T6 - Javascript Frameworks

T-JSF-600

Bootstrap

Node JS & Express JS

1.0

STEP 0

First of all, check this out.

As you will notice, there is plenty to read and learn, it can be hard at times.
Getting a good understanding of Node will be a good thing for your career as a developer.
Node is a very powerful tool.

For this project, synchronous functions are forbidden.
Node strength lies in asynchronous functions. Let's do things in the Node way!

In Node, callbacks can quickly make your code difficult to read and to review.
This is why it is relevant to organize your code into modules in order to encapsulate functionalities.

STEP 1

Using `HTTP module` (asynchronous), create a **server** module that starts a server on a given port (passed as parameter to a *start* method).
Each time a client connects, the server must write

```
Client connected
```

on the standard output and must, of course, write

```
Client disconnected
```

upon disconnection.
Since you are polite, your server should greet each new client by sending him the following content:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF8"/>
        <title>Welcome</title>
    </head>
    <body>
        <p>Greetings $NAME!</p>
    </body>
</html>
```

This html code must be found in a **index.html** file, and retriveved using Node's `File System` module.

The *$NAME* parameter is to be obtained through the URL (GET).
For example, a call to page *localhost:port?name=Martin* should display "Greetings Martin!" to the user.
If there is no name parameter obtained, the greeting should be "Greetings whoever you are!"

`URL` module.

Think about how you are going to replace your name variable in the HTML page that you will get with File System. It is simply a matter of text manipulation.

Each time a client sends a "ping" to the server, the server must answer "pong".
If a different message is sent, the server must answer, "Sorry, I don't understand. :(".

STEP 2

There is an ugly way to add more pages using multiples if and else.
Not for you… (other developers will thank you for your cleanliness; you will even thank yourself.)
Provide a way for your server to differentiate between the different URLs it receives requests for, and generate custom responses for each of these specific URLs.

To achieve this, you are going to implement a router.

Your router must check if the URL is found in the indexes of an associative array, and if the corresponding value is indeed a function.
If so, it will be called with a request object and a response object.
If no corresponding index is to be found, your router will reply with an error 404 in plain text simply stating:

```
404 error: Page not found.
```

To implement the custom responses, create a **pages** module that contains methods to display specific pages by receiving as arguments a request object and a response object.

Of course, in each case the header sent by your server must have a value of 200 and a Content-Type indicated as `text/html`.

Don't forget to modify the *start* method of your server so that it takes as parameters a port, your route method and an associative array handle.
Upon a new connection, your server will call route with the right arguments.

The associative array must contain paths as key and your pages methods as value. This must be instrumental in hadling routing neatly. Obviously this array must be declared only in one place.

STEP 3

Just a tiny bit more complexity: handling POST requests.
First create a little form named **form.html** sending to */index.html* with an *input* field named **name** and a **Submit** button.
Modify whatever should be to display your form on */form.html*.

If you hit the submit button, you are redirected to the index page. This is because the content of your form is not handled yet…

Modify the index method of your pages module so that when it receives a POST request with a field *name*, your index page uses it instead of GET in order to greet this visitor.

Keep in mind that your index method must be able to handle both behaviors.

STEP 4

Create a client module containing a method called **connect**, taking a host and a port as parameter.
As soon as the connect method is called, your client must connect on the given address and port, and send a message containing "ping" before disconnecting.
Any answers from the server will be displayed on standard output followed by a new line.

STEP 5

Before going further on, familiarize yourself with Express and its API.

As you may have seen with the Node project, configuring a basic server in Node JS requires a fair amount of code and is not easy to use and maintain.
Moreover, it's quite easy to do it in the wrong way.

Here comes Express to make things a lot easier.

STEP 6

First, use Express functionalities to implement a **start** method in an *app* module.
This method must take a port as parameter and launch an Express application listening on that port.

Use Express methods in a way that:

- **index.html** file must be accessible from paths */* and */index*
  This page must display `Greetings Traveler!` in plain text.
  1
- **image.html** file must be accessible from */image* path.
- **form.html** file must be accessible from */form* path.
- **form.html** file must be accessible from */form* path.
- **student.ejs** file must be accessible from */student/X* path where *X* is a number.
  This file must generate an HTML page, according to the following example.
  For instance, a call to *localhost:port/student/5?name=Martin* should generate this code:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Student</title>
    </head>
    <body>
        <p>Greetings, Martin, student number 5!</p>
    </body>
</html>
```

Do you know what is ejs?

If a user tries to access a page that does not exist, return a 404 error with an adequat message in plain text.

No need to improvise something for non-existing files.
Express already has an easy to use functionality for this.

STEP 7

Using **cookie-parser**, add a new path to you application: **/memory** that renders a page displaying last access data to page */student/X*.
It must comply with the following behaviour:

- an access to */student/999?name=rico* and then */memory* should display `rico, student number 999 was here.` in plain text.
- an access to */student/999* then to */memory* should display `student number 999 was here.`
- a direct access to */memory* displays nothing (you don't have to remove the cookie).

In order to save these parameters even if the user closes his browser, you will save them in a cookie named **name** and a cookie named **number**.