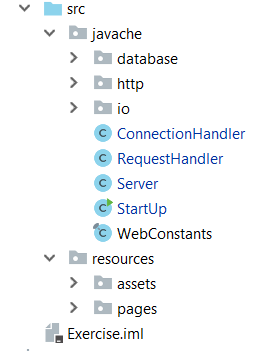
# Exercises: HTTP Protocol

Problems for exercises and homework for the [“Java Web Development Basics” course @ SoftUni](https://softuni.bg/courses/java-web-development-basics). Submit your solutions on the **course page** of the **current instance**.

Remember the mini HTTP Server we wrote during the lecture. Well, now you’ll extend it a little. This exercise will consist of a simple lab, at which you’ll rewrite what we wrote during the lecture, and extend it a little. It is highly advised that you create a new project from scratch, and just follow the steps.

## Project architecture

Like it was said above, create a new project, and create the following folders and architecture:



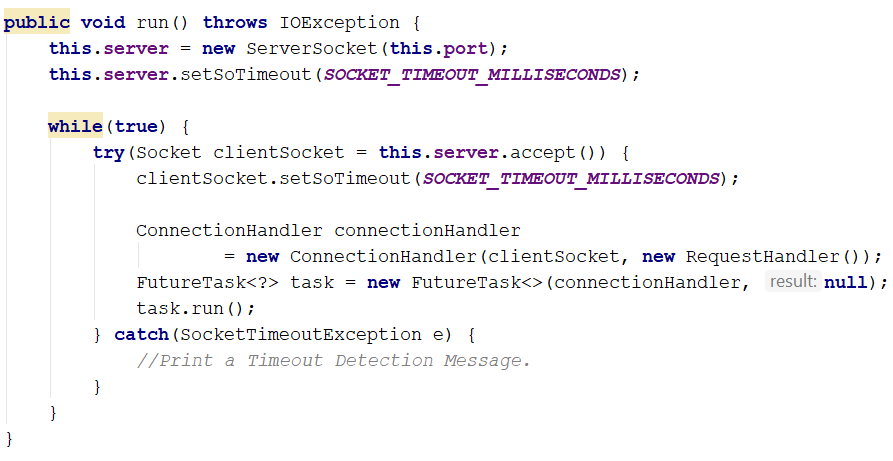
As you see we’ve put our server files in a folder corresponding to the Server’s name – **javache**. Now let’s look at the classes.

The StartUp class should be the holder of the main() method, where the Server class should be instantiated and ran.

## Server class

The Server class is the connection controller, it oversees the process with the **socket connections**. It should hold a ServerSocket and a port. The Server should have a run() method, which should instantiate the ServerSocket and it should create a **while loop** which **listens** for **connections**. Upon connection, the Server should send the **client socket** to a ConnectionHandler, which should take it from there.

Note: The Server should be **asynchronous**, so you’ll have to set some timeout on the Server and Client sockets. You should also make the ConnectionHandler extend the Thread class, so that you can @**Override** the run() method, and make a FutureTask, to make it **asynchronous**.



The SoTimeout is the **lifespan** of the **socket connection**. It won’t be good if the connection stays alive for too long, because both sides can send anything to the other, which would cause security issues.

When the lifespan **ends**, naturally, the **Server** will try to accept a **new Socket**, which may not be existent at that moment. That will **cause** an **exception**, which must be handled. There is really no way to handle this better, than just printing a **timeout detection message**, and **counting timeouts** for **example**, that’s why we have that nasty **try/catch**.

## ConnectionHandler

The ConnectionHandler class is the one that processes the client’s connection. It is the highest-level handler we will have in our simple HTTP server, for now …

The ConnectionHandler should have a RequestHandler, which should be passed through the **constructor**.

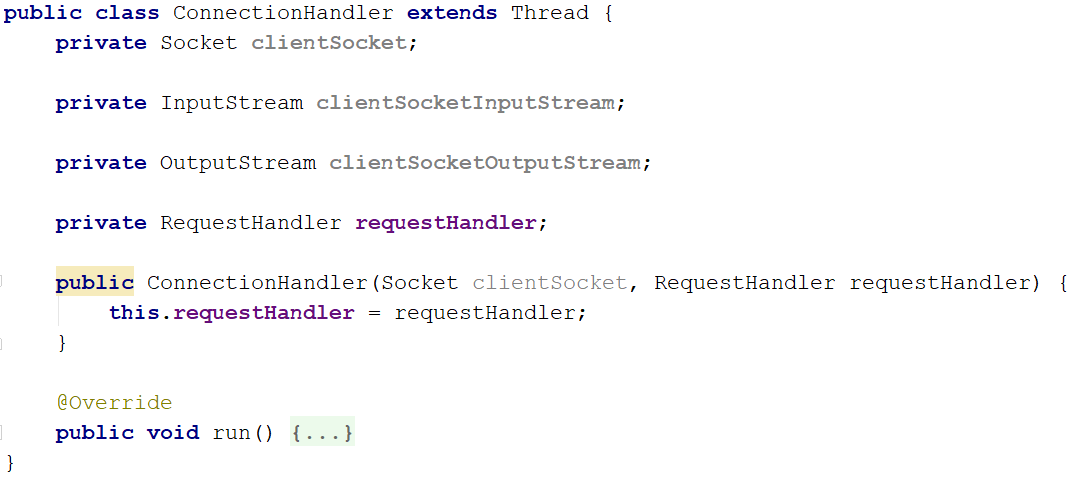
The ConnectionHandler should also hold a Socket, which is the clientSocket, which should also be passed through the **constructor**.

The ConnectionHandler should form an InputStream (**requestStream**) and an OutputStream (**responseStream**), from the Socket’s streams.

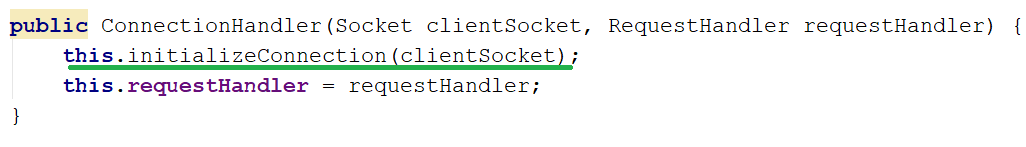
It should then extract the data from the InputStream (**requestStream**) and pass it to the RequestHandler. The RequestHandler should return a byte[] array of data, which should then be written by the **ConnectionHandler**, in the OutputStream (**responseStream**).

So let’s start building the class! This will be the **last major class** that will be given to you, the rest is up to you to be written. 😊 You will have to write it as you see it in your vision.

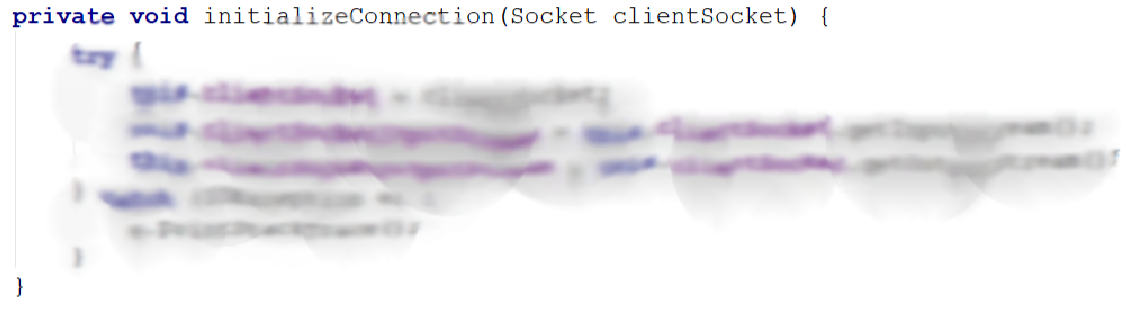
**NOTE**: Writing something in your way, without looking at a template or a helper, is the best way to learn that thing. When you come up with solutions to the problems, yourself, you discover knowledge, which would take a lot of time for you to learn elsewise.



We would probably need to initialize those socket elements somehow. But the constructor will get too heavy. Let’s add a new method for it:



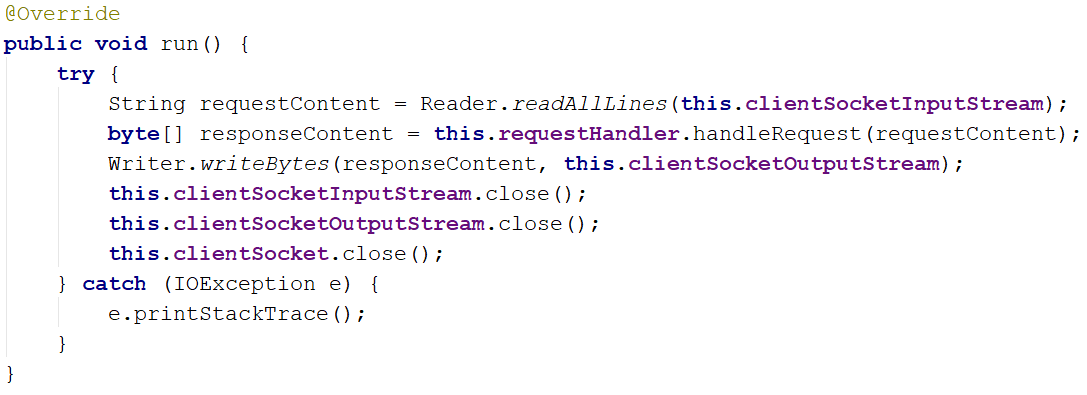
This method will initialize the clientSocket, the clientSocketInputStream and the clientSocketOutputStream. This one’s for you to implement.



The next thing you need to implement is the run() method. It’s not that hard, its actually pretty easy.

The run() method should extract the Request’s content through the InputStream, and pass it as a **single string**, with the lines separated by a line separator. It should be passed to the RequestHandler. The RequestHandler should return a byte[] array, which should then be sent as a Response, through the OutputStream.

Now let’s implement it by the logic, specified above:



As you see, everything is almost ready, we almost have a working server, and everything is tip top. But what are these Reader and Writer classes? Well, let’s check them now.

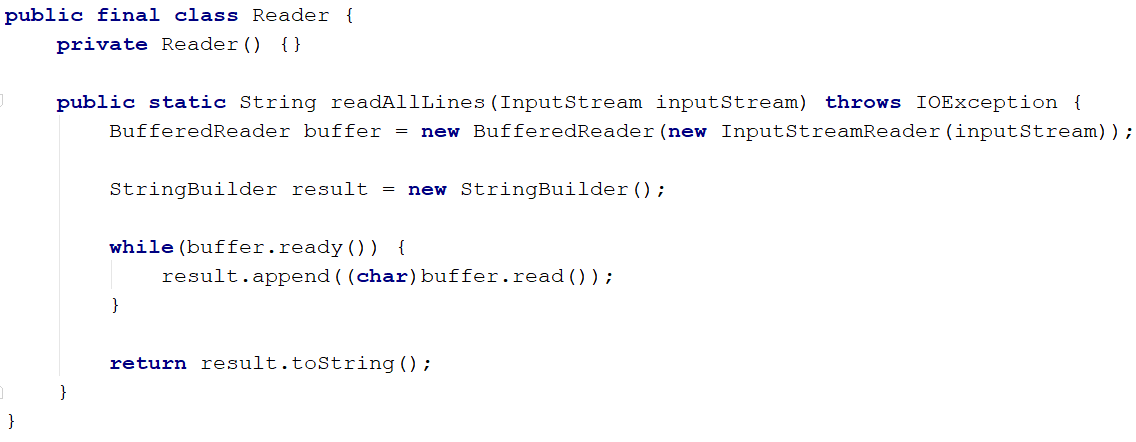
## IO Package

The IO Package will hold 2 static classes – The Reader and The Writer. These 2 will be utility classes, which we will use to **extract** and **send data through**.

### Reader

Let’s start with the Reader class. The Reader class should have a single method – readAllLines(), which should accept an **InputStream**. The method should instantiate a **BufferedReader**, and it should read all characters, from the **InputStream**. **Important**! **All characters**!

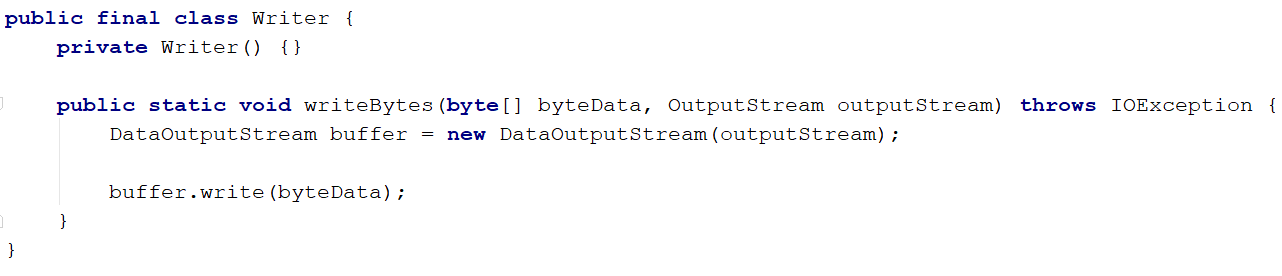
The class should be final and the **constructor** should be private! The readAllLines() method should be **static**, since its an **utility class**.



This is a better way of reading request, than the previous one we saw. This one actually works. 😉

### Writer

The other one is the Writer class. The Writer class should also be final and its constructor private. It should hold a single static method – writeBytes(). The method should accept a byte[] array, and an OutputStream, and should print the byteData, by instantiating a DataOutputStream.



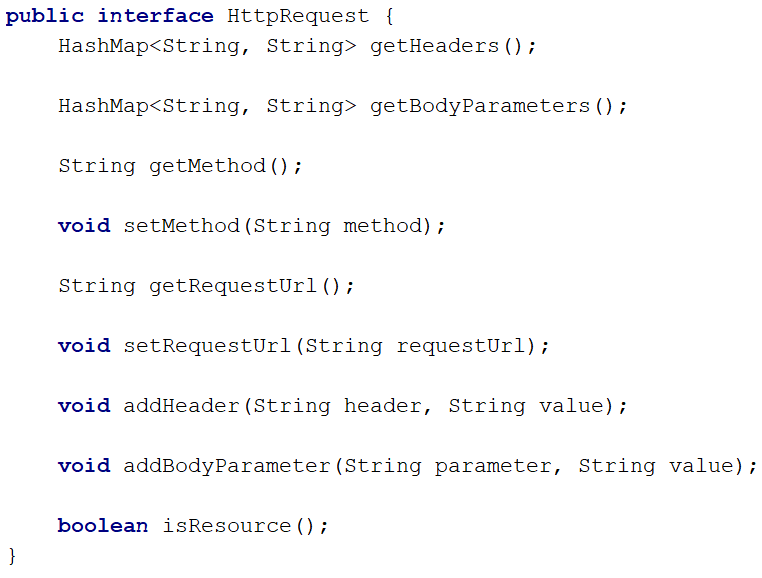
And we are done with this package! And with the examples. From now on, its up to you to build the other things. You’ll be given instructions though.

## HTTP Package

The http package is probably the first one you should start with. You should implement **2 classes** here, one for the HttpRequest and one for the HttpResponse. We are no longer going to keep these 2 things as flying strings. They will be objects. Here are their **interfaces** and their explanations:

### HttpRequest

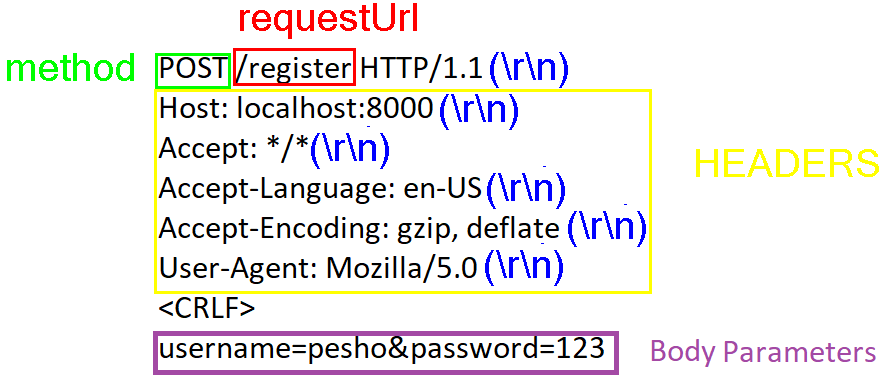
The HttpRequest:



Create a class which implements this interface, and write the logic behind the methods. The class should have a method, a requestUrl, a collection of headers and a collection of bodyParameters.

The class should be instantiated with a **single string**, which is the whole **content** of the **request**, passed all the way from the ConnectionHandler.

Remember, a request looks like this:

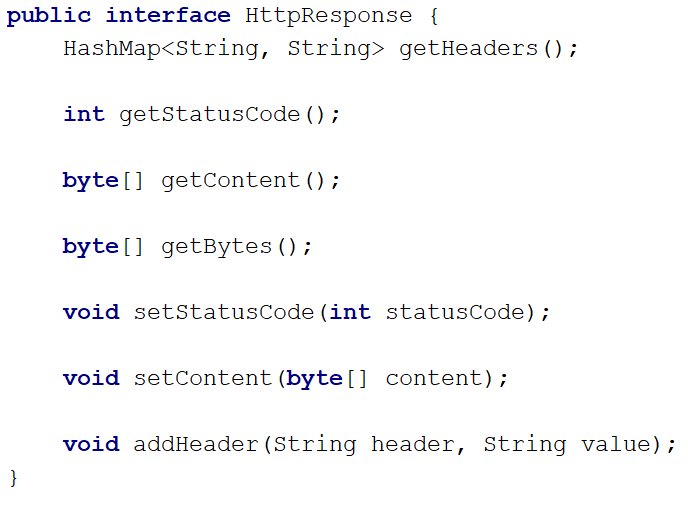


You should receive the **string** in the **constructor** of theRequestclass, exactly as shown above, with **every line**, **delimitered** by “\r\n”. **Everything** should be **parsed** and **formatted** **INSIDE** the class.

The isResource() method should **check** if the requestedUrl is a **resource** and **not an actual route**, and should return a **boolean result**.

### HttpResponse

And the HttpResponse:



Same as the HttpRequest above, you should implement this class, so that it **corresponds** to the **behaviour** defined by the **interface**.

The getBytes() method should **return** the **whole response** (**ResponseLine** + **Headers** + **Content**) as byte array.

## WebConstants class

This class should **hold all shared constants**, which will be used overall by the whole **Server**. Extract all magical values into this class, and by doing that you will perform the **most miniature refactoring** of the code you can.

## Resources Folder

This folder holds 2 subfolders – assets and pages. Download **several images**, **css files**, **text files**, etc... and put them in the assets folder. That’s where **all static** **files** reside.

Create some **.html** files, which use some of the assets. Put those **.html** files in the pages folder.

## RequestHandler

The time has come for the final class. Implement the RequestHandler class.

The RequestHandler class should have a **single public method** – handleRequest(), which **accepts** a **single string** – the requestContent. The RequestHandler should instantiate an HttpRequest with **that string**.

The RequestHandler should hold the **whole logic** (for now, until we come up with a **wiser architecture**). In other words, hit all your logic – **resource loading**, **page loading** etc... in this **class**.

Despite the fact that we are making a **God Object**, by design, try **to keep code quality** to the **maximum**. If you can, **divide** the **logic** into **different** **handlers**.

## Full Implementation

Implement your **Server** so that it works perfectly. Try **hosting** your **web pages**.

The server should **host .html files** from the pages folder, to specific **routes**.

“localhost:8000/index” – should receive index.html as Response.

The server should also **host resources** from the assets folder.

“localhost:8000/server.png” – should receive server.png as Response.

## This is not even my Final Form!

This is, by no means, the final server which you will use. This is a very simple http server, which processes HTTP Requests, and returns HTTP Responses, which may contain files. Its purpose is to exercise your skills in handling HTTP connections between Client and Server. There are a lot of things that can be refactored, changed, fixed... And they will be!

Try to identify the bugs, and if you can, fix them. Most of the times, locating the bugs is harder than fixing them, because, after all, at web level the architecture gets quite complex. A very little seemingly insignificant bug, may prove to be quite the challenge.