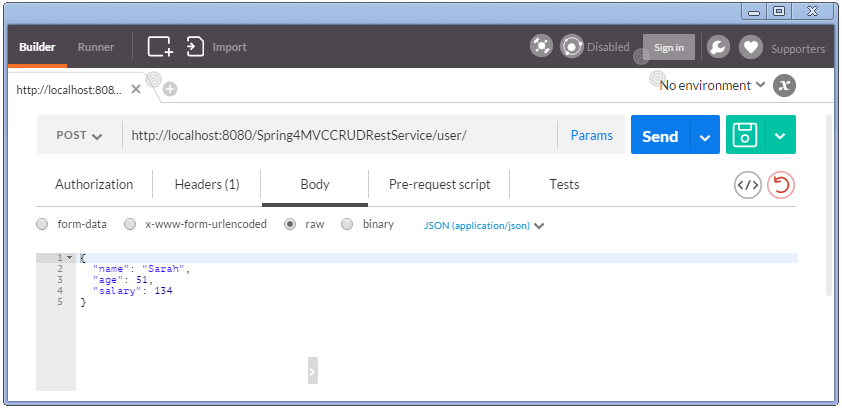


Now try to send a GET with invalid identifier, you should receive a HTTP 404.

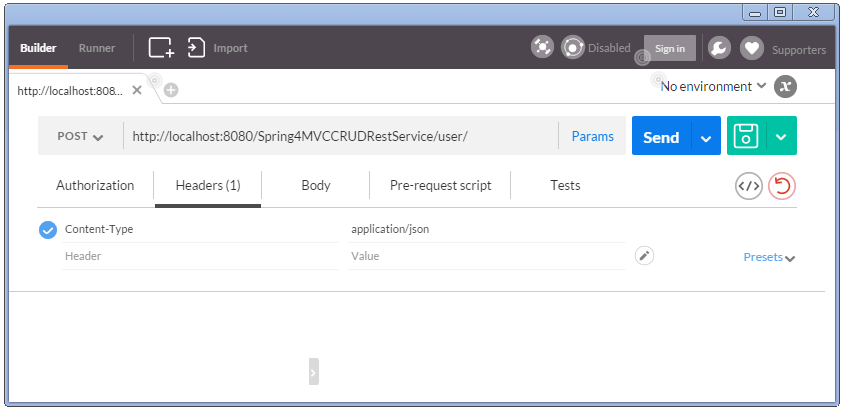


**3. Create a User**

Select the method as POST, specify uri as /user/, specify body in POSTMAN body tab, select the type [application/json].



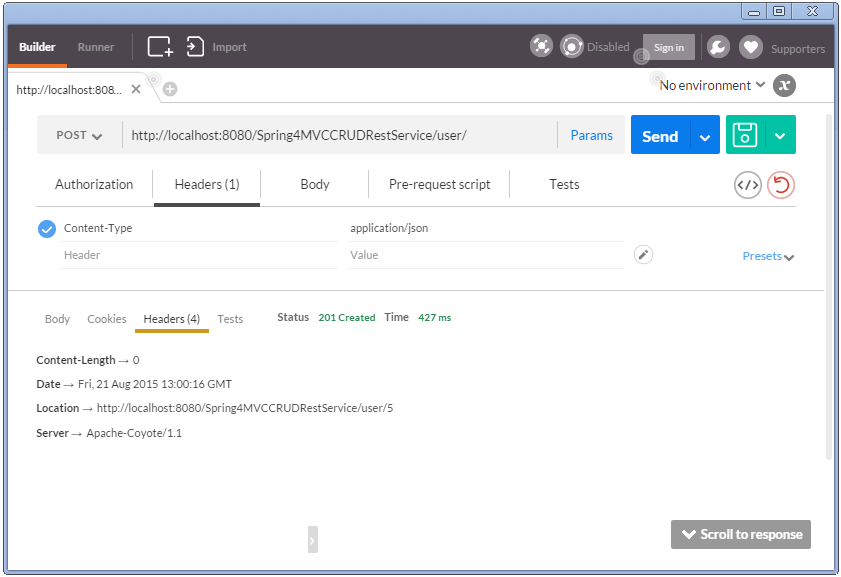
You might have noticed that POSTMAN automatically adds a header **Content-Type**.



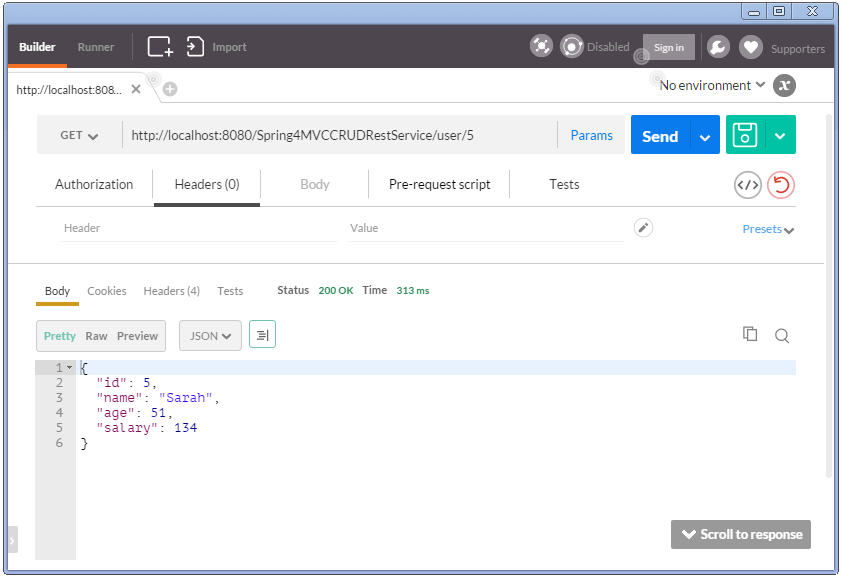
Along with POST and PUT request, clients send the data to the server and they should specify the actual content type of the data being sent.

**Remember :** Accept header says about what type client can understand. Content-Type header says what type of data actually is of.

Send. You should see HTTP 200 response with no body (as API don’t send anything in body). But you should find a **Location header** specifying the location the newly created user can be found at.

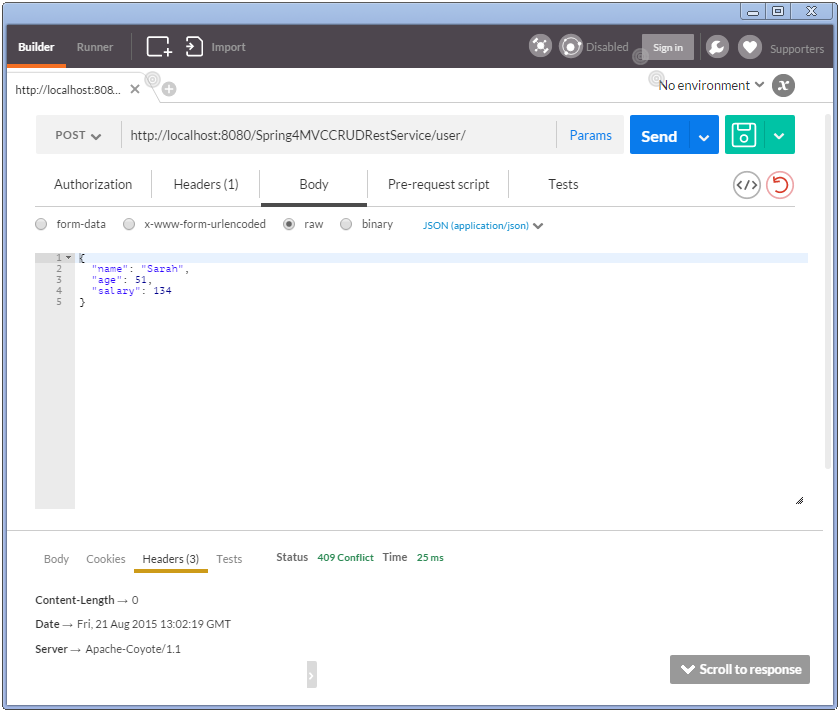


You can now fetch the newly created user.



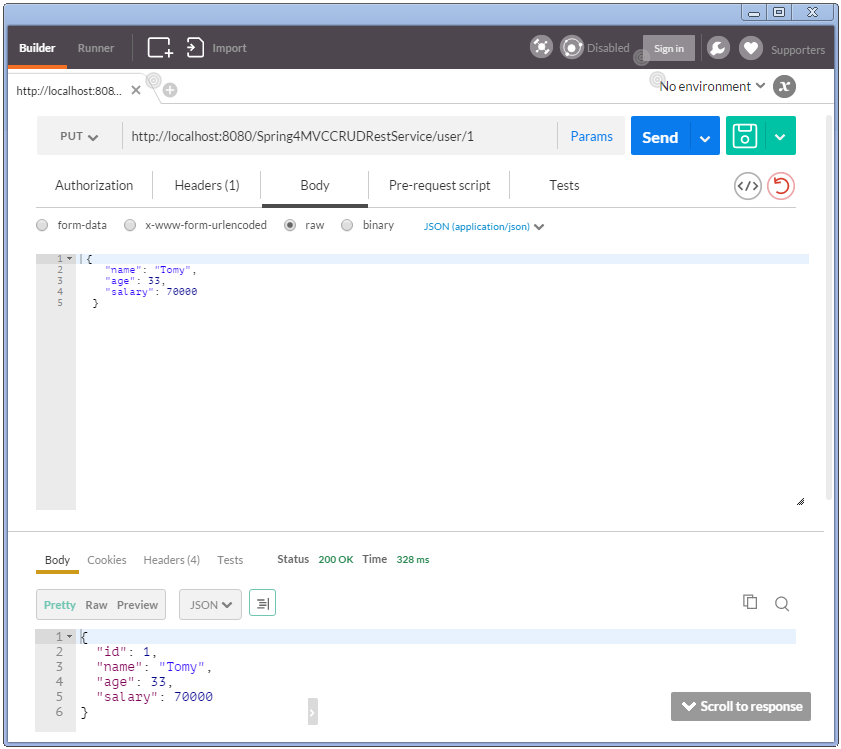
This way of implementation is common in REST. But no one stops you if you do want to send the content in Response body of a POST/PUT request. Will that still be REST complaint API? It’s a debatable point.

Anyway, Lets try to create the same user again.You should get HTTP Conflict response.



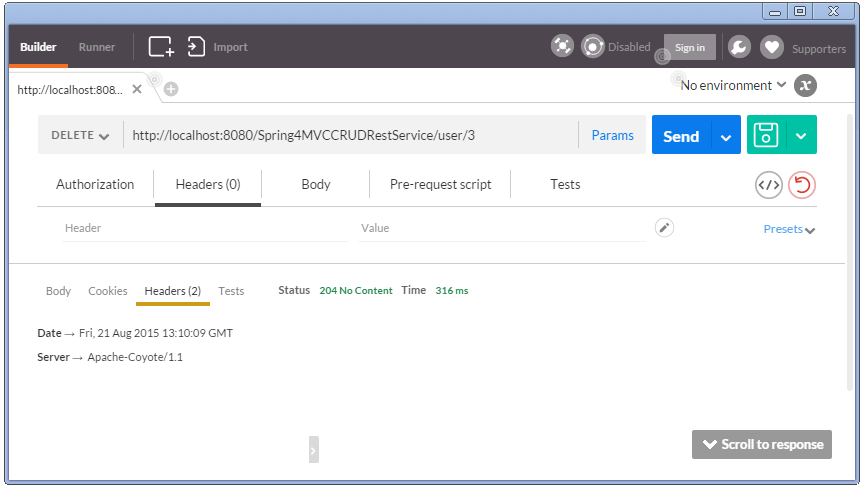
**4. Update a User**

Send a HTTP PUT request to update a user. Send along the new user details to be put in.

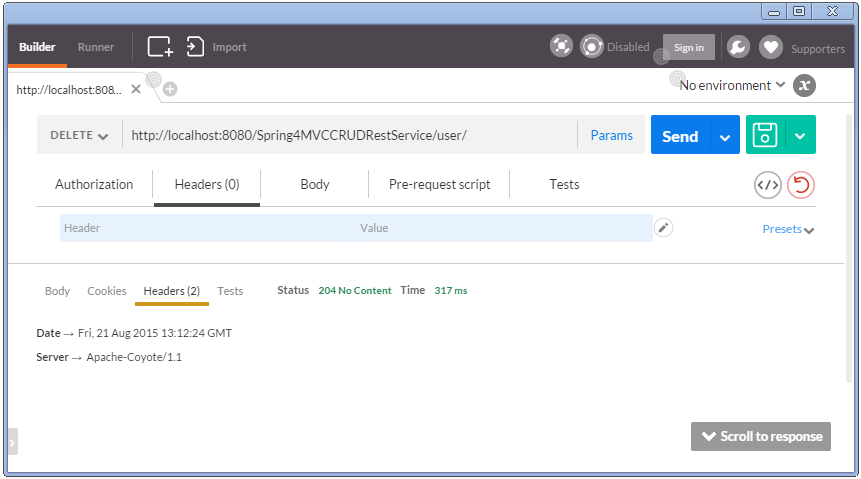


Notice that we have received response body this time. This is because the method implementation in controller is sending it. Again, one may decide not to send the updated details in response body, and just send the location header(as in create).

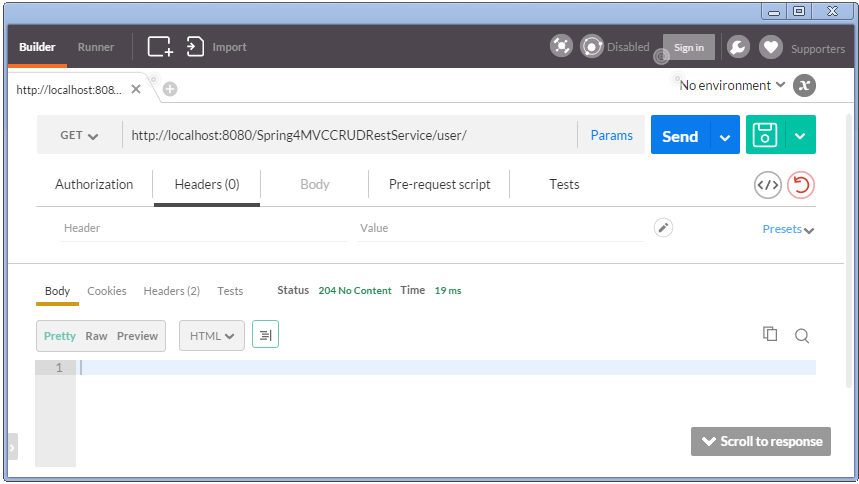
**5. Delete a User**



**6. Delete All Users**



**7. Verify users after delete-all**



## ****Writing REST Client using RestTemplate****

Postman tool we used above is a wonderful Client to test Rest API. But if you want to consume REST based web services from your application, you would need a REST client for your application. One of the most popular HTTP client is Apache HttpComponents **HttpClient**. But the details to access REST services using this are too low level.

Spring’s RestTemplate comes to Rescue. RestTemplate provides higher level methods that correspond to each of the six main HTTP methods that make invoking many RESTful services a one-liner and enforce REST best practices.

Below shown are HTTP methods and corresponding RestTemplate methods to handle that type of HTTP request.

**HTTP Methods and corresponding RestTemplate methods:**

* HTTP GET : getForObject, getForEntity
* HTTP PUT : put(String url, Object request, String…​urlVariables)
* HTTP DELETE : delete
* HTTP POST : postForLocation(String url, Object request, String…​ urlVariables), postForObject(String url, Object request, Class responseType, String…​ uriVariables)
* HTTP HEAD : headForHeaders(String url, String…​ urlVariables)
* HTTP OPTIONS : optionsForAllow(String url, String…​ urlVariables)
* HTTP PATCH and others : exchange execute

**Custom Rest client , consuming the REST services created earlier.**

|  |
| --- |
| package com.websystique.springmvc;    import java.net.URI;  import java.util.LinkedHashMap;  import java.util.List;    import org.springframework.web.client.RestTemplate;    import com.websystique.springmvc.model.User;    public class SpringRestTestClient {        public static final String REST\_SERVICE\_URI = "<a class="vglnk" href="http://localhost:8080/Spring4MVCCRUDRestService" rel="nofollow"><span>http</span><span>://</span><span>localhost</span><span>:</span><span>8080</span><span>/</span><span>Spring4MVCCRUDRestService</span></a>";        /\* GET \*/      @SuppressWarnings("unchecked")      private static void listAllUsers(){          System.out.println("Testing listAllUsers API-----------");            RestTemplate restTemplate = new RestTemplate();          List<LinkedHashMap<String, Object>> usersMap = restTemplate.getForObject(REST\_SERVICE\_URI+"/user/", List.class);            if(usersMap!=null){              for(LinkedHashMap<String, Object> map : usersMap){                  System.out.println("User : id="+map.get("id")+", Name="+map.get("name")+", Age="+map.get("age")+", Salary="+map.get("salary"));;              }          }else{              System.out.println("No user exist----------");          }      }        /\* GET \*/      private static void getUser(){          System.out.println("Testing getUser API----------");          RestTemplate restTemplate = new RestTemplate();          User user = restTemplate.getForObject(REST\_SERVICE\_URI+"/user/1", User.class);          System.out.println(user);      }        /\* POST \*/      private static void createUser() {          System.out.println("Testing create User API----------");          RestTemplate restTemplate = new RestTemplate();          User user = new User(0,"Sarah",51,134);          URI uri = restTemplate.postForLocation(REST\_SERVICE\_URI+"/user/", user, User.class);          System.out.println("Location : "+uri.toASCIIString());      }        /\* PUT \*/      private static void updateUser() {          System.out.println("Testing update User API----------");          RestTemplate restTemplate = new RestTemplate();          User user  = new User(1,"Tomy",33, 70000);          restTemplate.put(REST\_SERVICE\_URI+"/user/1", user);          System.out.println(user);      }        /\* DELETE \*/      private static void deleteUser() {          System.out.println("Testing delete User API----------");          RestTemplate restTemplate = new RestTemplate();          restTemplate.delete(REST\_SERVICE\_URI+"/user/3");      }          /\* DELETE \*/      private static void deleteAllUsers() {          System.out.println("Testing all delete Users API----------");          RestTemplate restTemplate = new RestTemplate();          restTemplate.delete(REST\_SERVICE\_URI+"/user/");      }        public static void main(String args[]){          listAllUsers();          getUser();          createUser();          listAllUsers();          updateUser();          listAllUsers();          deleteUser();          listAllUsers();          deleteAllUsers();          listAllUsers();      }  } |

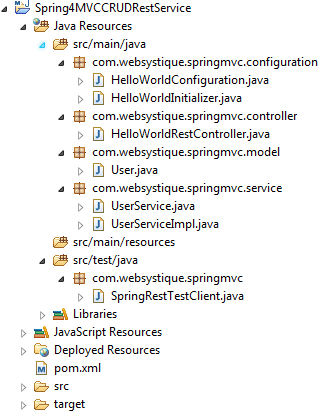
Restart server(In our example, data on server side is fixed.). Run above program.

**Output from above Client program**

|  |
| --- |
| Testing listAllUsers API-----------  User : id=1, Name=Sam, Age=30, Salary=70000.0  User : id=2, Name=Tom, Age=40, Salary=50000.0  User : id=3, Name=Jerome, Age=45, Salary=30000.0  User : id=4, Name=Silvia, Age=50, Salary=40000.0  Testing getUser API----------  User [id=1, name=Sam, age=30, salary=70000.0]  Testing create User API----------  Location : <a class="vglnk" href="<http://localhost:8080/Spring4MVCCRUDRestService/user/5>" rel="nofollow"><span>http</span><span>://</span><span>localhost</span><span>:</span><span>8080</span><span>/</span><span>Spring4MVCCRUDRestService</span><span>/</span><span>user</span><span>/</span><span>5</span></a>  Testing listAllUsers API-----------  User : id=1, Name=Sam, Age=30, Salary=70000.0  User : id=2, Name=Tom, Age=40, Salary=50000.0  User : id=3, Name=Jerome, Age=45, Salary=30000.0  User : id=4, Name=Silvia, Age=50, Salary=40000.0  User : id=5, Name=Sarah, Age=51, Salary=134.0  Testing update User API----------  User [id=1, name=Tomy, age=33, salary=70000.0]  Testing listAllUsers API-----------  User : id=1, Name=Tomy, Age=33, Salary=70000.0  User : id=2, Name=Tom, Age=40, Salary=50000.0  User : id=3, Name=Jerome, Age=45, Salary=30000.0  User : id=4, Name=Silvia, Age=50, Salary=40000.0  User : id=5, Name=Sarah, Age=51, Salary=134.0  Testing delete User API----------  Testing listAllUsers API-----------  User : id=1, Name=Tomy, Age=33, Salary=70000.0  User : id=2, Name=Tom, Age=40, Salary=50000.0  User : id=4, Name=Silvia, Age=50, Salary=40000.0  User : id=5, Name=Sarah, Age=51, Salary=134.0  Testing all delete Users API----------  Testing listAllUsers API-----------  No user exist---------- |

# Complete Example

### ****Project Structure****



### ****Declare project dependencies****

|  |
| --- |
| <project xmlns="<a class="vglnk" href="<http://maven.apache.org/POM/4.0.0>" rel="nofollow"><span>http</span><span>://</span><span>maven</span><span>.</span><span>apache</span><span>.</span><span>org</span><span>/</span><span>POM</span><span>/</span><span>4</span><span>.</span><span>0</span><span>.</span><span>0</span></a>" xmlns:xsi="<a class="vglnk" href="<http://www.w3.org/2001/XMLSchema-instance>" rel="nofollow"><span>http</span><span>://</span><span>www</span><span>.</span><span>w3</span><span>.</span><span>org</span><span>/</span><span>2001</span><span>/</span><span>XMLSchema</span><span>-</span><span>instance</span></a>"    xsi:schemaLocation="<a class="vglnk" href="<http://maven.apache.org/POM/4.0.0>" rel="nofollow"><span>http</span><span>://</span><span>maven</span><span>.</span><span>apache</span><span>.</span><span>org</span><span>/</span><span>POM</span><span>/</span><span>4</span><span>.</span><span>0</span><span>.</span><span>0</span></a> <a class="vglnk" href="<http://maven.apache.org/maven-v4_0_0.xsd>" rel="nofollow"><span>http</span><span>://</span><span>maven</span><span>.</span><span>apache</span><span>.</span><span>org</span><span>/</span><span>maven</span><span>-</span><span>v4</span><span>\_</span><span>0</span><span>\_</span><span>0</span><span>.</span><span>xsd</span></a>">    <modelVersion>4.0.0</modelVersion>    <groupId>com.websystique.springmvc</groupId>    <artifactId>Spring4MVCCRUDRestService</artifactId>    <packaging>war</packaging>    <version>1.0.0</version>    <name>Spring4MVCCRUDRestService Maven Webapp</name>        <properties>          <springframework.version>4.2.0.RELEASE</springframework.version>          <jackson.version>2.5.3</jackson.version>      </properties>        <dependencies>          <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-webmvc</artifactId>              <version>${springframework.version}</version>          </dependency>          <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-tx</artifactId>              <version>${springframework.version}</version>          </dependency>            <dependency>              <groupId>com.fasterxml.jackson.core</groupId>              <artifactId>jackson-databind</artifactId>              <version>${jackson.version}</version>          </dependency>          <dependency>              <groupId>javax.servlet</groupId>              <artifactId>javax.servlet-api</artifactId>              <version>3.1.0</version>          </dependency>        </dependencies>          <build>          <pluginManagement>              <plugins>                  <plugin>                      <groupId>org.apache.maven.plugins</groupId>                      <artifactId>maven-compiler-plugin</artifactId>                      <version>3.2</version>                      <configuration>                          <source>1.7</source>                          <target>1.7</target>                      </configuration>                  </plugin>                  <plugin>                      <groupId>org.apache.maven.plugins</groupId>                      <artifactId>maven-war-plugin</artifactId>                      <version>2.4</version>                      <configuration>                          <warSourceDirectory>src/main/webapp</warSourceDirectory>                          <warName>Spring4MVCCRUDRestService</warName>                          <failOnMissingWebXml>false</failOnMissingWebXml>                      </configuration>                  </plugin>              </plugins>          </pluginManagement>            <finalName>Spring4MVCCRUDRestService</finalName>      </build>  </project> |

### ****User Service****

|  |
| --- |
| package com.websystique.springmvc.service;    import java.util.List;    import com.websystique.springmvc.model.User;        public interface UserService {        User findById(long id);        User findByName(String name);        void saveUser(User user);        void updateUser(User user);        void deleteUserById(long id);        List<User> findAllUsers();        void deleteAllUsers();        public boolean isUserExist(User user);    } |
| package com.websystique.springmvc.service;    import java.util.ArrayList;  import java.util.Iterator;  import java.util.List;  import java.util.concurrent.atomic.AtomicLong;    import org.springframework.stereotype.Service;  import org.springframework.transaction.annotation.Transactional;    import com.websystique.springmvc.model.User;    @Service("userService")  @Transactional  public class UserServiceImpl implements UserService{        private static final AtomicLong counter = new AtomicLong();        private static List<User> users;        static{          users= populateDummyUsers();      }        public List<User> findAllUsers() {          return users;      }        public User findById(long id) {          for(User user : users){              if(user.getId() == id){                  return user;              }          }          return null;      }        public User findByName(String name) {          for(User user : users){              if(user.getName().equalsIgnoreCase(name)){                  return user;              }          }          return null;      }        public void saveUser(User user) {          user.setId(counter.incrementAndGet());          users.add(user);      }        public void updateUser(User user) {          int index = users.indexOf(user);          users.set(index, user);      }        public void deleteUserById(long id) {            for (Iterator<User> iterator = users.iterator(); iterator.hasNext(); ) {              User user = iterator.next();              if (user.getId() == id) {                  iterator.remove();              }          }      }        public boolean isUserExist(User user) {          return findByName(user.getName())!=null;      }        private static List<User> populateDummyUsers(){          List<User> users = new ArrayList<User>();          users.add(new User(counter.incrementAndGet(),"Sam",30, 70000));          users.add(new User(counter.incrementAndGet(),"Tom",40, 50000));          users.add(new User(counter.incrementAndGet(),"Jerome",45, 30000));          users.add(new User(counter.incrementAndGet(),"Silvia",50, 40000));          return users;      }        public void deleteAllUsers() {          users.clear();      }    } |

### ****Model class****

|  |
| --- |
| package com.websystique.springmvc.model;    public class User {        private long id;        private String name;        private int age;        private double salary;        public User(){          id=0;      }        public User(long id, String name, int age, double salary){          this.id = id;          this.name = name;          this.age = age;          this.salary = salary;      }        public long getId() {          return id;      }        public void setId(long id) {          this.id = id;      }        public String getName() {          return name;      }        public void setName(String name) {          this.name = name;      }        public int getAge() {          return age;      }        public void setAge(int age) {          this.age = age;      }        public double getSalary() {          return salary;      }        public void setSalary(double salary) {          this.salary = salary;      }        @Override      public int hashCode() {          final int prime = 31;          int result = 1;          result = prime \* result + (int) (id ^ (id >>> 32));          return result;      }        @Override      public boolean equals(Object obj) {          if (this == obj)              return true;          if (obj == null)              return false;          if (getClass() != obj.getClass())              return false;          User other = (User) obj;          if (id != other.id)              return false;          return true;      }        @Override      public String toString() {          return "User [id=" + id + ", name=" + name + ", age=" + age                  + ", salary=" + salary + "]";      }      } |

### ****Configuration class****

|  |
| --- |
| package com.websystique.springmvc.configuration;    import org.springframework.context.annotation.ComponentScan;  import org.springframework.context.annotation.Configuration;  import org.springframework.web.servlet.config.annotation.EnableWebMvc;    @Configuration  @EnableWebMvc  @ComponentScan(basePackages = "com.websystique.springmvc")  public class HelloWorldConfiguration {      } |

### ****Initialization Class****

|  |
| --- |
| package com.websystique.springmvc.configuration;    import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;    public class HelloWorldInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {        @Override      protected Class<?>[] getRootConfigClasses() {          return new Class[] { HelloWorldConfiguration.class };      }        @Override      protected Class<?>[] getServletConfigClasses() {          return null;      }        @Override      protected String[] getServletMappings() {          return new String[] { "/" };      }    } |

### Adding CORS support to your REST API

While accessing the REST API, you might face issues concerning [Same Origin Policy](https://en.wikipedia.org/wiki/Same-origin_policy).

Errors like :  
” No ‘Access-Control-Allow-Origin’ header is present on the requested resource. Origin ‘[http://127.0.0.1:8080′](http://127.0.0.1:8080%E2%80%B2/) is therefore not allowed access.” OR  
” XMLHttpRequest cannot load <http://abc.com/bla>. Origin [http://localhost:12345](http://localhost:12345/) is not allowed by Access-Control-Allow-Origin.” are common in such case.

Solution is [Cross-Origin Resource Sharing](http://www.html5rocks.com/en/tutorials/cors/). Basically, on server side, we can return additional CORS access control headers with response, which will eventually allow further inter-domain communication.

With Spring, we can write a simple filter which adds those CORS specific headers in each response.

|  |
| --- |
| package com.websystique.springmvc.configuration;    import java.io.IOException;    import javax.servlet.Filter;  import javax.servlet.FilterChain;  import javax.servlet.FilterConfig;  import javax.servlet.ServletException;  import javax.servlet.ServletRequest;  import javax.servlet.ServletResponse;  import javax.servlet.http.HttpServletResponse;      public class CORSFilter implements Filter {        public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws IOException, ServletException {          System.out.println("Filtering on...........................................................");          HttpServletResponse response = (HttpServletResponse) res;          response.setHeader("Access-Control-Allow-Origin", "\*");          response.setHeader("Access-Control-Allow-Methods", "POST, GET, PUT, OPTIONS, DELETE");          response.setHeader("Access-Control-Max-Age", "3600");          response.setHeader("Access-Control-Allow-Headers", "x-requested-with");          chain.doFilter(req, res);      }        public void init(FilterConfig filterConfig) {}        public void destroy() {}    } |

Then we can simply configure it in our Spring configuration like shown below:

|  |
| --- |
| package com.websystique.springmvc.configuration;    import javax.servlet.Filter;    import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;    public class HelloWorldInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {        @Override      protected Class<?>[] getRootConfigClasses() {          return new Class[] { HelloWorldConfiguration.class };      }        @Override      protected Class<?>[] getServletConfigClasses() {          return null;      }        @Override      protected String[] getServletMappings() {          return new String[] { "/" };      }        @Override      protected Filter[] getServletFilters() {          Filter [] singleton = { new CORSFilter()};          return singleton;      }    } |

That’s it. With these two additional steps, clients will be able to communicate with your REST API without worrying about Cross domain issues.

#### Download Source Code

**[Download Now!](http://websystique.com/?smd_process_download=1&download_id=1689" \o "SpringMVC4CRUDRestService)**

With CORS support:

**[Download Now!](http://websystique.com/?smd_process_download=1&download_id=1890" \o "Spring4MVCCRUDRestServiceWithCORS)**

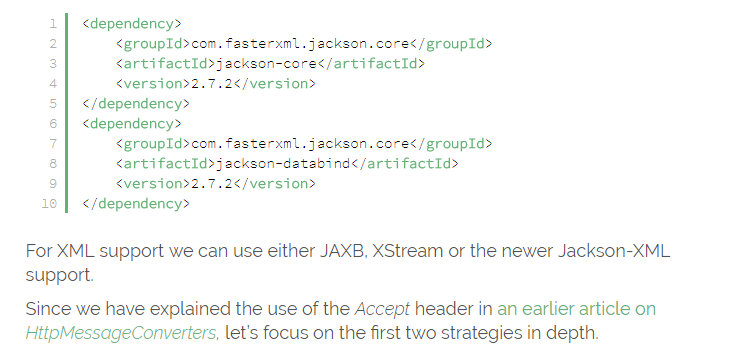
# Content negotiation in a Spring MVC

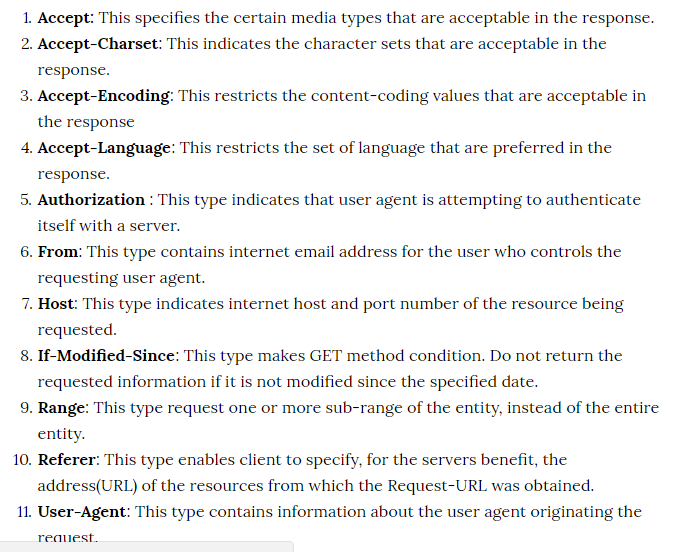
there are three options to determine the media type of a request:

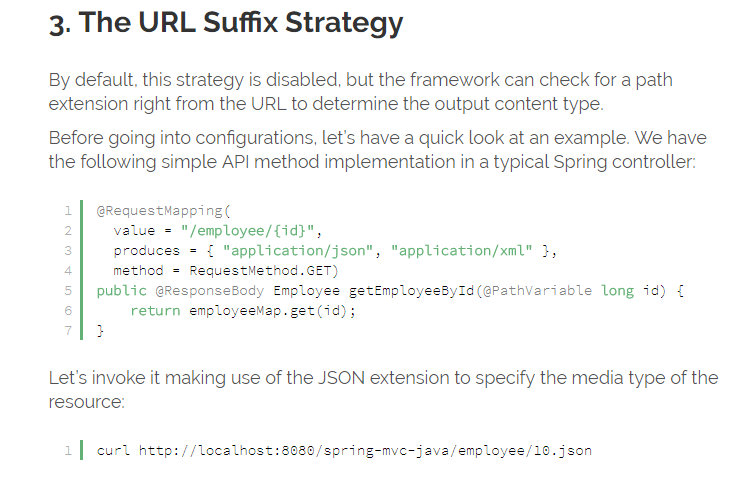
* Using URL suffixes (extensions) in the request (eg *.xml/.json*)
* Using URL parameter in the request (eg *?format=json*)
* Using *Accept* header in the request

By default, this is the order in which the Spring content negotiation manager will try to use these three strategies. And if none of these are enabled, we can specify a fallback to a default content type.

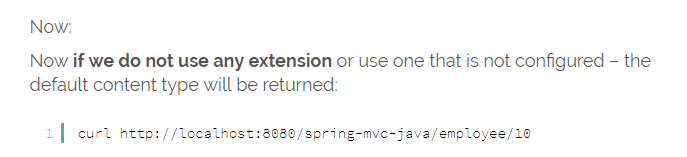
SON and XML representations, so for this article we’ll use Jackson for JSON:

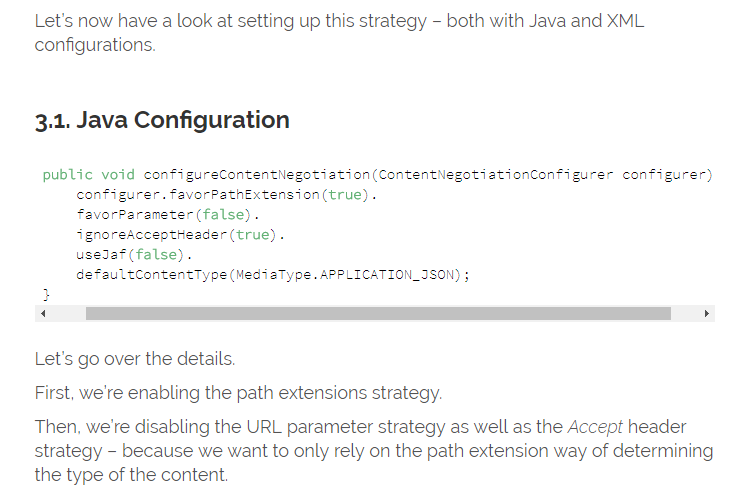












We’re then turning off the Java Activation Framework; JAF can be used as a fallback mechanism to select the output format if the incoming request is not matching any of the strategies we configured. We’re disabling it because we’re going to configure JSON as the default content type.

And finally – we are setting up JSON to be the default. That means if none of the two strategies are matched, all incoming request will be mapped to a controller method that serves JSON.

