JAX-RS: Response & Gradle

In this practical we will learn how to build a custom Response in the service and how to use Gradle build for dependency management.

Table of Contents

[1 Create custom Response 1](#_Toc19019774)

[1.1 GET operations with a custom Response 2](#_Toc19019775)

[1.2 PUT, POST and DELETE operations with a custom Response 3](#_Toc19019776)

[2 Using Gradle 3](#_Toc19019777)

[2.1 Create a Gradle module for service 3](#_Toc19019778)

[2.1.1 File build.gradle 5](#_Toc19019779)

[2.2 Create a Gradle module for client 7](#_Toc19019780)

[2.2.1 File build.gradle 8](#_Toc19019781)

[2.3 Some more Gradle things 9](#_Toc19019782)

[2.3.1 JavaFX Gradle plugin 9](#_Toc19019783)

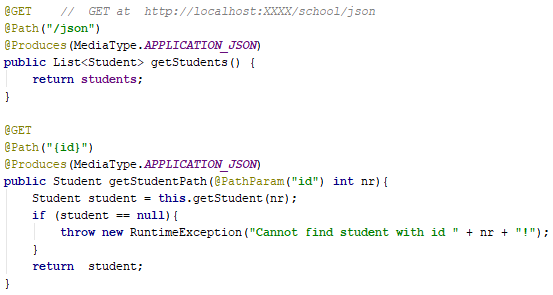
[2.3.2 Create serviceModel.jar file in the service and add it to client 9](#_Toc19019784)

[2.3.3 Run your client via gradle (from InteliJ) 10](#_Toc19019785)

[2.3.4 Create executable client.jar file for the client module 10](#_Toc19019786)

# Create custom Response

Sometimes you want to throw exceptions in the code of your service. For example, you what to throw exception when the a client tries to read (GET) a student who does not exist:



If an exception is thrown in the service, the status of the http response will be 500. If you host your service on Tomcat, then the entity of this http response message will be the whole stack trace of the exception! Try to call this GET method in the browser for a non-existing student number and see the entity of the response message!

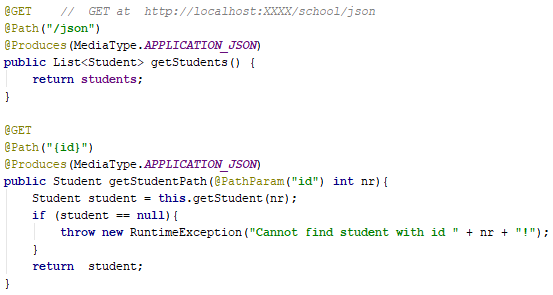
When you call this GET operation from a Java client, you want to show understandable error to the human user. So, you read the entity of the response message as a string, and show it to the user: the whole stack trace will be shown as the error message to the user :( !

If you want the service you return a http response with only your nice message in the entity, you must build your custom Response object in the service code. In following sections you can see how to build custom response for GET, DELETE, PUT and POST operations.

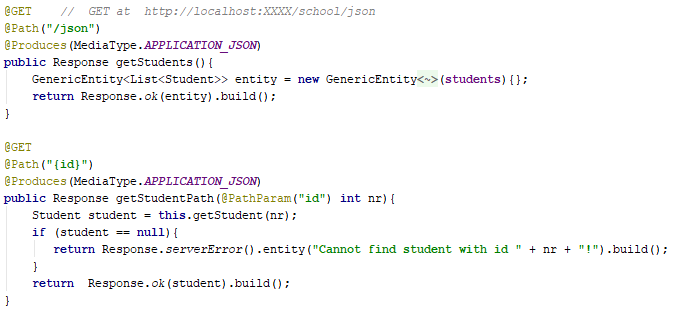
## GET operations with a custom Response

Below you can see few examples on how to change your GET operations when you want to build your own Response object. Note that in GET operations service returns a http response with an entity (e.g., Student object serialized to json, a String value, int value, etc.)

Before:

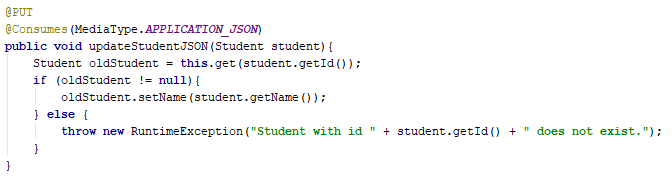


After:

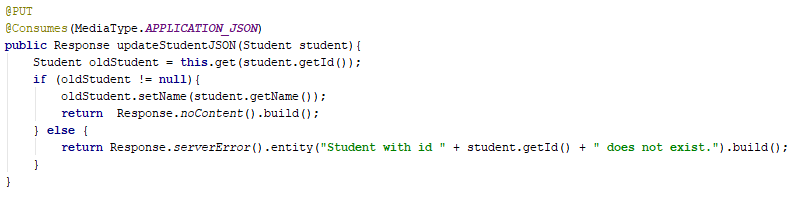


## PUT, POST and DELETE operations with a custom Response

Below you can see an example on how to change your PUT, POST and DELETE operations when you want to build your own Response object. Note that in PUT, POST and DELETE operations service returns a http response without entity (NO\_CONTENT). Below you can see an example how to change one PUT operation. You should change POST and DELETE operations in the same way.

Before: 

After:



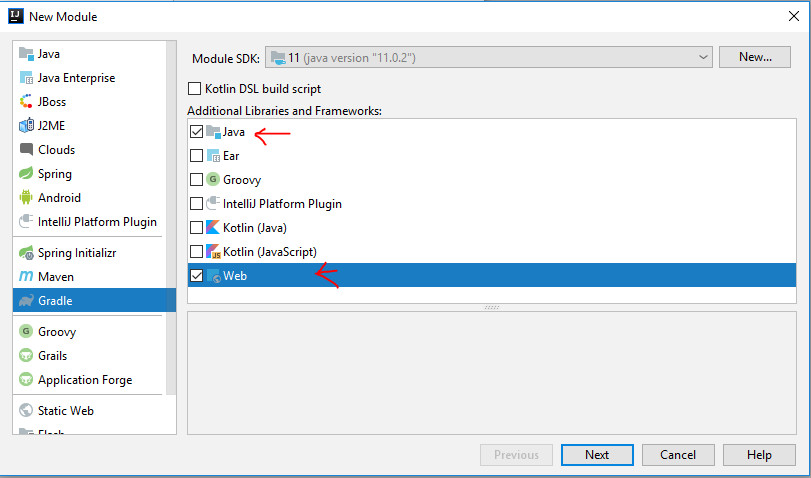
# Using Gradle

Gradle is a build tool which you can use for many things: building projects, dependency management, execution of unit tests, etc. Although you can install Gradle as a stand-alone application and use it from Command Line, in IntelliJ you can use Gradle for your projects without explicitly installing Gradle on your computer. So, you do not need to install Gradle now! You can read more information about Gradle at <https://gradle.org/>.

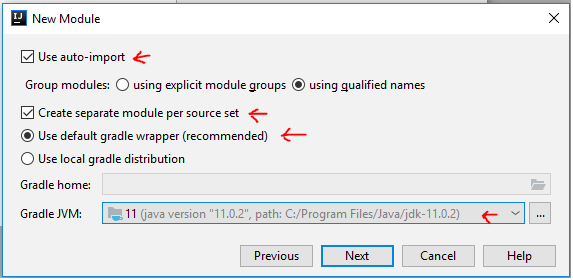
In SOT we will use Gradle mainly to manage dependencies (i.e., Gradle will automatically download all libraries you need). We will use Gradle in both Service and Client modules. Start by creating an Empty Project in IntelliJ. In following sections you will add one Gradle module for the service and one Gradle module for the client.

## Create a Gradle module for service

Create a Gradle module with Java and Web “libraries” (i.e, both java plugin and war plugin), as shown in the screen shot below. Enter “rest-gradle” as groupID, and “service” as artifactID.



Set-up this Gradle module as follows:

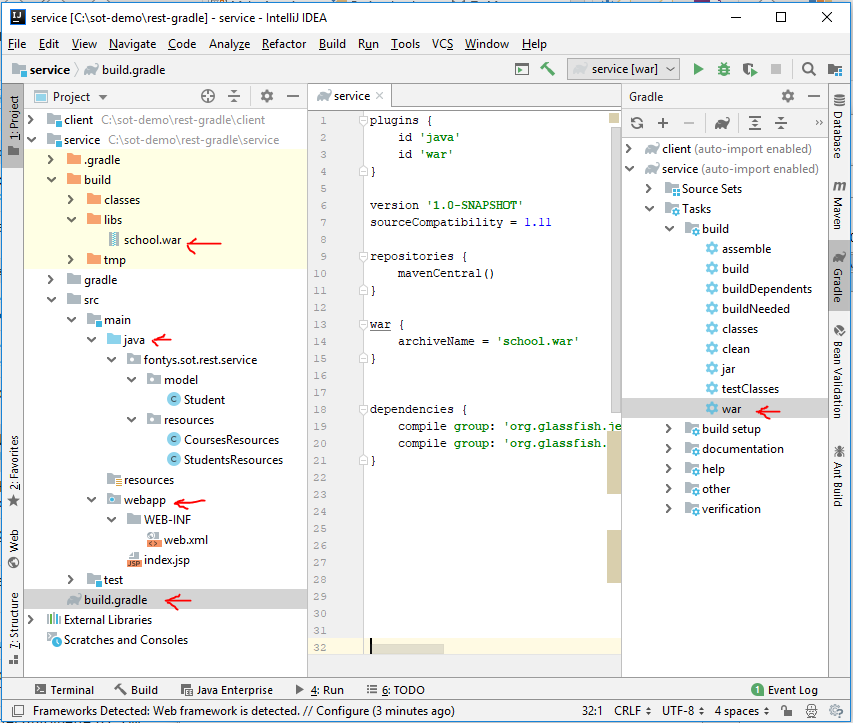


Note that your Gradle module must have a certain folder structure (folders for source code, test code and resources) required by Jar and War plugins. You can read more about this at <https://docs.gradle.org/current/userguide/java_plugin.html> and <https://docs.gradle.org/current/userguide/war_plugin.html>. IntelliJ will create the correct folder structure for you automatically.

Below you can see how your service module will look line in IntelliJ. Do the following:

1. Add your source code to folder src/main/java.
2. In src/main/webapp create WEB-INF/web.xml file.
3. Open build.gradle file. It only has dependencies for Unit tests (jnuit). You can delete these dependencies if you do not want to make unit tests. Your source code does not compile now because dependencies for Jersey are not added yet (in Section 2.1.1 it is explained how to add necessary dependencies).

(a) Contents of the build.gradle file, (b) Gradle task “war” and (c) how to create file *build/libs/school.war* are explained in the Section 2.1.1.



### File build.gradle

In file build.gradle you say to Gradle which dependencies to download for your module and specify some more settings.

First adjust sourceCompatibility the build.grade to match your Java version: for example sourceCompatibility = 1.11 says that you are using Java 11.

Now you need to add all necessary dependencies for build.gradle. You can search for dependencies on <https://mvnrepository.com> and copy-paste them from that page. For example, search for “javax.activation” (latest version) and you will find the Gradle line you have to add to your build.gradle. Below you can see all necessary Jersey dependencies for the service (use the latest versions):

* **jersey-container-servlet** (group org.glassfish.jersey.containers)
* **jaxrs-ri** (group org.glassfish.jersey.bundles)

Gradle will download these dependencies from mvnrepository (on some secret place on your computer) and use it when compiling/building your service module. If you double click Gradle task “war” (see the right side of the screen shot above), Gradle will compile your source code and create a **service-1.0-SNAPSHOT.war** file in build/libs. This war file is an archive file which has exactly the structure needed to deploy your service on Tomcat (open this **service-1.0-SNAPSHOT.war** file with 7zip or winrar an look at its structure).

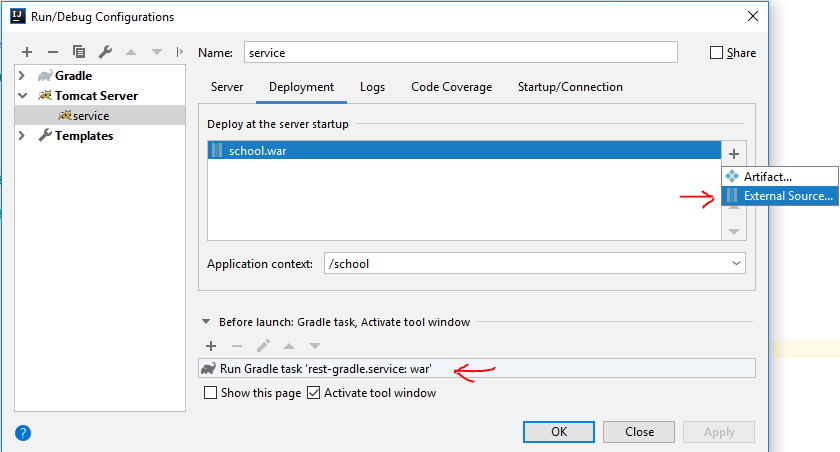
Now you can change the name of the .war file Gradle builds to be **school.war** by adding this to build.gradle (double click Gradle task “war” to test this):

war {  
 archiveFileName = **'school.war'**}

After you finish, your build.gradle file should look similar like this:

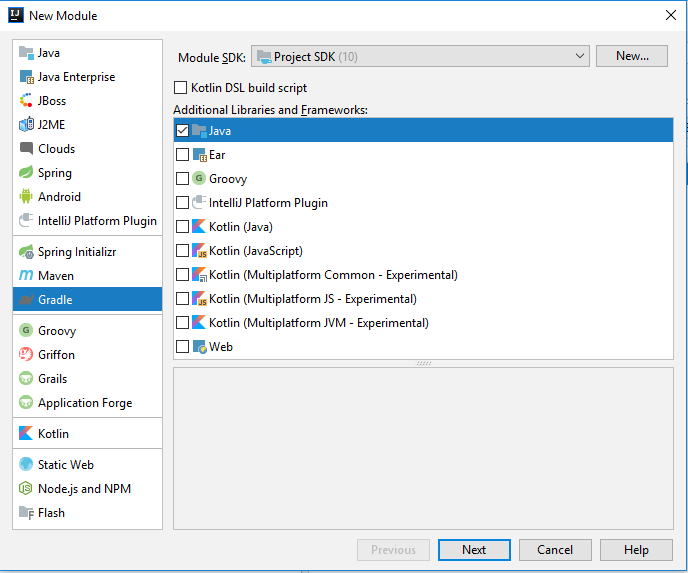
|  |
| --- |
| plugins {  id **'java'** id **'war'** }  version **'1.0-SNAPSHOT'** sourceCompatibility = 1.11  repositories {  mavenCentral() }  war {  archiveFileName = **'school.war'** }   dependencies {  compile **group**: **'org.glassfish.jersey.bundles'**, **name**: **'jaxrs-ri'**, **version**: **'2.28'**  compile **group**: **'org.glassfish.jersey.containers'**, **name**: **'jersey-container-servlet'**, **version**: **'2.28'** } |

You can deploy the build/libs/school.war file on Tomcat as External Source (in a similar way like deploying an artifact on Tomcat). Also specify that the “war” Gradle task in the service module should be executed before deploying on Tomcat as shown in the screenshot below:

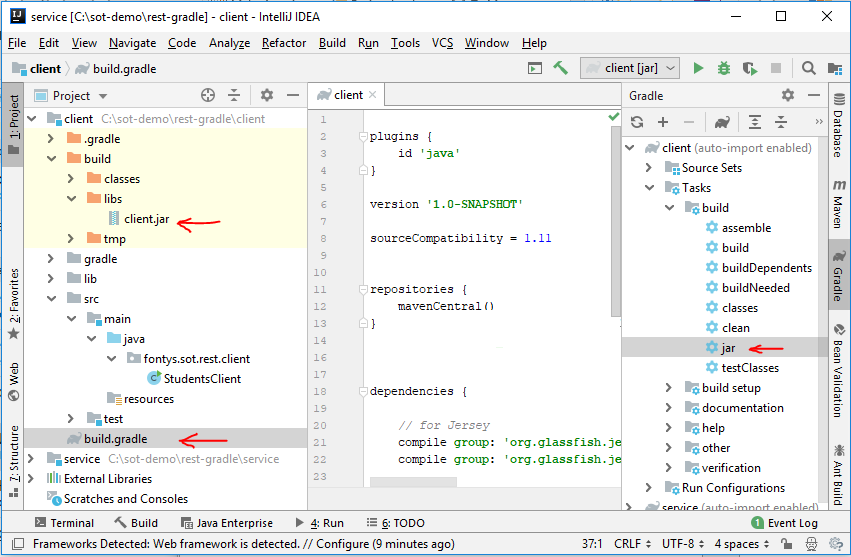


## Create a Gradle module for client

Add another Gradle module for the client, but now with only Java plugin. Enter “rest-gradle” as groupID, and “client” as artifactID:



Copy-paste your client code to src/main/java. Your client module should look like this:



(a) Contents of the build.gradle file, (b) Gradle task “jar” and (c) how to create file *build/libs/client.jar* are explained in the Section 2.2.1 of this document.

If you used the serviceModel.jar (with Student.class) in your client module, create folder called “lib” in the client module and copy-paste serviceModel.jar to it.

### File build.gradle

You should add following Jersey dependencies for the client (use the latest versions):

* **jersey-client** (group org.glassfish.jersey.core)
* **jaxrs-ri** (group org.glassfish.jersey.bundles)

Make the build.gradle file for the client like follows:

|  |
| --- |
| plugins {  id **'java'** }  version **'1.0-SNAPSHOT'** sourceCompatibility = 1.11  repositories {  mavenCentral() }  dependencies { compile **group**: **'org.glassfish.jersey.core'**, **name**: **'jersey-client'**, **version**: **'2.28'** compile **group**: **'org.glassfish.jersey.bundles'**, **name**: **'jaxrs-ri'**, **version**: **'2.28'** }  jar {  archiveFileName = **'client.jar'** } |

Now your client should compile: you can double-click Gradle task “jar” and “client.jar” will be created in client/build/libs. Open client.jar with 7zip or winrar and look at what is in it.

## Some more Gradle things

### JavaFX Gradle plugin

If you use Java version which does not contain FX, then you can use the "org.openjfx.javafxplugin" Gradle plugin to work with JavaFX 11+ <https://openjfx.io/>:

<https://github.com/openjfx/javafx-gradle-plugin>

### Create serviceModel.jar file in the service and add it to client

You can let Gradle create the **serviceModel.jar** file and include it in client module by:

1. In service module - create the serviceModel.jar file (with Student.class in it) and copy it to client/lib folder and
2. In client module - client/lib/serviceModel.jar as dependency in the client module

a) In service module - create the **serviceModel.jar** file and copy it to **client/lib** folder

Add two tasks shown below to the build.gradle of the service module:

|  |
| --- |
| *// create serviceModel.jar in directory /service/build/libs/ // with /fontys/sot/rest/model/Student.class* task createModelJar(**type**: Jar, **dependsOn**: compileJava) {  archiveFileName = **"serviceModel.jar"** from sourceSets.**main**.output  include **"fontys/sot/rest/model/\*"** println **"created serviceModel.jar from fontys/sot/rest/model/\*.class"** }  *// assuming modules are in directories “project/service/” and “project/client/”* *// copy “service/build/libs/serviceModel.jar” to “client/lib/serviceModel.jar”* task copyModelJarToClient(**type**: Copy,**dependsOn**:[createModelJar]) {   from **"**${buildDir}**/libs/serviceModel.jar"** into **"../client/lib/"** println **"copied "**+ **"**${buildDir}**/libs/serviceModel.jar"** + **" into "** + **"../client/lib/"** } |

b) In client module - include **client/lib/ serviceModel.jar** as dependency in the client module

You can do this by adding the following to build.gradle of the client module:

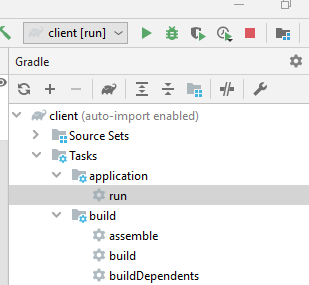
|  |
| --- |
| repositories {  mavenCentral()  flatDir { *//include all \*.jar files from local "lib" directory (e.g., serviceModel.jar)* dirs **'lib'** } }  dependencies {   *// all other necesasry dependencies Jersey, activation, genson …*  *//include all \*.jar files from local "lib" directory (e.g., serviceModel.jar)* compile files(fileTree(**dir**: **'lib'**, **includes**: [**'\*.jar'**])) } |

### Run your client via Gradle (from InteliJ)

If you want to run your application as a desktop application (console of GUI), you should use the ‘appplication’ gradel plugin. In order to do this, you need to add the following to build.gradle in the client module: (1) add the plugin and (2) set the main class:

|  |
| --- |
| plugins {  id **'java'**  id **'application'** }  …  …  application {  mainClassName = **'fontys.sot.rest.client.StudentsClient'**  }  …  … |

Now you can run the client application by executing task “run” of the application plug-in:



### Create executable client.jar file for the client module

Normally, you can create an executable .jar file with all dependencies in it in Gradle like this:

|  |
| --- |
| jar { *// make executable client.jar file and copy all dependencies in it* manifest {  attributes **'Main-Class'**: **'fontys.sot.rest.client.StudentsClient'** }  archiveFileName = **'client.jar'** *// add all dependencies to the executable .jar file* from { configurations.**compile**.collect { it.isDirectory() ? it : zipTree(it) } }  } |

You can now execute *java -jar client.jar* (in Command Prompt) and your client should be executed.