<https://madhawacperera.medium.com/simple-java-restful-web-service-with-jersey-jax-rs-implementation-615739857961>

|  |  |
| --- | --- |
| Simple Java RESTful Web Service with Jersey — JAX-RS implementation [Madhawa Perera](https://madhawacperera.medium.com/?source=post_page-----615739857961--------------------------------)  ·  Follow  10 min read  ·  May 8, 2019  74  In this article, I will explain how to implement a simple RESTful web service with Jersey and Apache Tomcat. I assume that you have an understanding of what is a REST API.  I have listed the tools that I have used in this implementation. It is not mandatory to use these tools but this is for your reference if you need to follow the exact steps along with me. Otherwise, always feel free to change it as you wish. If you do not want to spend time going through the whole article but want to analyze the code, you can find it [here](https://github.com/madhawap/jaxrs-sample-webapp) in GitHub.  Let's get started! Prerequisites  * IntelliJ IDEA: A Java IDE by JetBrains. * Apache Maven 3: A build tool (Do not worry if you haven't worked with this before). * Apache Tomcat Web Server version 9: Open source servlet container. * JDK version 1.8.   First things first. Before you start building your first JAVA REST API, there are certain things that you need to keep in your mind. This is to help you to get the background knowledge if you are new to work with Java web applications. You will hear/read lots of these terminologies when you start working with Java EE. What is JAX-RS? JAX-RS is a specification. Further explaining it a specification for implementing REST web services in Java, currently defined by the [JSR 311](https://jcp.org/en/jsr/detail?id=311), [JSR 339](https://jcp.org/en/jsr/detail?id=339) and [JSR-370](https://jcp.org/en/jsr/detail?id=370) which standardize development and deployment of RESTful web services. JSR stands for Java Specification Requests (JSRs) which are the actual descriptions of proposed and final specifications for the Java platform. Need not to worry about those, but if you are further interested to search them on the internet, go ahead. What are Jersey, Restlet, RESTEasy, CXF etc? Well, those are the implementation of the above JAX-RS specification. [Jersey](https://jersey.github.io/) is one of the most popular implementations of JAX-RS used for developing RESTful web services in the Java ecosystem. It is also important to notice that these implementations are shipped with JavaEE implementations (or I will call them as JavaEE Application Servers) such as GlassFish, JBoss, TomEE etc. Therefore you should know that Jersey implementation is shipped with GlassFish JavaEE application server. Knowing this will help you when building RESTful web services with Java. However, there are a few other implementations such as [RESTEasy](https://resteasy.github.io/) (shipped with JBoss EAP and WildFly) and [Apache CXF](http://cxf.apache.org/) (shipped with TomEE and WebSphere) etc. What are GlassFish, JBoss, WildFly, TomEE, Apache Tomcat, Payara, Jetty etc.? Technically explaining those are the implementation of Java EE[1]. Most commonly you can refer to them as Java EE Application Servers. Java EE is again a specification (some call it an abstract API) currently defined by the JSR 316 which everyone is free to implement.  But it is important to notice that all these aren't fully-fledged JavaEE application servers. Oracle [Glassfish 5 full platform](https://javaee.github.io/glassfish/download) and [JBoss EAP](https://developers.redhat.com/products/eap/overview/?referrer=jbd) are basically full Java EE Application Server whereas implementations like [Apache Tomcat 7](http://tomcat.apache.org/), [Eclipse Jetty 8](https://www.eclipse.org/jetty/) are only a Servlet containers which implement only the servlets and JSP specification.  You can search for the other names such as Wildfly, TomEE, Payara and see what they are capable of. I'm not gonna talk about those here as that will increase the length of this article. What is a servlet? Request handling is one of the fundamental things you need to understand when building a Java web application. "In order to respond to requests from the network, a Java web application must first determine what code will respond to the request URL, then marshal a response"[2]. In Java, servlets are used to this request-response handling. Think of a servlet is simply a Java class which is capable of responding to a network request. Like an HTTP request. Servlets run in a servlet container such as Tomcat, Jetty which handles the networking side complexities.  I'd recommend you to go through this article "[What are Java servlets? Request handling for Java web applications](https://www.javaworld.com/article/3313114/what-is-a-java-servlet-request-handling-for-java-web-applications.html)" which will be helpful for you to understand about servlets better. What is JSP? Java Server Pages (JSP) is a Java standard technology that enables you to write dynamic, data-driven pages for your Java web application. "To deploy and run JavaServer Pages, a compatible web server with a servlet container, such as Apache Tomcat or Jetty, is required"[3].  Again for further readings go through this article "[What is JSP? Introduction to JavaServer Pages](https://www.javaworld.com/article/3336161/what-is-jsp-introduction-to-javaserver-pages.html)" which covers all the core knowledge you need to possess with about JSP. Then what is the Spring framework? The Spring Framework is a framework that allows you to create Java enterprise applications[4][5]. It is important to notice that Spring is not a JAX-RS implementation as we discussed above. Spring can be seen as an alternative to the JAX-RS standard. You can build a RESTful web service using JAX-RS implementation without using the Spring framework. Once you finish reading this article (or if you have prior knowledge in working with JAX-RS) come back to this section and go through the following diagram. That will help you understand that the Spring framework is not an implementation of JAX-RS specification but an alternative.  Taken from stormpath.com  If you are still confused with JAX-RS and Spring framework, do not worry. I'd recommend you to read this article on "[JAX-RS vs Spring for REST Endpoints](https://stormpath.com/blog/jax-rs-vs-spring-rest-endpoints)" which will help you more. Let's get started Well, the above description is to provide you with a quick background assuming that you are new to Java web application development. In this article, my main focus is to let you start building your first RESTful web service. Therefore, I'm not going to use the Spring framework in my example. Why? Because using Jersey (JAX-RS implementation) is much easier and can be deployed on any Java application server such as GlassFish, Tomcat, WebSphere, JBoss etc without any problem.  Alright! open IntelliJ IDEA and create a new project. The following screen captures (from Figure 1–7) will show you how to set up your project.  Figure 1  In here, I'm using maven archetype. It means that 'maven' has a pre-built project structure/template to arrange your directories and project files when you are creating a web application. Rather than creating a Java Enterprise project and then add and delete files accordingly, it is easier and faster to use a given template/archetype (Less hazle). Therefore, select org.apache.maven.archetypes:maven-archetype-webapp and then click next.  Figure 2  In here, give GroupId and an ArtifactId. GroupID is to uniquely identify your project across all projects. Usually, we use a reverse domain name style. ArtifactId is used to name the jar file. For further reference read "[Guide to naming conventions on groupId, artifactId, and version](https://maven.apache.org/guides/mini/guide-naming-conventions.html)". Since this project is for learning purposes feel free to name it as you want. Refer to Figure 3 below.  Figure 3  You can see a summary of the details you entered along with the maven version that you will be using in your project. Refer to Figure 4 below.  Figure 4  Now give your project a name and a location to save and click finish. Refer to Figure 5 below.  Figure 5  When your project environment is creating you will see a popup window on the bottom right corner like in Figure 6 below. Click on 'Enable Auto-Import'.  Figure 6  If everything goes smoothly your IDE should look like Figure 7 below.  Figure 7  Now you are all set to write some codes. Before jumping to code there are some configurations you need to do.  Click on the 'main' directory in your project structure and create a new directory named 'java'. Refer to Figure 8 below.  Figure 8  Then click on the created 'java' directory and mark it as sources root. Refer to Figure 9 below.  Figure 9  After you marked it as sources root, create a package name org.madhawa.service. Naming choices are all yours. :D And then under that package create a Java class named HelloService. Refer to Figure 10 and 11.  Figure 10  Figure 11  Now the project structure should look like below. Refer to Figure 12 below.  Figure 12  Now that you have set up the environment. No more directory or file creations. :D  Let's start developing your first REST service, which intended to says “Hi {yourClientName}” to the client.  First, you need to know how to direct all the REST requests that are coming from your REST client to direct towards your HelloSerive class that you created above. We do that by defining a servlet dispatcher in the web application's web.xml file. You can find a web.xml file inside the webapp >> WEB-INF directory. In technical terms, we say that you need to direct all the REST requests to the Jersey container by defining a servlet dispatcher in the application’s web.xml file.  In API terminology we call the HelloService class as a resource. Thus, apart from declaring a servlet, we further define an initialization parameter (<int-param>) indicating the Java package that contains the resources. In this case our resource is inorg.madhawa.services package.  Also, notice that we are adding /rest/\* to the URL pattern inside the web.xml. Thus, when you are calling any resources in your web application after the hostname and the context (which is usually the web applications's name) you need to add the word "/rest/".  Let's create the web.xml. Replace the whole content in your web.xml with the following lines of codes. Refer Sample\_Code\_1:web.xml below.  Sample\_Code\_1: web.xml  Alright, now you can start implementing your resource (HelloService). Before that, you need to import certain packages/dependencies to your project. You do this by adding the dependencies to you pom.xml. Then the build tool (which is maven 3 here) will automatically import those dependencies for you to use in your Java code.  You need to import Jersey core client, Jersey Servlet and Javax WS RS API dependencies. Your pom.xml should be similar to one shown below. Refer to Sample\_Code\_2: pom.xml.  Sample\_Code\_2:pom.xml  Now open the HelloService.java file and you need to enter the following code. Refer the Sample\_Code\_3: HelloService.java.  Sample\_Code\_3: HelloService.java  Let's go through the code quickly.  In here as I mentioned earlier HelloService is your "Resource" or "Resource Class". You can consider this as a typical plain old java object (POJO). Next important aspect is the JAX-RS annotations that are used in the code. These annotations are defined in javax.ws.rs dependency that we imported via maven. These annotations are a part of the JAX-RS (JSR 311) specification.   * Annotation @Path defines the resource base URI. Formed with context root and hostname. In this article, the resource identifier will be something like http://localhost:8080/simple\_webapp/rest/hello/{clientName} * The annotation @GET indicates that greetClient method should responds to the HTTP GET method. * Finally @Produces annotation defines the response type. That means what type of response this method will generate. In here it's "plain/text".   Finally, don't forget to comment or delete index.jsp file(Better delete it).  Figure 13  Well, you are all done with implementing your first RESTfull Java web application. Now it's time to run and test it. Testing your web application. You can either set up Tomcat server inside your IDE or else you can run it separately. Setting up the server and control it inside the IDE is the easiest. Refer to "[How to run/debug your web application with IntelliJ and Tomcat](https://medium.com/@madhawacperera/how-to-run-debug-your-web-application-with-intellij-idea-and-tomcat-ea30c3d15ba0)" to setup Tomcat inside the IntelliJ IDEA. If you get any question here please leave a comment, I will add how to do this here as well. I'm not adding it here, otherwise, the article will get longer.  If you configured correctly then you should see Tomcat as in Figure 14 below.  Figure 14  Then, go to the 'Run' menu and hit on Run…  Figure 15  Now the Tomcat server will be started and your web application will be deployed. Now it is time to test your REST API. You can test your API using many methods. Easiest is to use a web browser. Open your favourite browser and type the endpoint that we designed which is [http://localhost:8080/simple\_webapp/rest/hello/{clientName}](http://localhost:8080/simple_webapp/rest/hello/%7BclientName%7D). In here replace {clientName} with any String that you want. I typed 'madhawa' as you can see in Figure 16. Note that you can use any REST client like Postman, CURL or any other HTTP client who can call this endpoint. The output should be 'Hello {clientName}'. Cheers!  Figure 16  Congratulations!! You wrote your first RESTful web application with JAX-RS/Jersey. Now it's time to do some experiments on your own.  If you like this article and the content that I create, you can buy me a virtual coffee ️😊;  Cheers! References [1]. <https://javaee.github.io/glassfish/>  [2]. <https://www.javaworld.com/article/3313114/what-is-a-java-servlet-request-handling-for-java-web-applications.html>  [3] . <https://en.wikipedia.org/wiki/JavaServer_Pages>  [4]. <https://spring.io/projects/spring-framework>  [5]. <https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html> |  |