JAX-RS: POST, PUT, DELETE & Apache Tomcat

In this practical you will learn how to use the rest of the CRUD operations: POST (i.e., create), PUT (i.e., update) and DELETE and how to deploy your REST Service on Apache Tomcat server.

Table of Contents

[1 POST, PUT and DELETE operations 1](#_Toc785451)

[1.1 DELETE operation 1](#_Toc785452)

[1.2 POST (create) operation 2](#_Toc785453)

[1.3 PUT (update) operation 3](#_Toc785454)

[1.4 Using form parameters in POST and PUT operations 4](#_Toc785455)

[2 Deploy your Service on Apache Tomcat 5](#_Toc785456)

[2.1 Download and Install Apache Tomcat 5](#_Toc785457)

[2.2 Deploy your service “manually” on Tomcat 6](#_Toc785458)

[2.3 Develop Tomcat service in Intellij 9](#_Toc785459)

[2.3.1 Create a new WebServices module in IntelliJ 9](#_Toc785460)

[2.3.2 Create and build an .war artifact in IntelliJ 11](#_Toc785461)

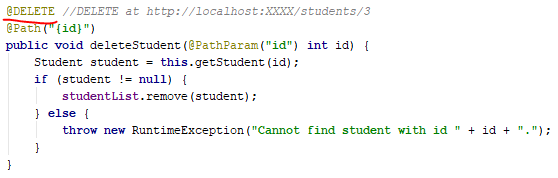
[2.3.3 Create Tomcat Run Configuration in IntelliJ 11](#_Toc785462)

# POST, PUT and DELETE operations

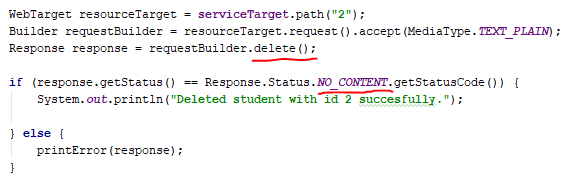
In this section you will learn how to use POST (create), PUT (update) and DELETE operations in the service and client. Note that web browsers only submit GET operations, so you will not be able to use a web browser to test POST, PUT and DELETE. You will add these POST, PUT and DELETE operations to your service and client modules from week 1 (i.e., continue working in the IntelliJ project from week 1).

## DELETE operation

Add one DELETE operation to StudentsResources class in your Service module. This method will be accessible via DELETE operation at [http://localhost:9090/students](http://localhost:9090/students/2)/2 (you cannot test this in the web browser!).



In the Client you can call this DELETE operation as shown below. Note that, if DELETE succeeds, response status will be 204 (i.e., NO\_CONTENT). This is because the http response message has an empty entity, so in your client you do not “read the entity”.



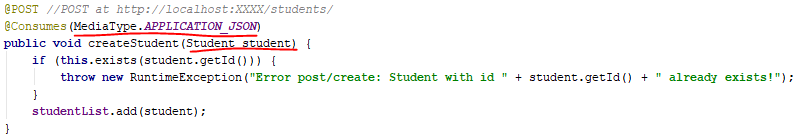
You can use the Burp tool to see how the http request and response messages look like in this case. You will see that:

1. The entity of the http request message is empty.
2. The code on the http response message is 204 NO\_CONTENT and the entity of this message is empty.

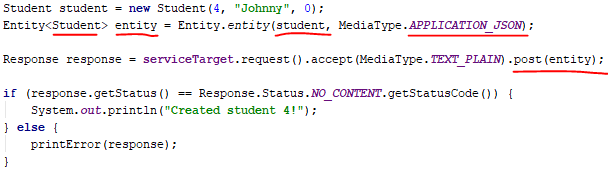
Note that, when you test this method from your Client application by deleting a student and then “getting” the list of all students, the deleted student will still be returned in the list of all students. This is because REST service is stateless (i.e., a new instance of StudentsResources class is recreated every time a client calls the service). This is how your service should be in the “real world”, because your service should work with some external data storage (e.g., file, database, etc.), instead of the hard-coded list of Students created in the constructor. If you want to continue working with your hard-coded list of students during the SOT module in the proper way, you should make the service state-full by adding @Singleton annotation before the StudentsResources class. In real-world projects you should avoid using this annotation.

## POST (create) operation

Add the following POST operation to StudentsResources class in your Service module. Note that the method has a Student object as parameter which will be passed by the client application as json in the entity of the http request message. This method will be accessible via POST operation at [http://localhost:9090/students](http://localhost:9090/students/2) (you cannot test this in the web browser!).



Below you can see how to call this POST operation from the client. Note that you have to prepare the entity of the request message as student object in json format. Also, you only need to check the code of the http response is success (which is 204/no\_content for POST). You do not read the entity of the response, because it will always be empty for POST.

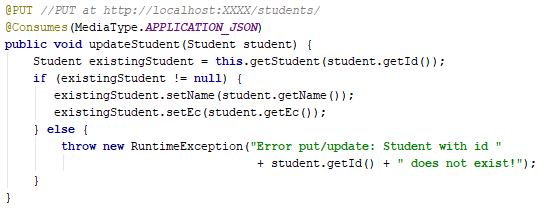


You can use the Burp tool to see how the http request and response messages look like in this case. You will see that:

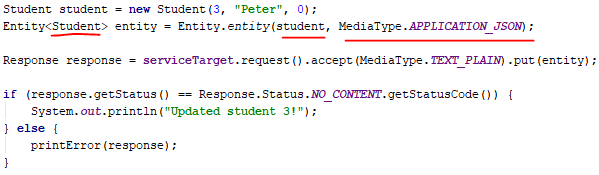
1. The entity of the http request message is json representation of the Student object.
2. The code on the http response message is 204 NO\_CONTENT, and that the entity of this message is empty.

## PUT (update) operation

Add one PUT operation to StudentsResources class in your Service module:



Below you can see how to call this PUT operation from the client. It is very similar like calling a POST operation:



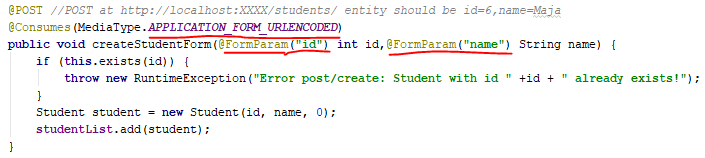
You can use the Burp tool to see how the http request and response messages look like in this case. You see that:

1. The entity of the http request message is json representation of the Student object.
2. The code on the http response message is 204 NO\_CONTENT, and that the entity of this message is empty.

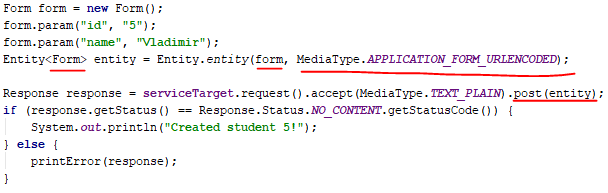
## Using form parameters in POST and PUT operations

In sections 1.2 and 1.3 you learned how to pass json serialisation of Student object as entity of the http request message in POST and PUT operations. In the Service module, both methods for POST and PUT operations have one Student parameter.

Another way to pass parameters in the entity of POST and PUT http request messages is as form parameters. For example, below you can see how you can implement the samePOST operation with form parameters in the entity (instead of Student json)[[1]](#footnote-2). Add following POST operation to your StudentsResources class:



You can call this operation as shown below from your Client:



You can use the Burp tool to see how the http request and response messages look like in this case. You see that:

1. The entity of the http request message are two form parameters.
2. The code on the http response message is 204 NO\_CONTENT, and that the entity of this message is empty.

Note that, technically, you can use all types of parameters: query, path, form, json in all four operations (GET, DELETE, POST and PUT). However, typically you should use form parameters and json in POST and PUT http request messages because then the client passes objects/data to service which should be added or updated as a resource.

# Deploy your Service on Apache Tomcat

## Download and Install Apache Tomcat

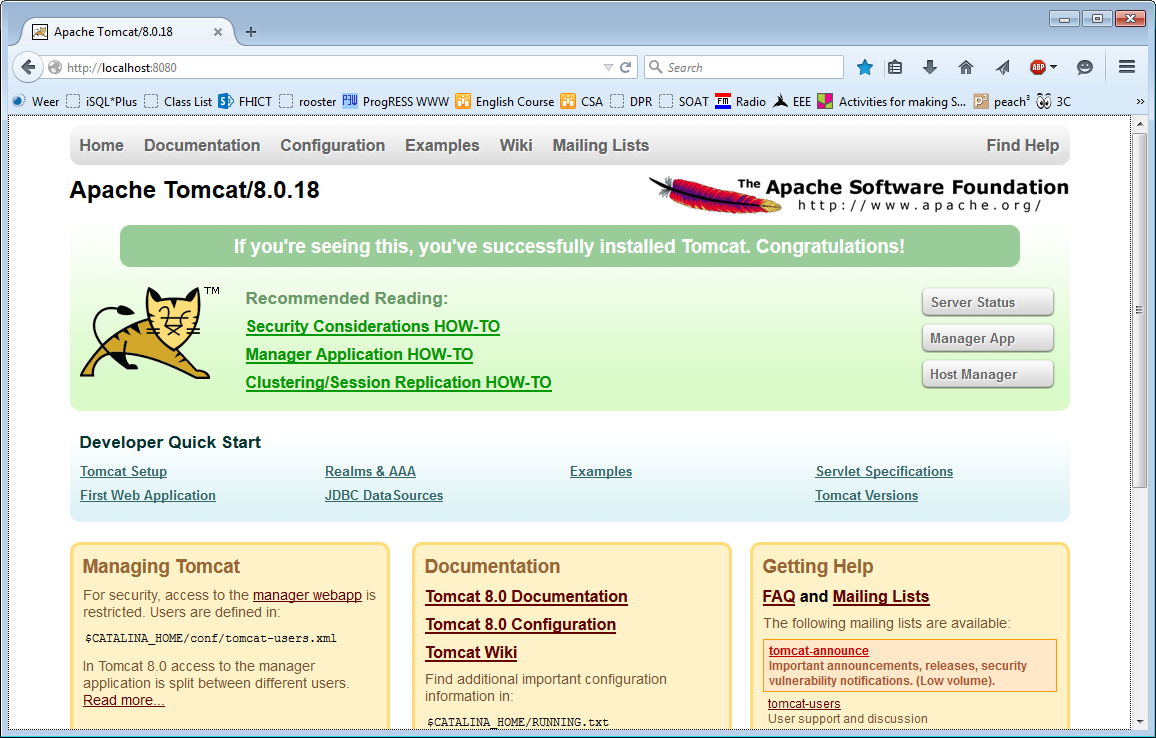
**Download** the latest version of Apache Tomcat from <http://tomcat.apache.org/>: download -> binary distributions -> core -> zip. Unzip it somewhere ( e.g., unzip it to *d:\temp\apache-tomcat-9.0\*).

**Install** Tomcat (read file RUNNING.txt from Tomcat installation folder):

* + Create/set environment variables (My Computer -> Advanced Properties -> Environment Variables -> System)
    - * + Set CATALINA\_HOME to the location of the root directory of the "binary" distribution of Tomcat.
        + Make sure that JAVA\_HOME is set to the location of your JDK.

|  |  |
| --- | --- |
|  |  |

**Start** Tomcat: Start the server with D:\temp\ apache-tomcat-9.0.11\bin\startup.bat. If this goes well, you should be able to access the server at [*http://localhost:8080*](http://localhost:8080)



Later you will be able to see your running service in **Manager App**. However, you will need to give yourself manager rights in Tomcat by following instructions given when you click on the Manager App button.

**Stop** Tomcat: Stop the server with D:\temp\ apache-tomcat-9.0.11\bin\shutdown.bat.

## Deploy your service “manually” on Tomcat

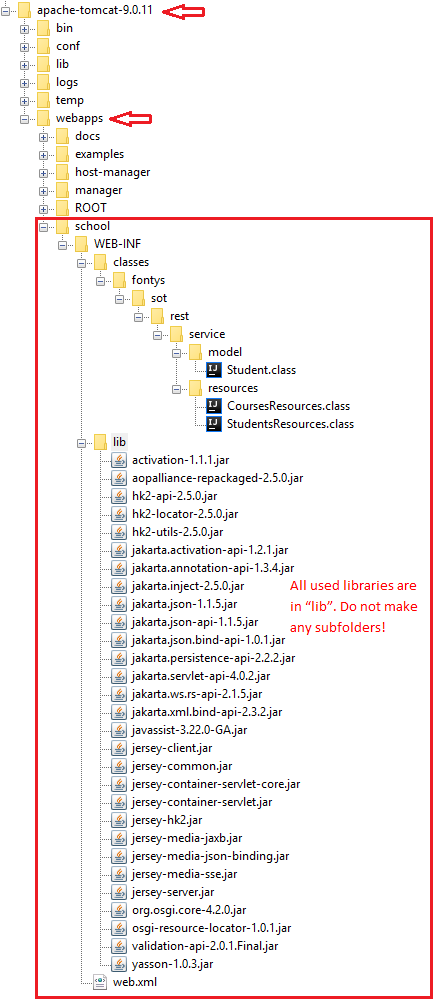
In this section you will deploy your service code **manually** on Tomcat. You will (more or less) copy-paste your service executables to Tomcat. This process will help you understand how a service is hosted on Tomcat. Also, when you finish programming your service and want to host it on Tomcat permanently, you should use the procedure explained in this section. In the next section you will learn how to develop a service for Tomcat in IntelliJ, so that you can develop, debug and test it in IntelliJ.

Because you can host many services on Tomcat (at <http://localhost:8080>), each service should have its own name. For example, our service will be named “school” and available at<http://localhost:8080/school>. From example, you could also deploy another service called “webshop” and that one will be available at <http://localhost:8080/webshop>. So, it is important that each service has a unique name. Our service name will be “school”, and we will make two endpoints: (a) one for working with students resources, and (b) one for working with courses resources. In order to create these two end points change your service as following:

1. Leave class StudentsResources and as it is. If the name of the service is “school”, then this endpoint will be available at <http://localhost:8080/school/students>.
2. Create a CoursesResources class with @Path annotation @Path(“courses”) in the same package like StudentsResources. Add one simple GET method to this class just to test how it works (e.g., return “Here you can work with courses!”). If the name of the service is “school”, then this endpoint will be available at  [http://localhost:8080/school/courses](http://localhost:8080/school/cousres).

Build your service module. Look at the \*.class files IntelliJ created in the *../out/production/service/* folder: in the next steps you will copy-paste these \*.class files (and their packages) to Tomcat.

1. Create the folder/files structure in Tomcat/webapps like shown in the screen shot shown below. Copy the folder/package structure with .class files as shown in the screenshot below. Do not copy *…/Publisher.class* file because you are not publishing the service from your code anymore.
2. Copy all *jersey* and *javax.activation* .jar files directly to the “lib” folder (do not make any subfolders in the “lib” folder). Do not copy the *jersey-container-jdk-http* because you are not publishing the service from your code anymore.



You will also need to create a **web.xml** file with the following content. Note that for “param-name” jersey.config.server.provider.packages you need to specify the Java package where you created your “resources” classes (StudentsResources and CoursesResources).

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_4\_0.xsd"  version="4.0"**>  <**display-name**>This is my first real service</**display-name**>  <**servlet**>  <**servlet-name**>Jersey REST Service</**servlet-name**>  <**servlet-class**>org.glassfish.jersey.servlet.ServletContainer</**servlet-class**>  *<!-- Register resources (and providers) under your sot.rest.service.endpoint java package. -->* <**init-param**>  <**param-name**>jersey.config.server.provider.packages</**param-name**>  <**param-value**>fontys.sot.rest.service.resources</**param-value**>  </**init-param**>  <**load-on-startup**>1</**load-on-startup**>  </**servlet**>  <**servlet-mapping**>  <**servlet-name**>Jersey REST Service</**servlet-name**>  <**url-pattern**>/rest/\*</**url-pattern**>  </**servlet-mapping**> </**web-app**> |

After you start Tomcat (e.g., tomcat/bin/startup.bat), your endpoints should be available at <http://localhost:8080/school/rest/students> and <http://localhost:8080/school/rest/courses>. The url of your service is built as follows. The beginning is always <http://localhost:8080/...> because it is Tomcat url. Further, you add the following to the Tomcat url:

* ../school/.. is the name of the folder you created in folder tomcat/webapps;
* …/rest/... is the <url-pattern> you defined in web.xml; and
  + …/students is StudentsResources endpoint;
  + ../courses is the CoursesResources endpoint.

In general, the executables of a web service should be compressed in a so-called WAR file. This is a WebARchive (WAR) file, which is a special type of JAR (Java ARchive) file. You can create a .war file of your service by compressing it to a single “school.war” file by using following command (go in Command Prompt to the folder where your WEB-INF folder is):[[2]](#footnote-3)

jar -cvf school.war \*

You can now open sc*hool.war* file with, e.g., 7zip and see it’s contents.

Instead of creating the school folder in the *Tomcat/webapps*, you can copy file *school.war* to *Tomcat/webapps*. After you start Tomcat, you can see that Tomcat automatically decompressed *school.war* and now there is *school folder* in *Tomcat/webapps*.

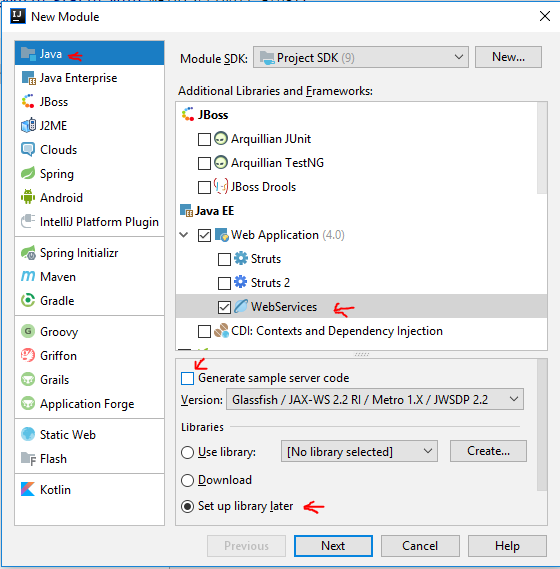
## Develop Tomcat service in Intellij

In this section you will learn how to make your service module as a Java Web module in IntelliJ, and then deploy service on Tomcat from IntelliJ[[3]](#footnote-4). Before you can work in your service and deploy it in IntelliJ, you must remove the “school” folder from “tomcat/webapps”. This is because you might have problems when the same service is deployed on Tomcat twice at the same time: (1) once manually as a folder in “webapps” and (2) once from IntelliJ.

In this part of the practicum you will make a Java Web module for your Service in IntelliJ. When you use this type of module, IntelliJ can automatically create the folder/files structure which is used in Tomcat and automatically deploy it on Tomcat. That way you can easily test how your service works on IntelliJ while programming. After you finish programming your service, you can deploy it permanently on Tomcat manually.

### Create a new WebServices module in IntelliJ

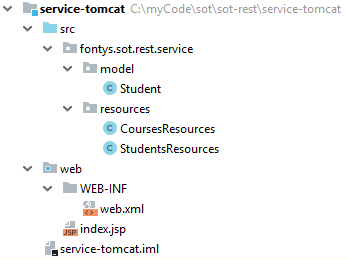
Create a new Java/Java EE/Web Application/WebServices module for your service as follows:



In order to be able to deploy your service on Tomcat, you need to create the following in your module:

1. **Source code.** Just copy the “resources” class from the Service of the previous week to the source folder of this module.
2. **Libraries.** In order to work with JAX-RS in you need to add Jersey and javax.activation libraries to this module, in the same way you did in the previous week.
3. **web.xml** Delete file sun-jaxws.xml from the web/WEB-INF/ folder, and create file web.xml as shown in Section 3.2.

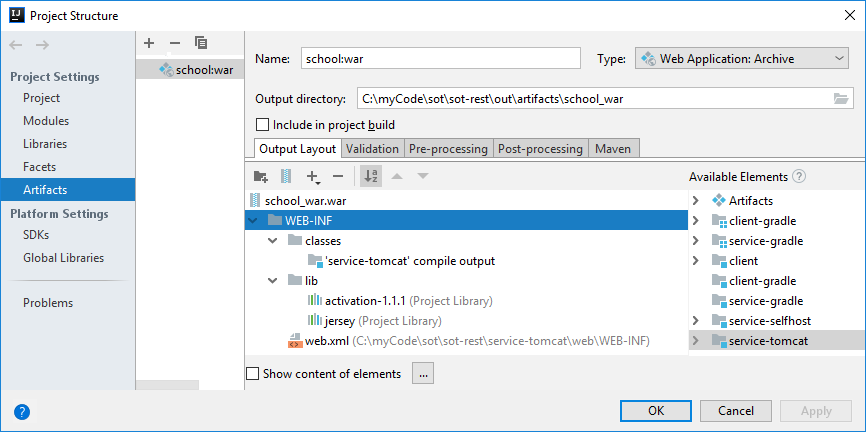
Once created, your IntelliJ module should look similar like shown below:



### Create and build an .war artifact in IntelliJ

Now you should build your project, so that IntelliJ creates .class files from your Java code. After you have built your project, you need to create an “artifact” (click on Project Structure to create an “war explaoded” or “war” artifact), which will be later deployed on Tomcat. This artifact must have the exact folder/files structure which is required by Tomcat (see Section 3.2):

1. In WEB-INF folder create the following:
   1. “classes” folder should contain all “executables”, i.e., compiled .class files of your code
   2. “lib” folder should contain all libraries your service uses (e.g., jersey, javax.activation, etc.)
   3. the “web.xml” file



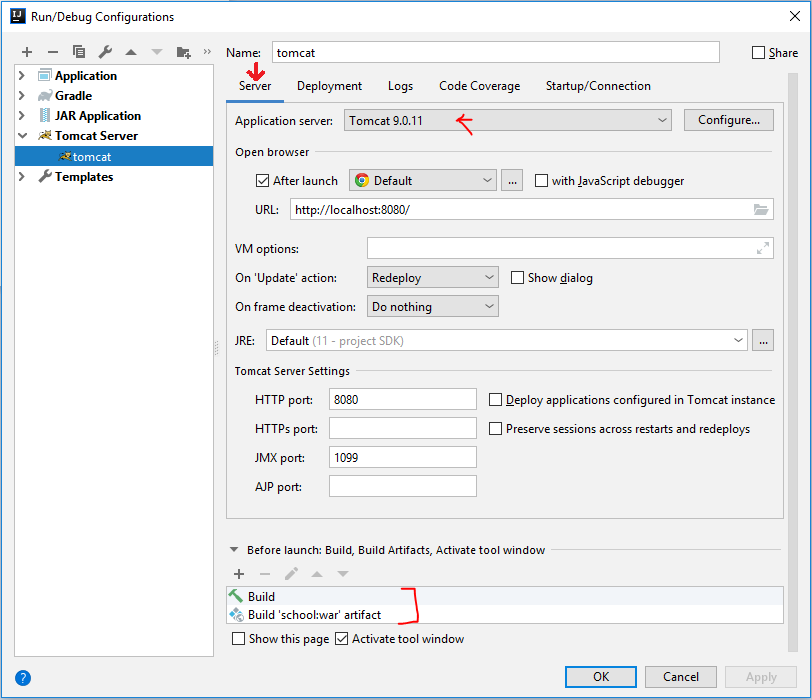
After you created the artifact, you can build it by going to “Build/Build Artifacts” in the main menu of IntelliJ. If this succeeds, you can find the artifact in “out/artifacts” folder of your IntelliJ project[[4]](#footnote-5). This is a .war (WebArchive) file: a special type of .jar file. You can open this /war file with, e.g., 7zip: you will see that it has the structure you specified. This .war file is actually the executable of your service:

1. you can deploy it to Tomcat from IntelliJ (as explained in the remainder of thiss section), or
2. you can copy it directly to Tomcat/webapps folder: when you re-start Tomcat the .war file will be automatically “extracted” in the Tomcat/webapps. Your service will then be permanently deployed on Tomcat, as explain in Section 2.2.

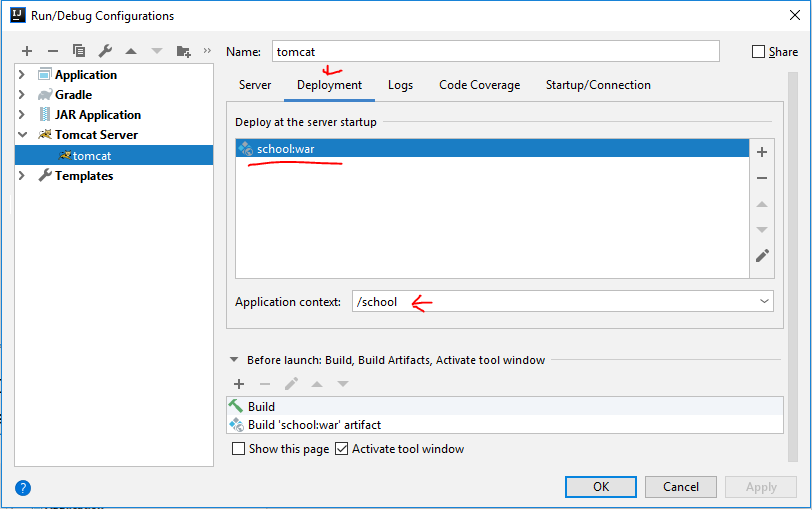
### Create Tomcat Run Configuration in IntelliJ

Now you can create e Run Configuration in IntelliJ which start-up Tomcat and will deploy the .war artifact on Tomcat.

You should add a Tomcat Server (local) run configuration:



On the Deployment tab, specify that you want to deploy your artifact[[5]](#footnote-6). Specify “school” as application context, which will be the name of the service:



Now you can run your tomcat Run Configuration to see if your service runs on Tomcat. You can also run your client from week 1 (remember to change the URL in the client code) to test how the client works with the service deployed Tomcat: client should work without any problems.

1. Form parameters can be used in PUT operations in the same way. [↑](#footnote-ref-2)
2. For more information about JAR files see <https://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html>. [↑](#footnote-ref-3)
3. Note that in this part of the practicum you will make only a new module for the Service. There is no need to make a new Client module, you can use the Client module from the previous week. [↑](#footnote-ref-4)
4. Note that you could also deploy your service manually on Tomcat by copy-pasting this artifact folder to “your\_tomcat\_installation/webapps” folder. [↑](#footnote-ref-5)
5. Note that, if you are making several services (have several service modules in your project), you can add them as different artifacts, with different url in the “Application context” parameter. That way you can run all services at Tomcat at once from IntelliJ. [↑](#footnote-ref-6)