Make a simple JAX-RS service and client

In this practical you will learn how to make a REST(full) Service and Client using JAX-RS. Your service will be deployed from a simple console application. In this practical you will learn how to use GET http operations (only **Read** in CRUD: **C**reate**R**ead**U**pdate**D**elete) and path and query types of parameters.

Table of Contents

[1 Install Java JDK and IDE 1](#_Toc536544299)

[2 Create new project in IntelliJ with Service and Client modules 2](#_Toc536544300)

[2.1 Add Jersey dependencies to both Service and Client modules 3](#_Toc536544301)

[2.2 Setup the Service module 4](#_Toc536544302)

[2.3 Setup the Client module 6](#_Toc536544303)

[3 Returning values in GET operations 6](#_Toc536544304)

[3.1 Returning an integer 7](#_Toc536544305)

[3.2 Returning objects in JSON format 8](#_Toc536544306)

[3.2.1 Returning a single object 8](#_Toc536544307)

[3.2.2 Returning a (generic) collection of objects 9](#_Toc536544308)

[4 Using parameters in GET operations 10](#_Toc536544309)

[4.1 Using a query parameter 10](#_Toc536544310)

[4.2 Using a path parameter 11](#_Toc536544311)

[4.3 Using multiple query and/or path parameters 11](#_Toc536544312)

[5 Look at the HTTP messages 13](#_Toc536544313)

# Install Java JDK and IDE

Before you start with the practical you need to install Java JDK and IDE:

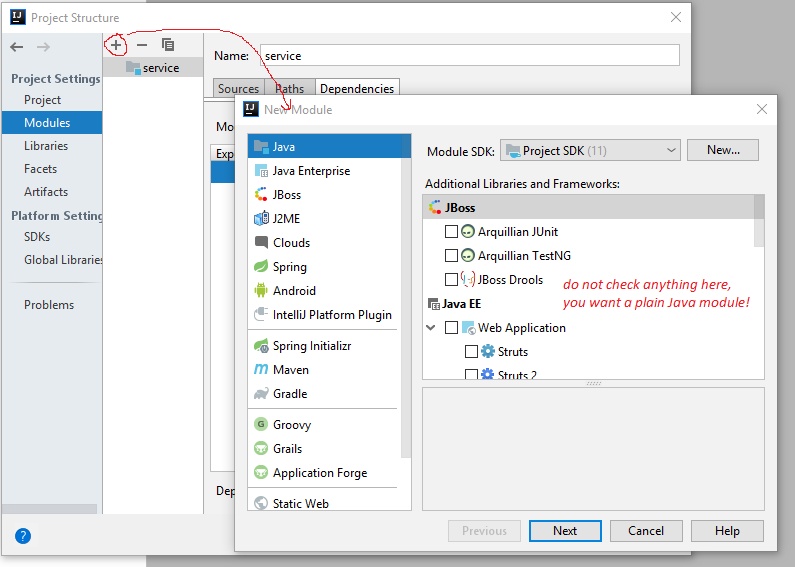
* In SOT module all demos and practical assignments use IDE IntelliJ IDEA Ultimate Edition (Community Edition does not work for SOT programming). The only way to get Ultimate Edition for free is to first register for a student account at <https://www.jetbrains.com/student/> using your [123546@student.fotnys,nl](mailto:123546@student.fotnys,nl) email address. *If you do not want op create such an account, you may use NetBeans or Eclipse as IDE. Note that, if you use NetBeans or Eclipse, it might happen that teacher is not able to help you with IDE-related issues.*
* IntelliJ installation directory contains a java JDK installation in directory “jbr” (e.g., *c:\Program Files\JetBrains\IntelliJ IDEA 2019.2.1\jbr*). You can either use this Java JDK or install manually the newest version of OpenJDK. Do not forget to (a) create the JAVA\_HOME environment variable for the JDK you want to use (e.g., *c:\Program Files\JetBrains\IntelliJ IDEA 2019.2.1\jbr* or the one you installed manually) and (b) add “*c:\Program Files\JetBrains\IntelliJ IDEA 2019.2.1\jbr****\bin***” (or your own “JDK/bin”) to the Path environment variable.

# Create new project in IntelliJ with Service and Client modules

Open IntelliJ and create a new **Empty Project**. Then go to Project Structure, and set the SDK (create new SDK configuration) on the **Project** tab:

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

Next got to the **Modules** tab of the **Project Structure** [[1]](#footnote-1)and add two **Java** **modules** (do not check anything on the right side of New Module screen) - one for the service and one for the client:



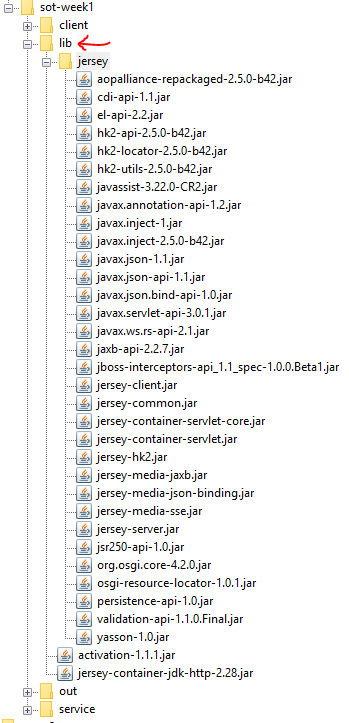
## Add Jersey dependencies to both Service and Client modules

In order to work with JAX-RS in both service and client modules you need to add dependencies to following libraries.

**A) First create a “lib” folder in your project, download following libraries and copy them to the lib folder:**

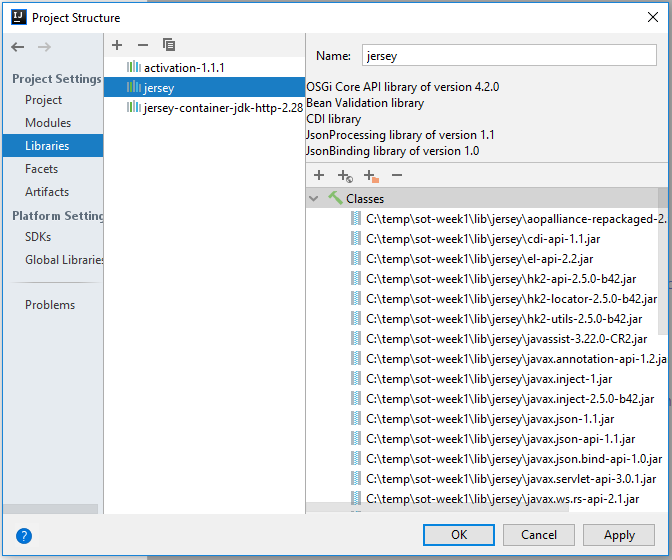
1. Jersey is a framework which implements the JAX-RS API: <https://jersey.github.io/> Download the JAX-RS bundle and unzip it. It has several sub-folders with many .jar files. Copy all .jar files (from all sub-folders) directly to the “lib/jersey” folder of your project .
2. Download javax.activation library from <https://mvnrepository.com>. Copy *activation-xxx.jar* file directly to “lib” folder.
3. Download “jersey-container-jdk-http” library from <https://mvnrepository.com>. Copy *jersey-container-jdk-http-xxx.jar* file directly to “lib” folder.

Your lib and lib/jersey folders should look like this:



**B) Add these files as libraries to your IntelliJ project and Service and Client modules:**

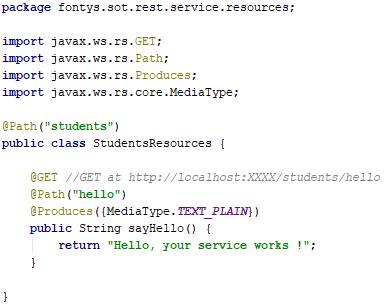
In Project Structure, go to Libraries tab and add three libraries (as Java libraries) by selecting (all) \*.jar files from your lib and lib/jersey folders as shown below. *Jersey* and *activation* should be added to both Service and Client modules and *jersey-container-jdk-http* only to Service module:



## Setup the Service module

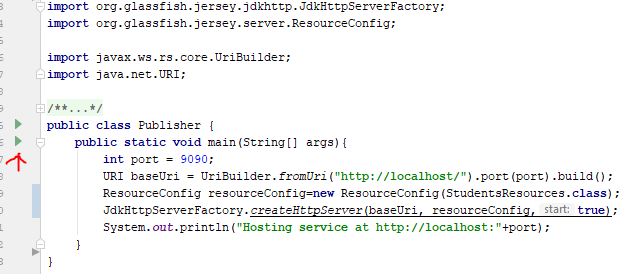
Service module will be a console application. Add the following two classes: (A) class **StudentsResources** will be the end-point and (B) class **Publisher** will start-up the service.

A) First, create a new **class StudentsResources** in package **fontys.sot.rest.service.resources** which will be the “service end point”, i.e., methods in this class will be accessed by client applications:

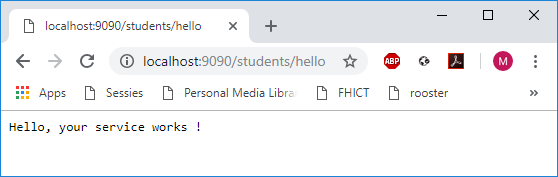


Assuming that you will host your service on <http://localhost:9090> and you add annotation **@Path(“students”)** to this class, all methods in this class will be accessible via [**http://localhost:9090/students/**](http://localhost:9090/students/)... For now, there is only one method with **@Path(“hello”) annotation**: it will be accessible to clients via GET http request on <http://localhost:9090/students/hello>. This method produces a plain text in http response.

B) Second, add a **Publisher class** which will start-up the service and host the StudentsResources. Here you use the “jersey-container-jdk-http” library to host the service:



Now you can deploy your service (host it) in IntelliJ by either (a) directly running the StudentsResources class on the green “run” button next to he main method, or (2) you can add a Run Configuration (type Application) which will run this class. When you run the Service application (Publisher class), you can access its GET methods from your internet browser. For example, in your browser you can test the first GET operation (for method sayHello) if you go to <http://localhost:9090/students/hello>:



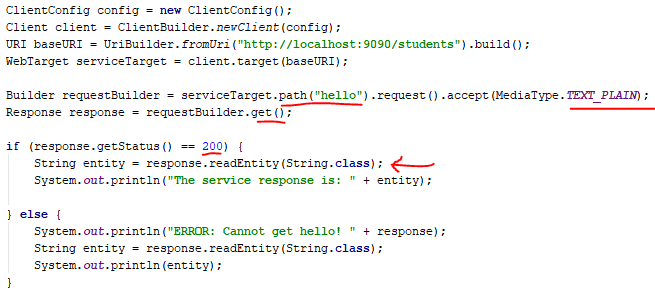
If you want to see whole raw HTTP messages (request and response) which are exchanged between the service and the client (i.e., your browser), you may use the Burp Suite tool like explained in Section 5. Below you can see how the http request and response messages look like when calling this method:

|  |  |
| --- | --- |
| (a) http request message  (sent from the client/browser to the service) | (b) http response message  (send from the service to the client/browser) |

## Setup the Client module

Client can be a console or gui (fx or swing) application, you may choose the type of application you like the most. Add one class with an empty main method.

Below you can see how you can call the GET operation from your Client module. Remember that you needed to add Jersey and javax.activation libraries dependencies to the Client module (see Section 2.1 ).



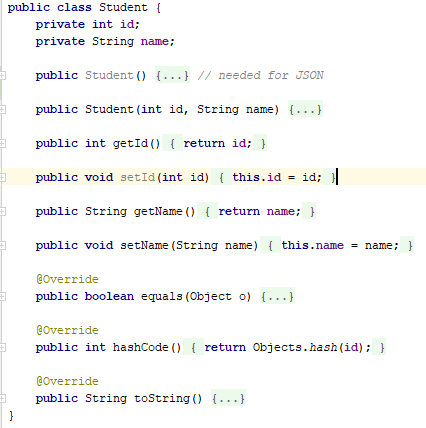
# Returning values in GET operations

All GET operations return some value. This value is returned by the Service in the entity (i.e., body) of the http response message. In the previous section you saw how to work with methods which return values/entities as String. In the following few sections you will see how to work with methods which return values as (1) integer and (2) JSON serialisation of one object (e.g., Student object or Array List<Student> object). Below you can see examples of four http response messages with different types in the message entity:

|  |  |
| --- | --- |
| (a) string entity | (b) json representation of a single Student object entity |
| (c) integer entity | (d) json representation of Students collection entity |

## Returning an integer

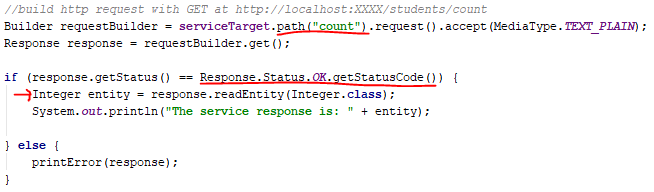
In order to be able to add somewhat more meaningful functionality to the Service, we first need to add a new class **Student** in package **fontys.sot.rest.service.model** as follows (start by creating Student class only with private fields, and then use Alt+Insert to let IntelliJ automatically create getters, setters, constructors, etc.):



Now you can add a list of several hard-coded students to the StudentsResources class. Also add a GET method which returns the size of the list, and has annotation @Path(**"count"**), like shown in the screen-shot below. Test the new GET method in your browser at <http://localhost:9090/students/count>:



Below you can see the Client code to call this GET operation, and reads the entity of the http response message as an Integer:

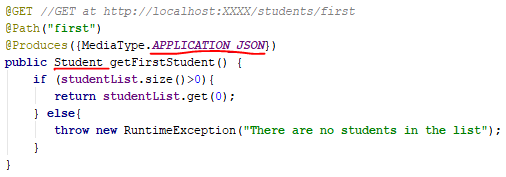


## Returning objects in JSON format

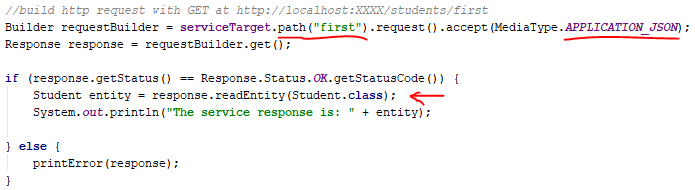
When you want to return objects in the response of the GET operation, you will need to serialize them in the JSON format. We will first add one method which returns a Student object and later one which returns the a list of all students.

### Returning a single object

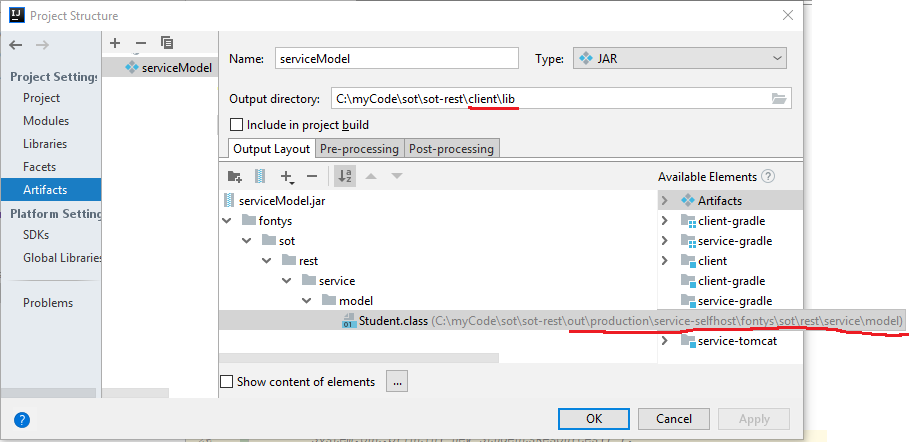
Add a GET method in the Service which returns the first student object from the list as JSON at <http://localhost:9090/students/first>:



Below is the Client code to call this GET operation. Note that now you have to ask that the entity of the response should be formatted in the JSON format:



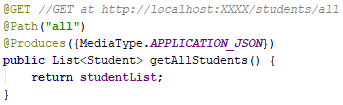
Note that your Client module will not compile because Student class in unknown. You can solve this in two ways: (a) manually re-create a new Student class in the Client module – it must have the same name of fields, getters, setters and a default constructor, or (b) import Student class from the Service module by creating one artifact as “empty” JAR file with *all/necessary/packages/Student.class* (you can find *Student.class* file in folder *out/production/service/…*) and adding this serviceModel.jar file as a library in the Client module (save the model.jar file in the lib folder in Client module). Next, you will need to add this serviceModel.jar file **dependency** to the Client module, on the **Modules** tab in **Project Structure** dialog.



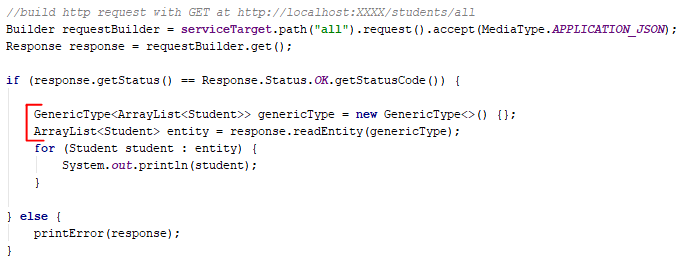
After you created the artifact, you can build it by going to “Build/Build Artifacts” in the main menu of IntelliJ. If this succeeds, you can find the artifact in “Output directory” which you specified.

### Returning a (generic) collection of objects

Add a GET method in the Service which returns the whole list of students at <http://localhost:9090/students/all>:



The Client code to call GET operations which return a GENERIC type (e.g., ArrayList<>) is somewhat more complex:



# Using parameters in GET operations

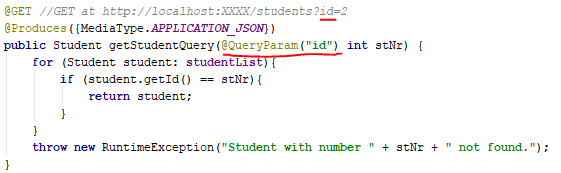
In Section 3.2.1 you made a GET operation which returns the first Student object in the list. In this section you will make a new GET operation which returns a Student object with a certain ID, which is given as a parameter.

In GET operations parameters are passed from a client to the service in the URL of the http request. In this section you will learn how to use path and query parameters, because these two types of parameters are used in GET operations[[2]](#footnote-2). Both path and query parametrs are a part of the URL of the http request. Below you can see two examples of URLs containing the ID value as a parameter:

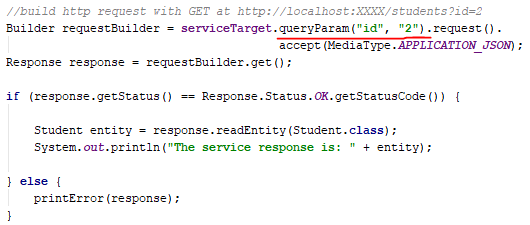
* As a query parameter: <http://localhost:9090/students?id=3>
* As a path parameter: <http://localhost:9090/students/3>

## Using a query parameter

In order to pass ID as a queue parameter, we will make a GET operation in the Service on URL <http://localhost:9090/students?id=3>:

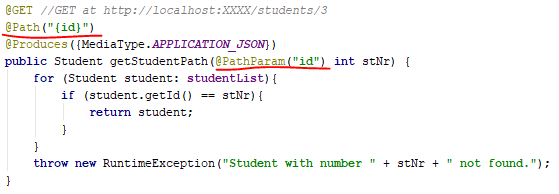


The Client code to call this method is:

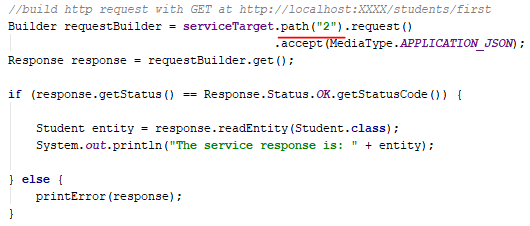


## Using a path parameter

In order to pass ID as a path parameter, you will make a GET operation in the Service on url<http://localhost:8080/students/3>. You need to add (1) the parameter name between curly brackets in the @Path annotation and (2) @PathParam annotation next to the parameter itself (parameter name in both annotations must be the same). This method will be accessible at [http://localhost:9090/students/{id}](http://localhost:9090/students/%7bid%7d), where {id} should be an int specifying the id. For example, <http://localhost:9090/students/2> will return the name of student with id value 2 (test this in your browser).

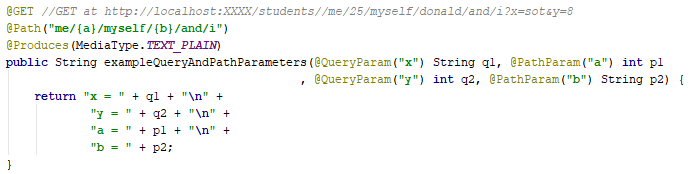


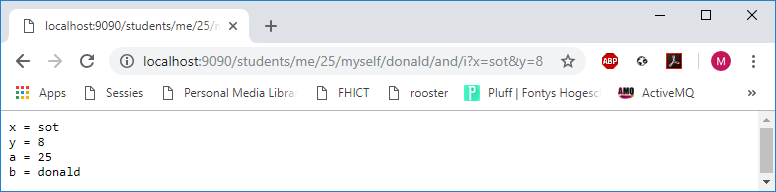
The Client code to call this GET operation is as follows:



## Using multiple query and/or path parameters

Multiple path and query parameters can be combined in one method as you like. Note that url names of parameters do not have to be the same like the parameter names in your source code. For example, you can make one GET method with 2 path parameters and 3 query parameters. Try the following:





# Look at the HTTP messages

Originally, the Service is published at port 9090, and all communication between the service and the client (exchange of HTTP messages) goes through this port:

9090

You can use the Burp Suite tool (<https://portswigger.net/burp/> ) for monitoring the exchange of http messages on a specific port. If your client sends all its requests to the port of a “Proxy Listener” in the Burp Suite , then this tool will:

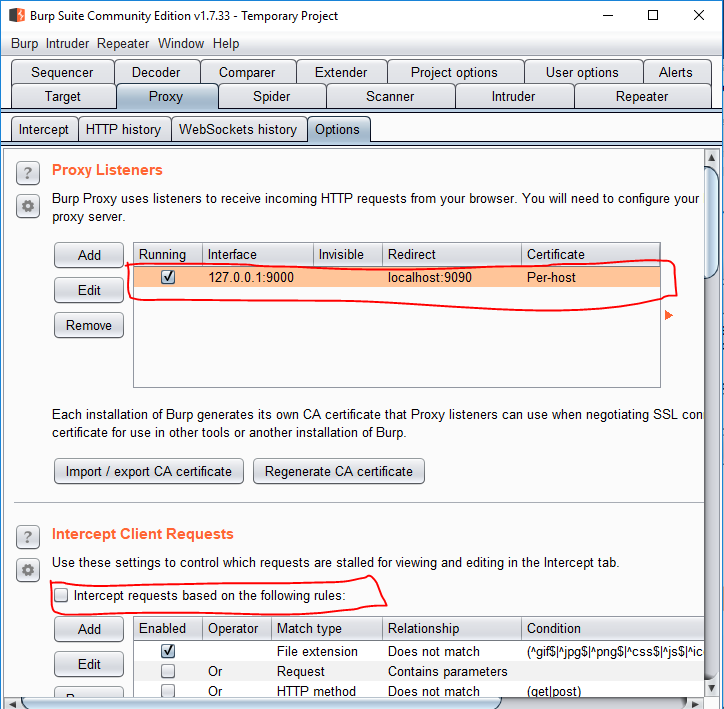
1. Receive the HTTP request message of the client;
2. Print the HTTP request message and forward it to the service;
3. Receive the HTTP response message of the service;
4. Print the HTTP response message and forward it back to the client.

9090

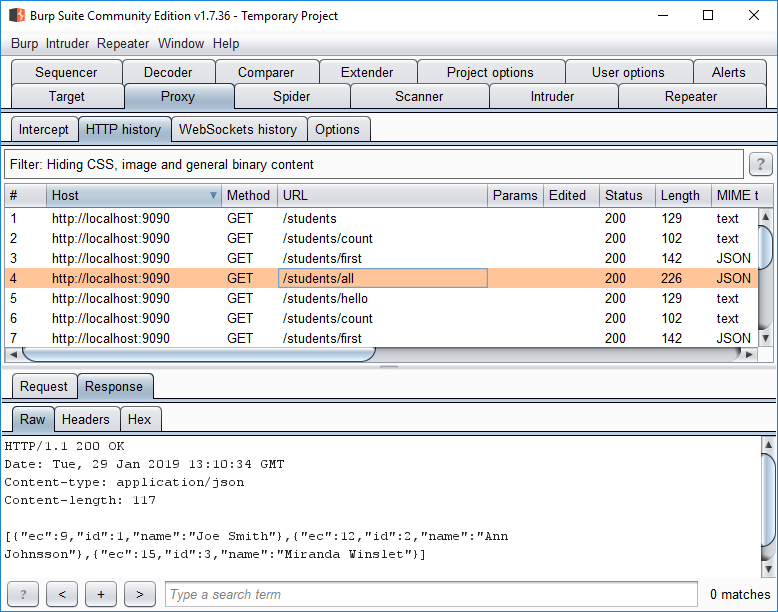
9000

Burp Suite Proxy Listener

Add a proxy listener in Burp as shown below. Make sure that rules for interception are disabled and the listener is running:



Now you can see the content of http request and http response messages on the HTTP History tab:



1. The most important settings of your project are in **File**->**Project Structure**. [↑](#footnote-ref-1)
2. Next week you will learn how to use the third type of parameters called “form parameters” in PUT and POST operations. [↑](#footnote-ref-2)