**Exercise 1 – Create a basic user REST API server with Dropwizard**

**What to do:**  
Create a User class with id, firstName, lastName.  
Create a UserResources class that contains the actual API + a collection (array/list/whatever) of Users in the system (emulates a DB).  
API methods: create, deleteById, findAll, update, findById.  
Any additional classes which might be required.

**Project creation and execution - technical notes:**

1. Create a Maven project (not Java project – which does not have a .pom file).

* Definitions:
* ***groupId*** *- uniquely identifies your project across all projects. A group ID should follow Java's package name rules. This means it starts with a reversed domain name you control. For example: org.apache.maven, org.apache.commons  
  You can create as many subgroups as you want. A good way to determine the granularity of the groupId is to use the project structure. That is, if the current project is a multiple module project, it should append a new identifier to the parent's groupId. For example: org.apache.maven, org.apache.maven.plugins, org.apache.maven.reporting*
* ***artifactId*** *- is the name of the jar without version. If you created it, then you can choose whatever name you want with lowercase letters and no strange symbols. If it's a third-party jar, you must take the name of the jar as it's distributed. eg. maven, commons-math, cloud\_dr*
* Additional details:
* groupId should start with org not com (org is used for private projects while com is for more public projects, with customers, or at least a website).
* groupId should be the path of packages in the project. Then a level of packages is added: Application, Resource1, Resource2… (the resources of the project – in this project – only 'User').  
  Under Application we will keep the main class of the project – the runnable.  
  Under each resource we will have a controller package and domain – that will have the User class.
* artifactId should pretty much be the name of the project – it will be the name of the jar (probably, not sure).

1. Create all classes in packages – not directories.
2. Adding plugins – done by only adding the correct section to the .pom (no need to download or import anything).
3. Execution:   
   Clean + Install the project.  
   Create a run/debug execution for the UserApplication class (the one with the main method).  
   Add the execution parameters in the 'Program arguments' field: 'server config.yml'
4. Once it's running – you can execute only GET request directly in the browser or use **Postman** (download from the internet) for all types of requests (in regular browser you cannot send any parameters).  
   Link: http://localhost:<port>/<path>  
   Port is taken from the config.yml – from the 'applicationConnectors' section.  
   Path is the path that is specified in @Path above the class name, appended by the @Path of the specific REST request you wish to use.  
   Example: <http://localhost:9179/users>

**Postman and annotations logic:**

1. **@Path** – the path above the class name, should be the type of the resource this API defines.  
   In the case of this exercise, the resource is a User – therefore the path of the class is "/users".  
   The path above each method will be appended to the path of the class when activating the request. The path of each method should signify the input of the method, *if it is simple (textual?)*.  
   For instance – if the input is 'id' (a string) – the path will include {id} in it. If the input is a more complex object, it will not be described in the path as a parameter with {}. **@Produces** - should be added above all methods that return something. It defines the data format in which the result must be returned.  
   **@Consumes** - should be added above all methods that have input parameters. Defines the format of the input that needs to be used in Postman (for example) in order that the REST method could be activated. If REST in Postman uses a different type – the method will not work.

* It's good practice to add the @Produces annotation in ALL the methods, since even when a method doesn't return a value, it could throw an exception – and we would like to have it returned.

1. In order for requests with @Produces/@Consumes to work, we need to define the input/output type in Postman – when executing the request.  
   This is done by going to the Headers tab of the request and adding either 'Context-Type' or 'Accept' as key, and the type that was defined in the method's annotation as value.  
   Examples:

* Method input handling:  
  If we have a POST request that gets some object/objects as input:   
  A possible annotation above the method: @Consumes(MediaType.APPLICATION\_JSON)  
  The KEY in the header: Context-Type  
  The value of the key: application/json
* Method output handling:  
  If we have a GET request that returns some object/objects as output:  
  A possible annotation above the method: @Produces(MediaType.APPLICATION\_JSON)  
  The KEY in the header: Accept  
  The value of the key: application/json

**Important note:**   
Methods can have either annotation or both – regardless of them being GET or POST – it is only dependent on whether they have input parameters and whether they return something.  
Whenever we define the ***@Consumes*** annotation, we need to add the ***Content-Type*** key in the header, and whenever we use the ***@Produces*** annotation we need to add the ***Accept*** key in the header.

**Jar  
  
Jar creation:**  
Jar is created when you run clean + package or clean + install.  
It's better to create the jar without the 'SNAPSHOT' part in its name.  
There are 2 ways to do that:  
- to change the version in the pom from the default: '<version>1.0-SNAPSHOT</version>' to '<version>1.0</version>'  
- to add <finalName>basic\_rest\_with\_dropwizard-1.0.jar</finalName> in the build section of the ppom.  
The second option is better practice, but notice it will create 2 jars – one with the snapshot and one with the final name.  
  
  
**Jar execution:**  
Quick execution from Git-Bash:  
Open Git Bash in the directory where the config.yml is, then run:  
java -jar target/<jar-name>.jar server config.yml  
  
Execute jar correctly (does not require having Git Bash):  
Copy the jar and the config.yml to some directory outside the project.  
Create a new text file and name it 'run.bat'.  
Edit the bat file with notepad and write the jar execution command in it:  
java -jar basic\_rest\_with\_dropwizard-1.0.jar server config.yml  
Save the bat file and double-click it to run the jar.  
To end the program – close the window that's opened when you run the bat file.  
When the project is final, and all the files in this directory are in their final draft, zip this directory and add it to the project, under the '\_Additional\_Information' directory that you should create in the same level as the config.yml.

**When you open the jar and look inside:**

1. META-INF directory contains the MANIFEST.MF file:  
     
   *Manifest-Version: 1.0  
   Archiver-Version: Plexus Archiver  
   Built-By: goldny  
   Created-By: Apache Maven 3.6.0  
   Build-Jdk: 1.8.0\_161  
   Main-Class: org.projects.application.UserApplication*

It describes the main class of the project, and other jars, in case the project is built from more than one jar.

1. All other directories (all but META-INF) represent dependencies in the project.  
   Every jar we added as a dependency is translated to a path of folders (folder for each package) - until the actual code.  
   The groupId is translated to a directory path inside the jar.  
   The packages match the groupId.

**Resources:**

1. The following 2 links were the most helpful for **writing the actual code of the REST requests**.  
   The second link was also helpful for **running the project** and **defining the pom file**.  
   <https://java2blog.com/dropwizard-tutorial/>  
   <https://howtodoinjava.com/dropwizard/tutorial-and-hello-world-example/>
2. This link had the same purpose, but was less helpful, since the example is less simple.  
   Might be more useful farther down the road.  
   It was however more **useful for running** the code:  
   <https://stackabuse.com/dropwizard-develop-restful-web-services-faster/>
3. This link helped me understand how to **create a maven project**:  
   <https://java2blog.com/how-to-create-dynamic-web-project-using/>
4. The following links were helpful for defining the **pom file**:  
   <https://www.javatpoint.com/maven-pom-xml>  
   <https://gist.github.com/13yo/1662915>
5. How to work with **Postman**:  
   <https://www.toolsqa.com/postman/post-request-in-postman/>
6. **HTTP protocol** – might be useful in the future:  
   <https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html>
7. @PathParam and @QuertParam:  
   <https://www.mkyong.com/webservices/jax-rs/jax-rs-pathparam-example/>  
   <https://www.topjavatutorial.com/webservices/rest/when-to-use-pathparam-and-when-to-use-queryparam/>