Lappeenrannan teknillinen yliopisto
School of Business and Management

Software Development Skills: Full-Stack

Learning diary

Steven Gobet
000738712

I. NodeJS

Period: April 7 – April 25

Started the introduction to Node.js. Node is a JavaScript runtime (allows us to run JavaScript code on a server). Why using Node? Because it is fast, efficient and highly scalable. It has moreover the same language for the front and backend. Node works on a single thread using non-blocking I/O calls (unsynchronized and we are not waiting that one thread finished his work). That means that events are run asynchronously, they are run one by one. Node is therefore not useful to run CPU calculations because it can block all the server. It is however useful for rest API (uses HTTP requests to access and use data), we can make all our relation between databases. MongoDB is good with Node because it is fast and scalable. Node is also great for real time applications (chats, social networks, etc.).

Node Package Manager (NPM) is used to install packages and modules such as frameworks, libraries and other smaller tools. When we install a package, it is stored in folder called node_modules, created automatically. All dependencies are stored in package.json which every node projects should include (name, version, list of all dependencies, which are the packages our app needs to run).

Useful commands:

Npm init => generate the package JSON file
Npm install => install the dependencies provided by the package.json

First step when creating a node project: *'npm init'* to set up a new Npm package. All the packages needed in our project must be added in the package.json file. So, when we do *'npm install'*, it will install all the packages listed in the file and all the dependencies associated. Node has a lot of modules and packages. We can install some with Npm, but we can also create our own modules, which are files that we can export in another file (can includes variables, functions, classes, etc.). In the website of NodeJS, we can see the common modules where they are documented.

By following the video, I learned to use some useful included modules, such as os, fs, url or events. For events, I learned to create an emitter and a listener that will realize something when a certain event sent by the emitter is caught by the listener. For example, when the event "event_message" is emitted by the emitter, the listener will print what is included inside this event, in our case an id and a message.

From those event principles, I also learned to run a web server with http module, how to create web pages according to a certain path and also created a dynamic path that should suit all the different paths. With the dynamic path, I also had to make a dynamic content-type so that can be linked to the type of the file that the customer wants to load. Thus, I was able to run a web server that can show the page that a customer has requested. I also created an account in Heroku to have a web server reachable from outside local host. For example, Heroku gave me the address https://damp-brushlands-57164.herokuapp.com.

So when a customer make a request for https://damp-brushlands-57164.herokuapp.com/, it will load the home page. If we search for …/about.html, it will load the about page. If we load something different than those two options, it will load the error page.

II. MongoDB

Period: April 25 – April 27

During this session, I learned how to install and setup Mongo DB locally. Mongo DB is a noSQL database, which means that we store data and collections of documents (unlike SQL database when we store data in columns and rows). Documents are like JavaScript objects or JSON objects, that's why it is good to use MongoDB with Node JS and also because it is very scalable. In MongoDB, we don't have to map out the data structure beforehand. In MySQL, we must create all the tables, the columns, every property, etc. In MongoDB, we have the freedom to structure the data however we want. The data can be stored and accessed together.

With the help of the application Compass for MongoDB, we can easily see, create databases and collections (like a table that hold documents of data). But the goal of this video is to make us comfortable with shell, the commands and different methods to work with documents.

By typing "show dbs" in our mongo terminal, we can see our added database named "TEST1", as well as the collections we also created inside: "Products" and "OtherProducts". So, via the terminal, we can create new databases, delete, etc. We have to keep in mind that the command "show dbs" only shows the dbs that contain something.

To insert a document in a collection, we use the command:

```
db.'CollectionName'.insert({
        title: 'Post one',
        body: 'Body of the title',
        category: 'News',
        like: '12',
        tags: ['news', 'events'],
        user: {
                name: 'Steven Gobet',
                status: 'author'
        },
        date: '25-04-2022' / or \ Date() / to put the current date \
})
```

In the example above, we can see that we are able to insert whatever we want: a string, a number, a date, an object or an array. We can of course add multiple documents at the same time by using *"db.'collectionName'.insertMany([{Post1…}, {Post2…}, …])"*.

Command to query (or find) a document: *"db.CollectionName.find().pretty()"*. Pretty is useful to make the document readable. We can also find the documents that have a certain field, sort the documents, count the number of documents, limit the research, etc.

Examples:

- *"db.posts.find({ category: 'news' }).count()"* → count the number of document that have a category "news" inside.

- *"db.posts.find().forEach(function(doc) { print("Blog Post: " + doc.title) })*" → Will print the title of each documents.

We can also update a document but we have to keep in mind that it will completely delete the document and recreate it the way we added into the "update parameter". To change only few lines of the document, we use *"$set"* and so it will just update the field mentioned in $set (or create it if it doesn't exist).

With mongoDB, we don't have to link two collections with primary and foreign keys. We just can embed a "subdocument" into the first one. We also can create separate document but this makes it easier in a lot of cases. By using the *"$set'"* before, we can just create a "comment" field for example and add it to the document. To search for the information inside the subdocument, we use the key word *"$elemMatch".*

It would be useful to make some text search. We can add indexes into the field that we want to search by.

- *"db.posts.createIndex({ title: 'text' })"* → Here the index is named 'text'.

- *"db.posts.find({ $text: { $search: "\"Post O\"" } })"* → Here it will search for the text "Post O" and show the different elements that have it in the title.

Thus, we can imagine that if we have a document with the title "Post One", it will print it.
Finally, I also learned to sort by doing a greater or less than something. For example, if we want to print the posts that have more than 6 views, we can do:

- *"db.posts.find({ views: { $gt: 6 } })"*

To really develop and test a mongoDB project, linked to an application, we can use Atlas which is a cloud database.

III. Express

Period: April 27 – May 6

Express is a fast and minimalist web framework for Node.js. Thus, Express makes building web applications with Node much easier and can be used whether as a server or API. It uses the language JavaScript and is very popular because it gives full control of request and response.

In the package.json file, we can create several scripts that will be loaded when we run *"npm run _Scriptname".* We then created two setups: one named "start" and will be run when we are in production and another called "dev", when we are in development. For the moment, "dev" mode just run the server with Nodemon which can track the changes in the code, so we don't have to reload it after every changes. Thus, Express helps us to make a Node.js web server in a very easy way. With this video, I then learned to create a simple web server with express that

print a webpage when doing a get through a route (or endpoint). However, this way of doing is unideal because we have to create route manually for every pages. Fortunately, Express provides functionality to create a static folder. By calling the function *'use'*, we can set a static folder that will automatically detect the request path and run the code included in the right route file. So if I search for localhost:5500/index.html in Google Chrome, it will automatically open the script included in index.html. The routes are then crated along the time.

With the help of Express, we reached what we achieved with Node.js very easily and fastly. With Node, we had to load all the html and CSS files. Here with Express it is just with one line of code to create a static folder. Usually, Express is used to make more complicated functionalities like JSON APIs that we can connect from the frontend, create dynamic app, etc. Middleware functions are functions that can they have access to both request and response object, and modify it (add things, change things, etc.). For example, I can create a middleware function that will print the URL requested by a customer in our terminal.

A good point I learned also is to subdivide the project into several little files. It could be easy to export a variable, but it wasn't so easy to export a *'app.get'*. I had to do *'express.Router()'* to be able to use *"router.get"* the same way as an "*app.get"* in the main file (using "*app.use"* into it).

I also learned to deal with data we received from the frontend, while creating a user for example. Express has very good functionalities to make thing easier because it includes parser, that we just have to initialize as a middleware. Then we use *"express.json()"* to handle JSON format and *"express.urlencoded()"* to handle URL encoded data. To update a member, we use a *"put"* request. To delete a member, we use a *"delete"* request from the frontend. For these 2 scenarios, we can use the same route, which is *"localhost:5500/api/members/<id's member>"*. We can use the same route because *"get"*, *"put"*, *"delete"* are different methods.

In the video, it also dealt with template engine. A template engine enables us to use static template files in our application. At runtime, the template engine replaces variables in a template file with actual values and transforms the template into a HTML file sent to the client. This approach makes it easier to design a HTML page. The template engine I used for this course is Express Handlebars which helps to build semantic templates effectively.

IV. Angular

Period: May 6 – May 9

In this tutorial, I learned how to create a Tour of Heroes, which is an application that uses Angular. Angular is an open-source web application framework based on TypeScript, which is a language closed to JavaScript.

Firstly, I began by creating an initial application using the Angular CLI[1]. Thus, I learned how to set up the environment of the app, create a new workspace and initial application project,

[1] The Angular CLI is a command-line interface tool that we use to initialize, develop, and maintain Angular applications directly from a command shell.

serve the application and make changes to the application. After this first step, we have a web application that shows ‘Tour of Heroes’ in the webpage.

So, the application has a basic title. Next, I created a new component to display hero information and place that component in the application shell (= web application page). At the end, it shows one hero with its id, and we can modify the name of the hero from the application shell.

Once this step concluded, it could be the right moment to show an entire list of heroes, where a user can select a hero and display the hero's details. Thus, a list of heroes is shown with the help of the tutorial, and we can click on a hero to see its details. We are also able to modify its name.

At the moment, the HeroesComponent displays both the list of heroes and the selected hero's details. Keeping all features in one component as the application grows will not be maintainable. We'll want to split up large components into smaller sub-components, each focused on a specific task or workflow. I thus moved the hero details into a separate, reusable HeroDetailComponent. Refactoring the original HeroesComponent into two components yields benefits, both now and in the future. Indeed, it reduces the HeroesComponent responsibilities. We can evolve the HeroDetailComponent into a rich hero editor without touching the parent HeroesComponent. We also can evolve the HeroesComponent without touching the hero detail view. Finally, we can re-use the HeroDetailComponent in the template of some future component.

The Tour of Heroes is getting and displaying fake data. Components shouldn't fetch or save data directly and they certainly shouldn't knowingly present fake data. They should focus on presenting data and delegate data access to a service. In this tutorial, I created a HeroService that all application classes can use to get heroes. Services are a great way to share information among classes that don't know each other. Moreover, I implemented a message box that shows the history of which hero I clicked on. We can also clear the message history.

Then, I added a Dashboard view and added the possibility to navigate between the Heroes and Dashboard views. So, when a user clicks on a hero name in either view, he can navigate to a detail view of the selected hero.

Finally, I learned how to make HeroService gets hero data with HTTP requests. So, users can add, edit, and delete heroes and save these changes over HTTP. I also implemented a feature that allows users to search for heroes by name. This tutorial was very dense and it was maybe too much of reading. As we can’t see the app building in real time, it was harder to understand what happened. The videos are far more pleasant to follow and instructive in my opinion.

V. MEAN

Period: May 9 – May 22

Video n°1:

With the help of the first video, I learned how to make an authentication. To do it, I built a backend Express server and an API to let users register, log in and get a JSON web token when

they authenticate. One authenticated, users are able to log in and see the dashboard, the profile and every route we want to protect with authentication.

For this project, we used almost all the technologies we saw through the previous courses. The project includes a REST API using Nodejs and Express (Mangoos and Passeport were new for me). With Mangoos, we have a user model with name, password, etc. The API will be used to create endpoints for registration, adding users to the database, authentication, getting user's profile, data, etc. What was also new was Token generation and authentication.

CORS will be used to make the link between frontend and backend because they are using different ports. So, with CORS, we are able to send request between them. Angular-JWT is very useful to verify the users' Token and give him the access to his profile for example.

Video n°2:
In the app.js (main server entry point), we initialized it by bring in all the dependencies. At first, I thought that the name *app.js* wasn't important so I called the file *server.js*, as it was our server file. I just realized after that this special name was needed, otherwise it cannot find the module while running the *"npm start"*.

All this video was a good remaining of what has been done during the first course: how to initialize the server, and all the backend globally. But it was also a good remaining of what has been done during the third one (express): how to initialize a static folder, its use, etc.

CORS is a good way to make routes because it is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served. So, with CORS, there is an easy way to control the access of certain resources. I made the website public so we didn't need to restrict the access from a domain but it would have been useful to authenticate because we can disable some routes if the user doesn't send the right token.

To sum up this video, I installed and imported all our useful dependencies for this project, I connected to my MongoDB database with Mongoose and checked for connections or error in the database. I thus created a database, a route and a secret to access it. I also initialized express, brought users' folder from routes' folder and so created a route for users within the website (register, authenticate, profile, etc.). Finally, I initialized CORS middleware to control the access of certain resources and then started the server to see all the actual functionalities running.

Video 3:

During this video, I created a mongoose model for the users, that specifies the required fields: name, password, email, username, etc. It encapsulates all the database functionalities within a model file to have everything centralized into that model file. So, at the end of the video, I was able to store a new user threw the route *"localhost:3000/users/register"* and encrypted the password. In the user model, I created a function *'addUser'* that hashs the password and store the user in the database (which we can check with the Mongo database).

To test our post routes, I used postman which I didn't know and is a very good tool to do it.

Video n°4:

After this video, I implemented the authenticate functionality as well as made the token system working (protect certain routes).

During the video, I had to face a problem when I wanted to test the post request to authenticate a user. I had to use the *"express-session"* middleware to be able to login into the database. I also had to modify the code, when initializing the token during password comparison (in the user model → /models/users.js), because I had a payload error.

One thing very important I learned during this video was how the token is working to protect a route. When we get authenticated, the server generates a token and will send it in return to the user. This token will be needed to have access to the routes we protect (for example /users/profile). The users thus must include his token within the header when requesting the web page. I finally noticed that instead of the Youtuber's version, the id of jwt_payload in passport.js is correctly stored so I just needed to call it with jwt_payload._id.
Thus, when including the token wile asking the get request to profile page, I am authorized to access. When note included, I can't access the profile page.


```
{
  _id: '627abd0ceaf3a3ac70d17157',
  name: 'Steven Gobet',
  email: 'steven.gobet@insa-lyon.fr',
  username: 'stevengo',
  password: '$2a$10$gFOmRd3VP8j/uBP/oEdxg.NHUA3XNSFz6nDr8IWcoIrexXyZiH8d.',
  __v: 0,
  iat: 1652288315,
  exp: 1652893115
}
```


Video n°5:

The backend is almost finished so I started to right some front-end with angular.

It was impossible to make it run with angular-cli. I downgraded node to v12, installed angular-cli but I had an error in the /node_modules/@types/node/index.d.ts (20,1): Invalid 'reference' directive syntax. Everything was broken and when I wanted to resolve dependencies issues, new errors came again and again. So, I just reinstalled all the newest version with the hope to manage the issues mentioned in the video n°7.

Video 6:

During the video, I faced several issues that differ from the video's code. Firstly, I had to put *""strictPropertyInitialization": false"* into the tsconfig.app.json file to avoid initializing all the variables. It shown an error while compiling. In the same manner, I had to add *""noImplicitReturns": false"* to be able to not be forced to put an *"else"* after an *"if"* (error TS7030: Not all code paths return a value). Finally, I also had to add *""noImplicitAny": false"* in the same file (error TS7006: Parameter 'user' implicitly has an 'any' type) for avoiding having to explicitly define all paramaters.

I found all those information in stack overflow.

I finally had to modify FlashMessagesService because it didn't work with the syntax of the video. The correct way to implement flash messages has been found on:

https://github.com/moff/angular2-flash-messages.

At this point, when we try to register, it pops up an error if a field is missing or if the email is not valid. However, I noticed that when I filled in the fields and deleted the content, instead of showing again "Please fill in all fields", it showed "please use a valid email", or simply nothing. It is because the function *"validateRegister"* didn't take into account the length of the content, just the type. So, since a field had a string at some point, it was not undefined anymore, it was a zero-length string. I thus also checked if *"user.field.length == 0"* and resolved this little issue.

Video n°7:

The goal of this video was to register a new user. I faced an issue due to my angular/cli version but I managed to resolve it. Firstly, I had to remove *"map"* in the auth.service and just return

"this.http.post('http://localhost:3000/users/register', user, { headers });".

Then, I had an error for the post request and so the redirection could not happen. I had to change the *"register.component"* by declaring a *"data.Register"* variable to store the data received and then, everything worked (even though I don't see any difference).

Video n°8:

The goal of this video was to be able to login with the frontend, as well as logout. I also stored the token returned by the backend, so we were able to secure some routes. Everything went well and was interesting.

Video n°9:

I implemented angular ajwt, created the profile page and send the token within the header so the user can visit his profile page. I spent literally 15 hours to resolve a 401 unauthorized from the server, even with the token correctly sent. I checked every line of code by comparing it with the course code and the video's code, where everything seemed to be correct. I also checked every comment below the YouTube video, did a lot of Internet searches, changed node module, changed the version, etc. At a point, I completely broke my Npm and I had to resynchronize all my project (and tried to find a more stable version of dependencies). The problem with changing of node modules in a middle of a project was that can lead to have to write different code. And so, in my passport.js, I didn't had problems at the beginning with "*User.getUserById(jwt_payload._id, …"* but changing of modules changed this syntax. I thus had to check the way that the data was stored and change it into "*User.getUserById(jwt_payload.data._id, …"*, as you can see in the screenshot.

In brief, I changed a lot of things both inside the code and inside the dependencies which brought me more problems. Resolve my dependencies problem brought me issues in the code that was considered as finished and worked since 5 videos and I had to spend a very long time to find the solution because it wasn't obvious to retest all the previous working functionalities.

For hiding the routes if we are logged in or not, I had to use *"@auth0/angular-jwt"* instead of *"angular2-jwt"*. It is quite the same code, the function *"loggedIn()"* is just reversed (named *"isTokenExpired()"*), returns False if Token is good, True if not. So I put *"!isTokenExpired"* into the function to return the correct value.
Finally, I protected the route /dashboard and /profile so that we get redirected to /login if we request these pages without being authenticate.

Video n°10:

With this video, I deployed my application with Heroku, which we already used previously in this course. This time, as my GitHub account was already linked with Heroku for another application, I had to change some things to make it works. I spent a lot of time trying to make it works but at the end, I had to separate my project because I couldn't launch the Heroku server. I thus created a new git repository with Heroku to deploy the application. I used MongoDB Atlas to deploy the database. You can follow the link to visit it: https://boxing-ring-lut.herokuapp.com

Added features:

I wanted to create an application that shows the next upcoming boxing fights, research by category, and the last news in the boxing area. I didn't have the time to achieve those features and synchronize the data of the website with another site seems very tough.

However, I tried to add some functionalities to the existing code. The website can thus check if the Username already exists when registering. I added the code into the users' route (backend) and into the register component (frontend). We can also modify our password from user's profile. I added a button in the profile page, added a password component and all the required code to make it works. The route is protected in a way to not be able to access the page without being authenticate.

I wanted to have a web development experience and this project was the first web project I ever did, which was both interesting and complex. The videos were very clear and helpful to understand what happens and why we do some things. The hardest parts were to face issues relative to our version as it frequently required to change little things. It could have been great to have more time to achieve something more complete but that was what I can do giving the restricted time and the fact that I was quite busy at the end of the semester. I am thus satisfied with the features I was able to implement and with all that I was able to learn during this course.