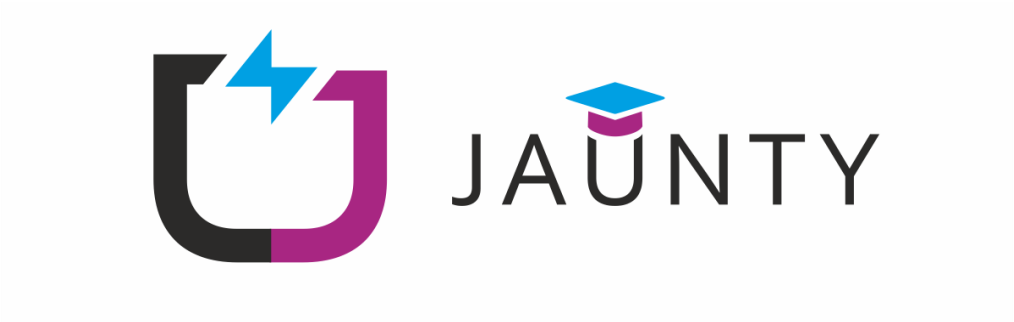
Developing REST APIs using NodeJS



Author: Public Power Corporation S.A

Co-funded by the
Erasmus+ Programme
of the European Union

Copyright

Consisting of

Coordinator:	Technical University of Sofia	Bulgaria
Partners:	University of Western Macedonia	Greece
	International Hellenic University	Greece
	Public Power Corporation S.A.	Greece
	University of Cyprus	Cyprus
	K3Y Ltd	Bulgaria
	Software Company EOOD	Bulgaria

Co-funded by the Erasmus+ Programme of the European Union

Table of Contents

1. Abbreviations

REST	REpresentational State Transfer
API	Application Programming Interface
I/O	Input / Output
CPU	Central Processing Unit
REPL	Read-Eval-Print-Loop
HTTP	HyperText Transport Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
JSON	JavaScript Object Notation
XML	Extensible Markup Language
JWT	JSON Web Tokens

2. Scope

The scope of this laboratory module is to introduce the NodeJS platform, along with useful libraries that exist in its ecosystem, and utilise it to develop a web-based REST API. In that sense, a full framework will be presented, which the students will use to create web applications. Along the way useful concepts like REST APIs, tokens, cookies, etc. will be explored.

2.1 Specific outcomes

Upon completion of this lab, individuals will be able to:

- Write a NodeJS project.
- Create REST APIs and test them with Postman.
- Understand how to secure web APIs with JSON Web Tokens.

2.2 General description

In this laboratory, we will explore the NodeJS platform and an important part of its ecosystem. NodeJS is a JavaScript runtime environment. That means that NodeJS can run JavaScript code outside the Browser, by incorporating the V8 engine, developed by Google for Google Chrome and Chromium. Along with the runtime environment, NodeJS provides a rich ecosystem of libraries that simplify development and design [1].

NodeJS is built around some very important concepts that define the way applications are designed and developed:

- **Single-Threaded**: NodeJS uses a single-threaded model but at the same time is highly scalable. This is possible using Event mechanisms, which allow the applications to respond in a non-blocking way. This model enables the developer to develop efficient applications in cases of high I/O usage, high number of requests etc. However, it is not efficient in cases of CPU-intensive applications, as a heavy computation would block the single-thread and halt the progress of the whole application.
- **Asynchronous and Event Driven**: Complementary to the single-threaded nature is its asynchronous nature. That means that almost all NodeJS code is non-blocking, NodeJS commands run and move on to the next command without waiting for a response. Any response is handled in an Event-based system, where call-back functions emit events when they are finished.

The above features define a platform that is both scalable, very fast and highly efficient for specific application like web applications.

Finally, this lab also deals with security aspects of REST APIs, by demonstrating the JSON Web Tokens (JWT) technology. JWT is an open standard that leverages web technologies to provide authentication, authorisation, and integrity check for REST services and transmitted data. JWT is a self-contained data

structure (token), comprised of two JSON objects (header and payload) and a digital signature (in the form of header.payload.signature, encoded in base64), meaning that it contains all the necessary information to allow or deny any given requests to an API [2].

An example of JWT usage is depicted in Figure 1. In this example, Amazon Cognito is the identity provider, i.e., a service that undertakes only to authenticate users, and the Amazon API Gateway is the service that requires authentication for granting access. JWT allows the authentication functionality to be decoupled from the service requiring authentication. In particular, the client first contacts the Identity Provider to authenticate itself by providing a set of credentials (1). After successful authentication (2), the Identity Provider returns a JWT as a response (3). JWTs are self-contained structures, meaning that they can be processed "offline" by other apps/services (e.g., the Amazon API gateway in our example) to authenticate. "Offline" means that the JWT can be validated any time by the service with no need to contact the Identity Provider. Ultimately, this provides scalability since many web services could be added in the infrastructure with no need for each one to have an internal authentication database or to communicate with a central authentication point.

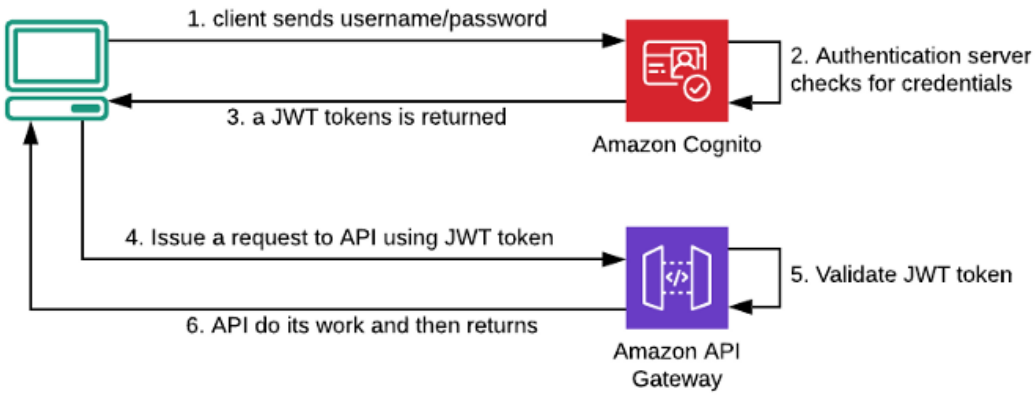


Figure 1: JWT Example [2]

2.3 Lab configuration

The following parameters are provided by the lab instructor:

Property	Value
Workstation IP address	
Workstation username	
Workstation password	
Access method for the Workstation	SSH / VNC

3. Exercise 1: Introduction

As we have discussed in the general description, NodeJS is a JavaScript runtime environment. That means that the code we will be writing in the lab course will be mainly JavaScript. Therefore, before diving into NodeJS, it would be helpful to see a few things about JavaScript along with some NodeJS specific concepts.

To implement the steps of this lab module, access the workstation by using the credentials provided in section 2.3.

In case you know and understand JavaScript you can skip the following step. However, you are encouraged to at least skim it.

Step 1: JavaScript Basics

The most basic way to access NodeJS runtime environment is through the console. Open the command line and run the following command. By this command you will access what is called the NodeJS REPL (Read-Eval-Print-Loop).

```
node
```

You will be presented with a '>' prompt and you are able to enter commands in series. NodeJS will run these commands and keep a state, which means that any variables you define will be kept if the REPL is running.

The REPL is a good way to test some basic commands, we will explore here. Alternatively, you can write these commands in a file and run them in the following way.

```
node [file.js]
```

Before moving on, try your intuition on the REPL. Try to do some basic math and transfer concepts from other languages you may know (like Python or C) in JavaScript. Can you spot any differences?

As in almost every language, in JavaScript we use variables to store data. You can define a variable by using the `let` keyword and a constant by using the `const` keyword. Like so:

```
1  let a = 42
2  const b = "hello"
```

Of course, the basic operations are supported.

```
1  let a = 42;
2  const b = 5;
```

```
3  let add = a + b;        //add equals 47
4  let sub = a - b;        //sub equals 37
5  let mul = a * b;        //mul equals 210
6  let div = a / b;        //div equals 8.4 (there is no native Euclidean
                             division in javascript)
7  let mod = a % b;        //mod equals 2
```

JavaScript is a weakly typed language, which means that types exist, but are implicit, so you don't have to define that `a` is integer, or the `div` is float. Additionally, JavaScript casts the variables implicitly. This can sometimes lead to strange outcomes like the ones below.

```
1  const c = "5";
2  let add = a + c;        //add  equals "425" (a is cast to string)
3  let sub = a - c;        //sub equals 37 (c is cast to integer)
```

You can also declare variables using the `var` keyword, or no keyword at all. There are differences, however. Among others, `var` variables have the scope of the function they are declared, `let` variables have the scope of the block they are declared and when you omit any keyword, they are considered global variables. In general, we would use the `let` keyword.

You can print messages by using the following command:

```
1  console.log("a is: " + a + " and b is: " + b)
2  console.log("a-b equals " + sub)
3  console.log("a+b equals " + (a+b))
```

Notice how in line 6 the plus sign is used in two different contexts. As an addition of two integers and as a concatenation of two strings.

Like almost all programming languages, JavaScript supports `for`, `if` and `while` statements. Additionally, it supports collection of items, like arrays. In JavaScript you can define arrays as follows:

```
1  let fruits = ["apple", "banana", "pear"]
2  let alt_fruits = new Array("apple", "banana", "pear")
```

To iterate over an array, you can use the usual for loop

```
1  for (i = 0; i < fruits.length; i++) {
2          console.log(fruits[i])
3  }
```

Or you can use the `forEach` method of an array:

```
1  fruits.forEach( x => {
```

```
2           console.log(x)}
3   )
```

The `forEach` method takes a callback function as an argument. Callback functions are functions that are passed as arguments and are called when a specific event happens. NodeJS, due to its single-threaded nature, relies heavily in callbacks.

In our case above we defined the callback function using what is called a **lambda function** or **arrow function**, which are functions that are defined and used in its place. Lambda functions are also anonymous functions and cannot be called in other areas.

We could have also defined a function and passed it as an argument:

```
1   function print(x) {
2           console.log(x)
3   }
4   fruits.forEach(print)
```

Alternatively, you could have named the lambda function:

```
1   print = ((x) => {
2           console.log(x)}
3   )
4   fruits.forEach(print)
```

The above ways have slight differences and would be used in different cases, however for our lab course we will consider them the same.

The two different ways of declaring a function shown above have similar differences to the `var` / `let` difference. The way with the `function` keyword, is called a **function declaration**, when we name a lambda function, we use a **function expression**. The difference is that function declarations are global in scope, while function expressions have the scope of their declaration.

Step 2: Asynchronous Programming

Now that we have seen JavaScript basics, we will move on to examine some very important concepts. As we have described before. NodeJS is single-threaded, however at the same time it provides asynchronous functionalities. It is very important to understand how this works. We will check this functionality with the following example.

```
1   const a = () => console.log("a");
2   const b = () => console.log("b");
3   const c = () => {
4           console.log("c");
5           a();
6           b();
```

```
}

c()
```

In the example above we have 2 functions (`a` and `b`), that print "`a`" and "`b`" respectively. Then we have another function that prints "`c`" and then calls the other two. We expect to see

```
c
a
b
```

Let's change the code a little and add a timeout:

```
1  const a = () => console.log("a");
2  const b = () => console.log("b");
3  const c = () => {
4          console.log("c");
5          setTimeout(a, 0);
6          b();
7  }
8
9  c()
```

The above code is almost identical but for line 5. Here we call the `setTimeout` function that takes as arguments a function and a time. After that time, it calls the function. As we have set the time to be zero (0), we would expect to see the same result as the above. However, that is not the case.

We would instead see this result:

```
c
b
a
```

That is because, the `setTimeout` function, would put the callback function on the end of the stack, that is after `b()`. That is the price we pay for the simplicity of single-threaded runtime. However, this simplicity provides us with an easier way to handle concurrency issues. Additionally, we have ways to circumvent the aforementioned problem by using **promises.** Promises are used to execute any asynchronous functions as soon as possible, instead of executing them at the end of the stack.

```
1  const a = () => console.log("a");
2  const b = () => console.log("b");
3  const c = () => {
4          console.log("c");
5          setTimeout(a, 0);
6          new Promise((resolve, reject) =>
7                  resolve("should be right after b, before a")
8          ).then(resolve => console.log(resolve))
9          b();
10 }
11
12 c()
```

Executing the above code should give you

```
c
b
"should be right after b, before a"
a
```

Let's see the above code. On line 6, we define a new Promise. Promises have three states. **Pending state** is the state the promise starts into, the calling function continues executing and when the promise resolves, it will return to the calling function the appropriate data. When a promise ends, it will be either in a **resolved state** or in a **rejected state**. Which would correspond to the execution completing successfully or failing.

That is why on line 6, we have two arguments to our lambda function. The lambda function is the callback we pass to the promise constructor, and it defines what we want to execute in this promise. The callback function should return a resolve and/or a reject statement like so:

```
1  const prom = new Promise((resolve, reject) => {
2          […]
3          if (success) {
4                  resolve("The job was done successfully")
5          } else {
6                  reject("The job failed")
7          }
8  })
```

Then you can match whether the promise succeeded or failed by using the `then()` and `catch`() methods:

```
1  prom
2          .then(result => console.log(result))
3          .catch(err => console.error(err))
```

By using promises, we can create programs that while single-threaded, are asynchronous. As such, promises are an important part of the NodeJS that allow its programs to scale and extend their functionality.

When we want a function to return a promise, we can define it as an `async` function, and then it will implicitly return a promise, even if we didn't define one in its body:

```
1  const testFunction = async () => {
2          return "success"
3  }
4
5  testFunction().then(x => console.log(x)) //this should print "success"
```

Additionally, we can use the `wait` command to block some function until a promise, returns. Like so:

```
1  const helper = () => return ""should be right after c, before b"
2
3  const a = () => console.log("a");
4  const b = () => console.log("b");
5  const c = async () => {
6          console.log("c");
```

```
7           setTimeout(a, 0);
8           await helper().then(x => console.log());
9           b();
10  }
11
12  c()
```

The output should be the following:

```
c
"should be right after c, before b"
b
a
```

Promises and Async functions could be tricky to wrap your mind around. Study the examples above and try some tests of your own to grasp the concept. Even if we will not use them right away, they are an important part of how we structure our applications when we use NodeJS, and most libraries use them.

Another way to work in an asynchronous way is to use Events and the EventEmitter like below:

```
1   const EventEmitter = require("events")
2   const eventEmitter = new EventEmitter()
3
4   eventEmitter.on("fire", args => {
5           console.log("fired with argument " + args);
6   }
7
8   eventEmitter.emit("fire", 25)
```

In line 4 we define an event callback. We are defining that when the `"fire"` event is emitted, the callback function will be run. Then in line 8, we emit the `"fire"` event along with the argument that is needed in line 4.

Experiment with events and promises before moving on to the next step.

Step 3: Modules and NPM

In the above code, in line 1 we used the `require()` statement. This statement imports what are called modules. You can think of modules like libraries and the `require()` statement like C/C++ `include` directive or the python `import` statement.

In general, when we want to import some functionality from a library, we would use a `require` statement to include the module, like in line 1. Then to use the functionalities of the module, we would have to create an object of that type, like in line 2. From then on, we can use the module's functionalities through that object.

Before moving on to how you can find new modules and what you can use to organise your work, we will see how you can export your own modules.

First, create a file named `adder.js` containing the following function:

```
1  function add10 (x) {
2          return x + 10
3  }
4
5  exports.add10 = add10
```

Here, we declare a function, `add10`, which takes a number and adds 10 to it. Then in line 5, we use the `exports` object, which is provided out of the box by NodeJS.

Now we can use the `require()` statement in another file to import and use the `add10` function there, like so. (Notice that we can omit the .js filename extension.):

```
1  const adder = require("./adder")
2  console.log(adder.add10(5))            //this should print 15
```

The NodeJS platform has an extensive list of available modules for various needs. To be able to use and access all these modules in an organised way, NodeJS comes with the npm (Node package manager). We will use npm to create NodeJS projects and to manage the various dependencies on packages.

Create an empty directory named `test`. Then run the following command to initialize a NodeJS project:

```
npm init
```

You will be presented with the message depicted in Figure 2, asking you for your input. For now, we will leave these to the default values. The fields are self-explanatory.

After that, npm would have constructed the package.json, which describes the project. Open this file with a text editor and you should see the json that was produced above. Note that it does not have any dependency fields.

As we have said before, npm is also used to manage the module dependencies. This works in two complementary ways. Managing the installation of individual modules and porting projects to other environments.

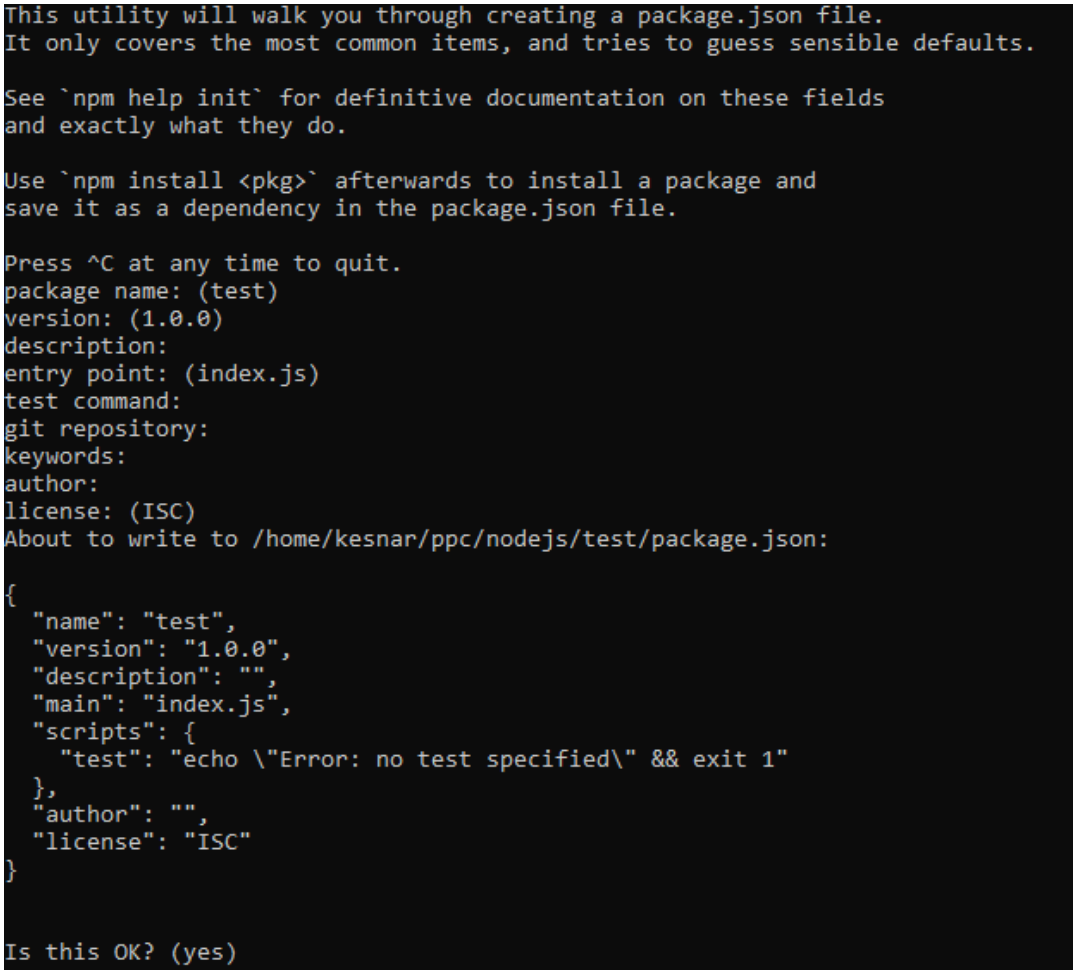


Figure 2: Dialogue prompted by npm init

For example, Express is a NodeJS web application module that provides minimal but flexible features to develop web applications. To install it in our project we must run:

```
npm install express
```

Then, npm will download express, along with any other dependencies and install them to a folder under our current project named `node_modules`. Additionally it will create a file named `package-lock.json`, which describes the dependency tree npm generated. This helps in subsequent module installations, not to install redundant modules.

Now to import the express module you will have just to insert this into your code:

```
1  const express = require("express")
2  const app = express()
```

By installing some modules, npm has also edited the original package.json file. Open it now on a text editor and note that there is an extra field called dependencies. Now this should contain only one subfield, which is the express module we have installed.

Create another directory on the same level as the test directory before, copy the package.json file in it and run:

```
npm install
```

Now npm will read the package.json file and download any dependencies it finds there. That means that a NodeJS project requires only the source code we have written along with the package.json file to be ported in another environment. Of course, npm will have to run to download the dependencies.

4. Exercise 2: A Simple Web Server

Now that we have explored the basic concepts of JavaScript and NodeJS, we will go on to create a simple Web Server.

Step 1: Hello World!

Create a file named `server.js`, containing the following code:

```
1  var http = require('http');
2
3  http.createServer(function(req, res) {
4          res.writeHead(200, {"Content-Type": "text/html"});
5          res.write("Hello World!")
6          return res.end();
7  }).listen(8080);
```

Before reading the explanation of the code above, try to read it and understand what it does. Find patterns that we have already met in the introductory exercises. What are the parameters of the `http.createServer()` method?

The code above describes the most basic http server you can have. In line 1, we import the http module, which is a native NodeJS module that allows us to manipulate http requests. Then in line 3 we call the `createServer()` method, which takes as a parameter an anonymous function and we invoke the `listen()` method (line 6), which describes the port number the server is listening to. The anonymous function that is used as a parameter to the `createServer()` method is actually a callback function. Every time a request is sent to the server in port 8080, the callback function is called. This function has two parameters, `req` that describes the request and `res` that correspond to the response of the server to that particular request. In our case, we write the 200 code in the head, signifying that the response is ok and “Hello World!” in the body.

Now save the file and run it with NodeJS, by issuing the following command:

```
node server.js
```

To check the server functionality, you can visit **http://***WORKSTATION_IP***:8080/** in your browser. You should see the response depicted in Figure 3.

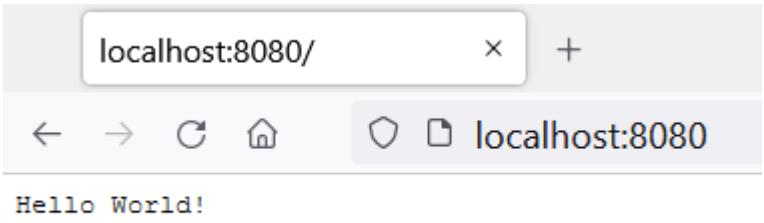


Figure 3: Preview of the server NodeJS app

Alternatively, you can use the `curl` command from your workstation to get a response from the server.

```
curl http:// ***WORKSTATION_IP***:8080
```

Step 2: HTTP client

We could also try to build our own client to send http requests to our server. Create a new file named `client.js`, containing the following code:

```
1  const http = require('http');
2
3  let options = {
4          host: "localhost",
5          port: "8080"
6  }
7
8  let callback = function(response) {
9          var body = "";
10         response.on("data", function (data) {
11                 body += data;
12         });
13
14         response.on("end", function() {
15                 console.log(body);
16         });
17 }
18
19 let req = http.request(options, callback);
20 req.end();
```

Make sure that our server is running and, in another terminal, run the `client.js`. You should receive the response to the terminal.

Let us see now how that code works. First in line 1, we import the `http` module. Then we declare an object `options` with two properties, `host`, and `port`. Additionally, we declare a function that we will be using as callback. That function takes a response and checks whether it is a `data` or an `end` event. If it is `data`, it concatenates the received data to a string. If it is `end`, it prints all the data received to the console. Finally, on line 19, we make the request, providing the `options` object and the `callback` function.

We saw a different approach on how to define a callback function in this case. Instead of defining it in place, we defined a function beforehand and provided it to a caller. Both ways are valid and could serve different purposes. In general, if you are going to reuse a callback function it is better to name it and define it separately from the caller.

Notice however that in that code above the port and the host are hardcoded in our program. If we want to test another URL or another port, we must change the code itself. It would be easier if we could take that input from the command line.

For that we will use the `process.argv` array. Create a file named `test.js`, containing the following:

```
1  console.log(process.argv)
```

If you run this program, you will receive as output something like this:

```
1  [ '/usr/bin/node', '/[…]/test.js' ]
```

Try running the following command

```
node test.js one two three
```

What is the output? What do you think that the `process.argv` array stores?

As you can see the `process.argv` array stores the command line arguments received by the shell when you call the NodeJS program. As such, it stores also additional arguments that you provide when you call your program. We will use that to receive user input. Change the original `client.js` as such.

```
1  const http = require('http');
2
3  var args = process.argv.slice(2);
4
5  let options = {
6          host: args[0],
7          port: args[1]
8  }
   […]
9  let req = http.request(options, callback);
10 req.end();
```

In line 3 we use the `slice()` method, to keep only the values after the first two. That is because the first two are the NodeJS program and our source code.

Now when you call your `client.js` program you must provide the `host` and `port` arguments, like so:

```
node ./client.js ***WORKSTATION_IP*** 8080
```

You should be able to receive the reply from the server.

By introducing user input, we have introduced the possibility of the user not giving the right kind of input.

What happens if you give less arguments to your program? If you give wrong kind of arguments, e.g., port is not a number?

For that reason, we must include some checks in the beginning of our program.

```
1  const http = require('http');
2
3  var args = process.argv.slice(2);
4  if (isNaN(parseInt(args[1]))) {
5          console.log("Port must be a number!")
6          process.exit()
7  } else if (!(args[1] >= 0) && (args[1] < 65536))  {
8          console.log("Please enter a valid port number, between 0 and
   65536!")
           process.exit()
9  }
10 […]
```

In line 4 we check if the second argument (port) can be parsed as a number and in line 7 we check if it is in the range of [0, 65536], which is the valid range for the port numbers.

While we are at it, we could also put an error checker for the `http.request()`.

```
1  […]
2  let req = http.request(options, callback);
3
4  req.on("error", function(e) {
5          console.log("Cannot connect to the server!");
6  });
7  req.end();
```

Step 2: File System

In the example in Step 1, we created a server that replied with a “Hello World!” string to any request. In most cases however, we would want to respond with some html code to display a proper page to the user.

To do that we first have to create an html file. Let’s name it test.html:

```
1  <html>
2  <body>
3  <h1>Sample Header</h1>
4  <p>Sample Paragraph.</p>
5  </body>
6  </html>
```

We will not delve into HTML or front-end JavaScript in this lab, just use the examples you are given in such cases.

Next, we want to change our original server code, so that every time we respond to a request by sending the test.html file.

```
1  var http = require('http');
2  var fs = require('fs');
3
4  http.createServer(function(req, res) {
5          fs.readFile("test.html", function(err, data) {
6                  res.writeHead(200, {"Content-Type": "text/html"});
7                  res.write(data)
8                  return res.end();
9          });
10 }).listen(8080);
```

By looking at the code, we can see that in line 2, we import the `fs` (filesystem) module and we use it in line 5 to read the test.html file. Note that the server must be in the same directory as the test.html file, as we have given a relative file path to the `readFile()` function.

As you can see the `readFile()` method takes a callback function, which is used to manipulate the read data. Again, this is a recurring and very important concept in JavaScript. Make sure that you have grasped that concept!

Run the new server and access it through the browser and the client we have built in step 2. Are there any differences? In most browsers you can right click on a page to view the page source. Is there any difference now?

The `fs` module is a very useful module that allows us to read, create, manipulate, delete etc. files in the filesystem.

For example we could use the `writeFile()` method to create the test.html on the beginning of our server, instead of having it be a static file. Our new server will take two arguments when run, which will correspond to custom Header and Paragraph instead of “Sample Header” and “Sample Paragraph”.

```
1  var http = require('http');
2  var fs = require('fs');
```

```
3
4  var args = process.argv.slice(2);
5
6  fs.writeFile("test.html", "<html><body><h1>"+ args[0] +"</h1><p>" +
                                args[1] + "</p></body></html>", function
   (err) {
7          if (err) throw err;
8          console.log("Created test.html");
9  });
10
11 http.createServer(function(req, res) {
12         fs.readFile("test.html", function(err, data) {
13                 res.writeHead(200, {"Content-Type": "text/html"});
14                 res.write(data)
15                 return res.end();
16         });
17 }).listen(8080);
```

As you see we have used the `process.argv` array in order to capture the user input. Of course, the above code is highly inefficient and is given here as an example for experimentation. In latter steps, we will see how to handle such things better.

Find what other methods the fs module provides.

What does line 6 do? Should we have done the same for `fs.readFile()`. If yes, try to implement it.

Experiment with what you have learned until now.

Step 3: File Upload Server

Until now, we were mostly concerned with experiments and showcases, not with useful applications. In this step, we will use what we have learned so far to create a simple file upload server. Where the user will be able to access our server and select a file to upload. The server will receive the file and store it.

For this example, we will need to use an external module, called formidable, which allows us to handle html forms in an easy and efficient manner. To begin with, create a project with **npm** and run the following to install the formidable module.

```
npm install formidable
```

Then create a file, named fileserver.js with the following contents. Also create a directory named files.

```
1  var http = require('http');
2  var fs = require('fs');
3  var formidable = require('formidable');
4
5  http.createServer(function (req, res) {
6          if (req.url == "/fileupload") {
```

```
7                   var form = new formidable.IncomingForm();
8                   form.uploadDir = "./files/";
9                   form.parse(req, function (err, felds, files) {
10                          var oldpath = files.filetoupload.path;
11                          var newpath = "./files/" +
files.filetoupload.name;
12                          fs.rename(oldpath, newpath, function (err) {
13                                  if (err) throw err;
14                                  res.write("File uploaded and moved!")
15                                  res.end();
16                          });
17                  });
18          } else {
19                  res.writeHead(200, {'Content-Type' : 'text/html'});
20                  res.write('<form action="fileupload" method="post"
enctype="multipart/form-data">');
21                  res.write('<input type="file"
name="filetoupload"><br>');
22                  res.write('<input type="submit">');
23                  res.write('</form>');
24                  return res.end();
25          }
26  }).listen(8080);
```

Before moving on to explaining the code, try it first. Visit **http://***WORKSTATION_IP***:8080** and try uploading a local file to the server. If everything goes right a message should inform you that the file was uploaded, and you should find the file in the file's directory of your project.

Regarding the code itself, there are a couple of things going on. First, look at lines 18-24. Here we describe the basic html that contains the form and should appear when a user accesses the server. It may look strange, but it is just another way to send the html code, instead of having it stored in a file and have our program read the file, we do that in place. The html code itself does not concern us for the time being, just check line 20 that contains "`form action="fileupload"`". This is the page that our server will redirect the user after the upload has happened.

With that part explained, look at the main body of our program, lines 6-17. In line 6, we see that there is a check whether we the request is for `/fileupload`, which means that we have arrived here by the route of the form. Then, at lines 7-9, we check the formidable object that contains the form contents and we call a callback function (lines 8-17), which takes the uploaded file and moves it to the local ./files directory. Check that the copy is implemented using the `fs.rename()` method.

Spend some time reading the above code. Try making changes and uploading various files to see what happens.

Try to visit **http://***WORKSTATION_IP***/fileupload** without uploading a file. What happens? Why? Can you try to fix it?

We have seen that the form redirects the user to the **Error! Hyperlink reference not valid.** . If you try to access that specific URL without uploading a file, the server will crash, and you will get no response. That is because, our code specifies that the `/fileupload` URL has some specific function tied to it that require a form to work. When we access it not through the form, we do not have the necessary information that is needed. We could of course change our code to check for that issue but to really fix this situation we should delve a little bit further into the HTTP protocol, its methods and the way we structure services based on it. These services usually come in the form of REST APIs, which would be the subject of the next exercise.

5. Exercise 3: REST API

A REST API, which stands for REpresentational State Transfer, is an architecture for services that is used in the World Wide Web and uses the HTTP protocol [3]. In the REST architecture, there are resources, which can be accessed and manipulated by use of some standard HTTP methods. The four basic HTTP methods are:

- **GET**: used to access a resource in a read-only mode
- **POST**: used to update or create a resource
- **PUT**: used to create a new resource
- **DELETE**: used to delete a resource

A server that uses the REST architecture is said to provide RESTful web services, or to provide a REST API. That means that the web service can respond to requests made to specific URIs (Uniform Resource Identifier) and the response will contain a representation for the specific resource, usually in plain text, JSON, XML, etc.

Step 1: GET methods

From the above, we understand that our web server must be able to handle specific operations on specific addresses. To do this efficiently we are going to use the `express` module.

As we have done before, create a new project with npm and install the `express` module. The `express` module will allow us to write servers, web applications and REST APIs in a much easier way than using the native libraries. To begin with, we will start with a basic server to see how `express` works.

Create a file named server.js containing the following:

```
1  const express = require('express');
2  const app = express();
3
4  app.listen(8000, function() {
5          console.log("server is running");
7  })
```

In the above code, we "import" the `express` module and initiate our server, which listens to port 8000 but does nothing more than printing a message to the console.

Try accessing the server through your web browser. What do you see as a response? Why is that?

The code above does nothing useful for the time being. We will have to implement some routes. Routes are pieces of code that describe specific functions tied to specific URLs. Usually in a big project, we would have written all our routes in a separate file to be used by our main server, however here we will keep it to one file for simplicity, as we do not have so many routes. Change server.js as bellow:

```
1  const express = require('express');
2  const app = express();
3
4  const mockSensors=[
5          {id: 'TempSensor'},
7          {id: 'HumSensor'}
8  ]
9
10 app.get('/sensors', function(req, res) {
11         res.json({
12                 success: true,
13                 message: 'succesfully got sensors.',
14                 sensors: mockSensors
15         })
16 })
17
18 app.listen(8000, function() {
19         console.log("server is running");
20 })
```

Here we first declare an array in line 4 with two sensors in JSON format. Then we declare a route in line 10, which would run when someone accesses the `/sensors` URI by a GET method. Then we will reply with a message in JSON format, with 3 values, including the sensors array.

You can also use a dynamic URL by using the colon (`:`) identifier, like in the below router:

```
1  […]
2  app.get('/sensors/:id',function(req,res){
3          id = req.params.id;
4          if (mockSensors.some(x => x.id == id)) {
5                  res.json({
7                          success: true,
8                          message: 'got a sensor!',
9                          id: id
10                 })
11         } else {
12                 res.json({
13                         success: false,
14                         message: 'no such sensor!',
```

```
15                              id: ""
16                              place: ""
17                      })
18              }
19      })
20      […]
```

Here in line 2, we define a URL that includes a colon. That binds the value that follows to the `id` variable. Based on that value we check whether our `mockSensors` array includes that sensor id. If it does, we respond by success and the name of the sensor, otherwise we respond that no such sensor exists.

Check line 4, what does the `some()` method do? How does it work? Can you think of other ways to do the same thing?

Check line 9. Is `id` on the left side the same as the one on the right side? What is the semantics of each one?

Step 2: POST methods

In the same way, we will write routers for other methods, like POST. For this example, we will change our `mockSensors` array to include more information for each sensor like so:

```
1   […]
2   const mockSensors=[
3           {id: 'TempSensor', place: 'PPC'},
4           {id: 'HumSensor', place: 'UOWM'}
5   ]
6   […]
```

However, while GET methods use only the URL itself, POST methods also use the Body of the request. To parse the body, you must add this line in the beginning:

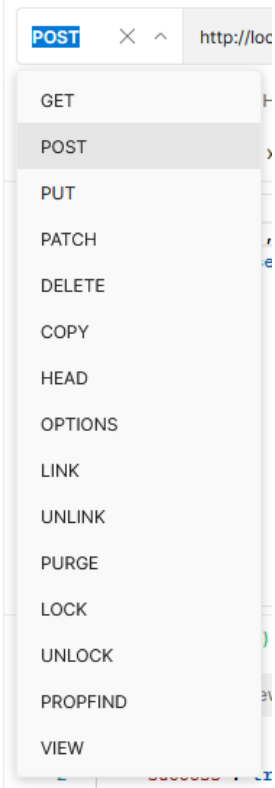
```
1   app.use(express.json());
```

Now insert the following post router:

```
1   […]
2   app.post('/insert',function(req,res){
3           const id=req.body.id;
4           const place=req.body.place;
5
7           if (mockSensors.some(x => x.id == id)) {
8                   res.json({
9                           success: false,
10                          message: 'sensor already indexed!',
11                          sensor: {id: id, place: place}
12                  })
13          } else {
14                  mockSensors.push({"id": id, "place": place});
```

```
15                      res.json({
16                              success: true,
17                              message: 'succesfully inserted sensor!',
18                              sensor: {id: id, place: place}
19                      })
20              }
21      })
22      […]
```

Try accessing the server through your web browser. What do you see as a response? Why is that? Does the browser send POST request?



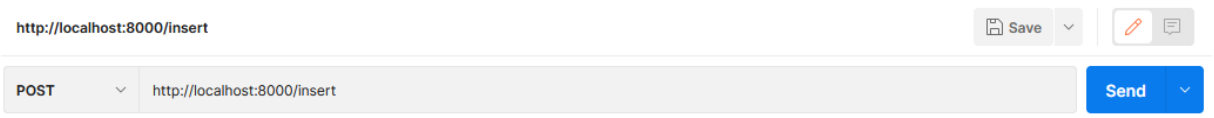
To test the POST API, we must find a way to send POST requests. Unfortunately, our browser sends only GET requests. For this, we will use Postman.

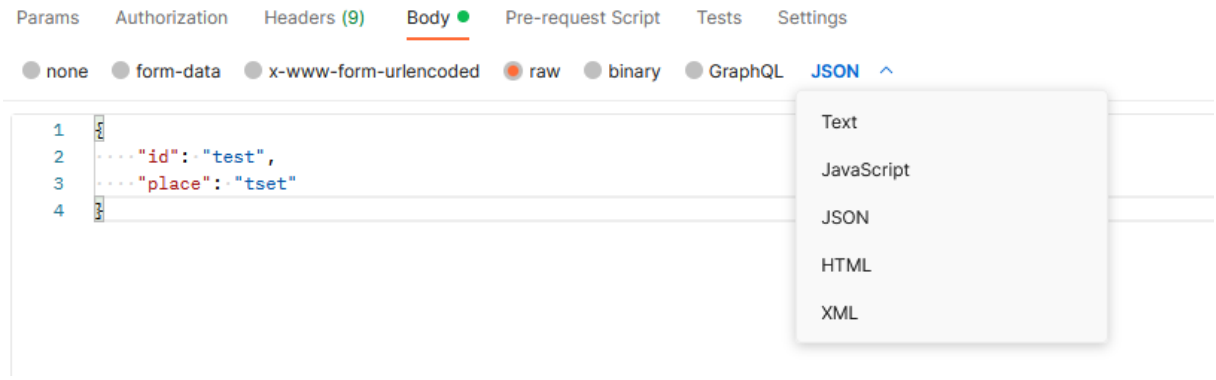
Postman allows us to send a variety of request by selecting from the dropdown menu as shown below.

Open the Postman application on your workstation.

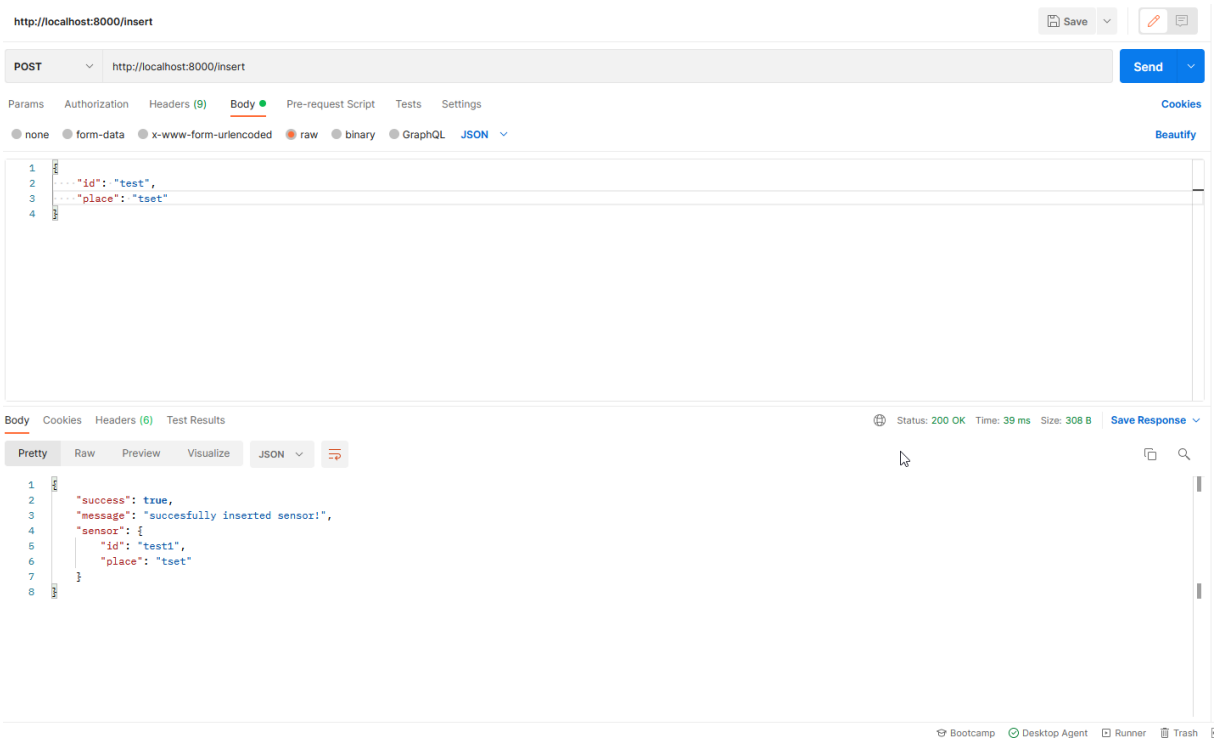
For now, select POST. Then, we have to insert the URL we want to send the request to.

As we have said before, POST uses the body of the request. As such, we have to select the Body tab. There select the raw format and make sure that JSON is selected from the dropdown menu. Insert the sensor we want to add to our array, as shown below. Make sure to not forget the quotes around `id` and `place`.

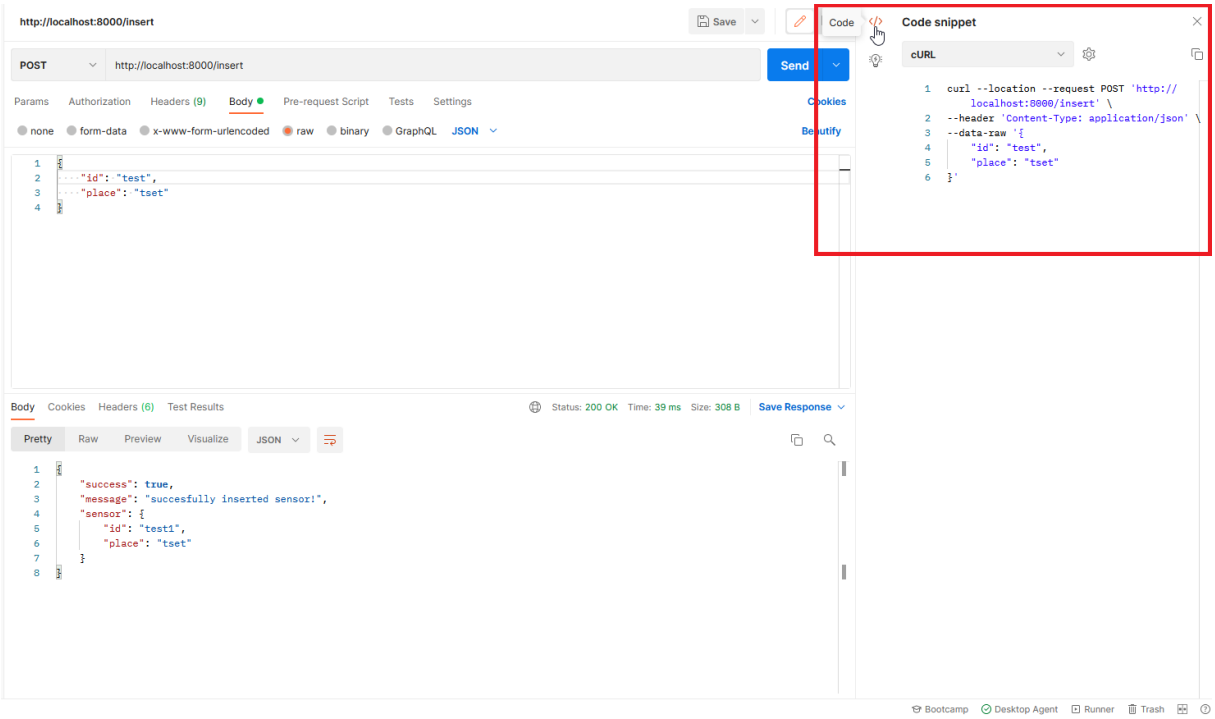




Then by clicking on Send you should receive the reply on the bottom pane.



You can also use the `curl` tool if you are familiar with cli tools. Fortunately, Postman can show us the `curl` equivalent command.



Before moving forward experiment with the API we have created so far. Insert some more sensors through postman and check that they are really in the array by visiting `/sensors`. Try to alter the `/sensors/:id`, so that it also replies with the place of the appropriate sensor. Think of other ways you can extend and enrich your REST API.

6. Exercise 4: Cookies and JSON Web Tokens

In the above example we created a basic API for an index of sensors. With it, one could import new sensors, get a list of every sensor indexed and check specific ones. For all this, the user had to just know the specific addresses and the format of the data, no authentication was implemented.

In this exercise, we will see a type of authentication using JSON Web Tokens (JWT) and storing them as cookies. Cookies are small blocks of data created by a server and stored in the user's web browser. JWT is a way to create data that is encrypted and signed to assert some claim. For example, we can create a JSON web token that will claim that a user has logged in to our server and store this information as an encrypted cookie.

Step 1: Storing cookies

To begin with, create a new npm project and install the following libraries:

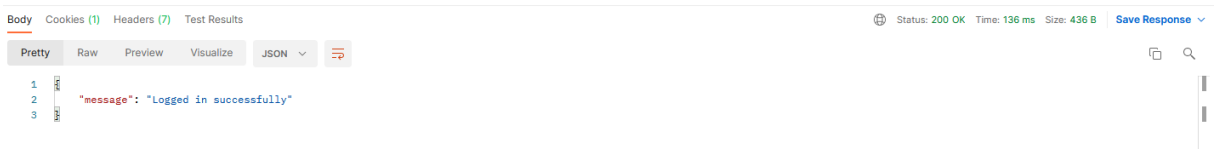
```
1  npm init -y
2  npm install express cookie-parser jsonwebtoken
```

What does the `-y` flag do?

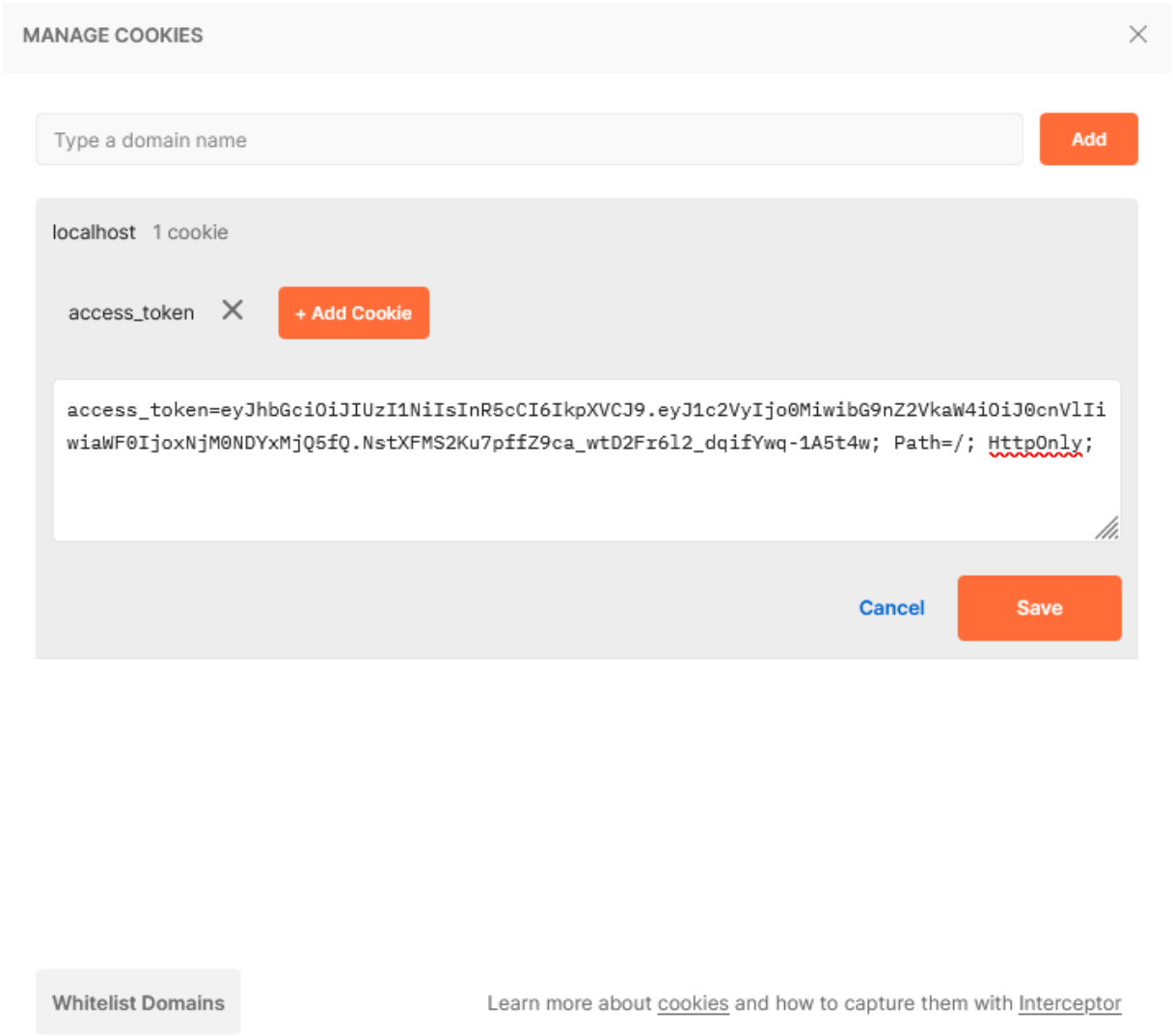
Now we will create a basic server with one route in `/login`, which will create a JSON web token and store it as a cookie:

```
1  const express = require("express");
2  const cookieParser = require("cookie-parser");
3  const jwt = require("jsonwebtoken");
4
5  const app = express();
6
7  app.use(cookieParser());
8
9  app.get("/login", (req, res) => {
10         const token = jwt.sign({ user: 42, loggedin: "true" },
   "PASSPHRASE");
11         return res
12                 .cookie("access_token", token, {
13                         httpOnly: true,
14                 })
15                 .status(200)
16                 .json({ message: "Logged in!" });
17 });
18
19 app.listen(8000, function() {
20         console.log("server is running");
21 })
```

Before explaining what the code does, run the server and send a `GET` request to the server with postman.



You should receive a message that you have logged in successfully. Now press the **cookies** button to see the stored cookies



As you can see, with the above request, a token was saved in our browser, which contains encrypted information about the log in.

This token is created on line 10 of our code. There we create a token with contents of `{ user: 42, loggedin: "true" }` and sign it with the string "`PASSPHRASE`". Normally, we would sign it with a private key, so that we can verify that the server has really encrypted this token. Then on line 11 we create the return object. In previous examples we only return a JSON message. Here we return 3 things. A json message on line 16, a status code in line 15 and a cookie in line 12-14. The cookie contains a header "`access_token`", the token itself and some parameters that define its behavior. In our case we set the `httpOnly` flag as true. This means that the cookie can only be accessed when an HTTP request is sent from our browser and not through JavaScript on the client side for example.

Find a list of the most common HTTP status codes.

What other parameters can we define for our cookies?

Step 2: Accessing cookies

Now that we have stored a cookie to our user's browser, we will use this to grant them access to specific information. We will create another route `/protected`, which will return data only if the user has an access token, for this we will create a helper function that takes care of this authentication.

```
1  […]
2  const authorization = (req, res, next) => {
3          const token = req.cookies.access_token;
4          if (!token) {
5                  return res.sendStatus(403);
6          }
7          try {
8                  const data = jwt.verify(token, "PASSPHRASE");
9                  req.user = data.user;
10                 req.loggedin = data.loggedin;
11                 return next();
12         } catch {
13                 return res.sendStatus(403);
14         }
15 };
16
17 app.get("/protected", authorization, (req, res) => {
18         return res.json({user: req.user, loggedin: req.loggedin });
19 });
20 […]
```

Now restart the server. Try to access the `/protected` route, before `/login`. Then access the `/login` and try again. The first time, you should see a Forbidden message. However, after you have visited `/login`, you should see the user number and the loggedin flag. This happens due to the authorization function. In line 17, we set the `/protected` route, but this time, we do not pass just a function but two! The first one is `authorization`. Then we pass a function that returns a JSON message as usual.

In line 2, we define the function parameters, here we have the usual `req` and `res` but also a `next` function. So, we can understand the anonymous function of line 17 is actually passed as the `next` parameter to the `authorization` function. In the `authorization` function, we first check that the token exists (line 3, 4). If it does not, we return status 403 error, if it exists, we try to verify it, by decrypting it, with the same passphrase as the one we have encrypted it. After we have encrypted it, we create a `req` object and pass it to the `next` function, whose body is in our case in line 18. In case we cannot decrypt it for any reason, we send a status 403 error.

Step 3: Deleting cookies

Now as a last step, we must implement a `/logout` route, which will delete the access token. This is easy:

```
1  […]
2  app.get("/logout", authorization, (req, res) => {
3          return res
4                  .clearCookie("access_token")
5                  .status(200)
6                  .json({ message: "Successfully logged out" });
7  });
8  […]
```

We again use the `authorization` function to make sure that the correct token is present and then we delete it (line 4) and return a message and a status code 200. Restart the server and try logging in and out while accessing the `/protected` route.

Using the above, we can securely store information on the client's side, such as logged in information, etc. JSON web tokens are an integral part of secure REST APIs.

As you can see, in the `/login` route, we do not really authenticate a user, we just store a cookie with a token in the client's browser. If we wanted to authenticate a user with a log in form, we would need to have stored a list of users and have a log in form, which we could parse with `formidable`. Using what we have seen in Exercises 2 and 3, as an exercise try to implement such a route.

7. Exam Questions

1. What protocol does a REST API utilise?
2. What is the difference between MQTT and REST APIs?
3. Which are the most common HTTP methods to interact with a REST API?
4. What is a call-back function?
5. What is the role of the Read Eval Print Loop (REPL) in NodeJS?
6. What are the benefits of JWT for authentication and authorisation?

References

[1] A. A. Prayogi, M. Niswar, Indrabayu and M. Rijal, “Design and Implementation of REST API for Academic Information System,” in *IOP Conference Series: Materials Science and Engineering*, 2020.

[2] M. Calandra, “Why do we need the JSON Web Token (JWT) in the modern web?,” Medium, 6 September 2019. [Online]. Available: https://medium.com/swlh/why-do-we-need-the-json-web-token-jwt-in-the-modern-web-8490a7284482. [Accessed 28 April 2022].

[3] Red Hat Inc., “What is a REST API?,” 8 August 2020. [Online]. Available: https://www.redhat.com/en/topics/api/what-is-a-rest-api. [Accessed 28 April 2022].

8. Contacts

Project Coordinator:

- Name: Technical University of Sofia
- Address:
    - Technical University of Sofia,
      Kliment Ohridsky Bd 8
      1000, Sofia, Bulgaria
- Phone: +3592623073

Output 2 Leader:

- Name: FOSS Research Centre for Sustainable Energy, University of Cyprus
- Address:
    - University of Cyprus,
      Panepistimiou 1 Avenue
      P.O. Box 20537
      1678, Nicosia, Cyprus
- Email: foss@ucy.ac.cy
- Phone: +357 22 894288