Lappeenrannan teknillinen yliopisto

School of Business and Management

Sofware Development Skills

Pyry Santahuhta, 0545254

LEARNING DIARY, FULL STACK MODULE

LEARNING DIARY

You can check each part’s status from when it was concluded in the courseworks/part # folder in github. The Project folder contains the most up to date version of the portfolio.

1. NODE

05.06.2021

I started the course today. I started by going through the course Moodle page and grasping the core contents of the course. I recently completed the Front-end course from the same series as this course, and I think it was smart doing these in this order.

Learning one possible full stack solution and managing it all on my own will give a good glimpse into what is going on behind many websites. I have heard of all the parts of the MEAN-stack, but I’ve never actually used them myself, so learning all of those while also delving into full stack development is very efficient. MongoDB is interesting since I have had two SQL courses, so learning a new database model which isn’t relational sounds fun. NodeJS and ExpressJS will also be useful for my expanding my Javascript skills.

Today I didn’t do much actual work else than making a local git repository with the base items needed for the course, and of course I started this learning diary.

07.06.2021

Today I started on the first section, Node. The crash course started on learning a bit about Node, how it works asynchronously and via events and callbacks. I also learned that Node is single threaded, so it isn’t ideal for CPU heavy projects. Other types of projects Node manages well due to it being asynchronous.

The lesson continued with some simple demos and test on how classes and modules are imported using a system called CommonJS. This is simply used by creating variables to hold the module with require to get it. After the class testing I did some demos of the path and File system modules. Learning a bit of the functions these methods use was fun and I think they will prove useful later on in the project. With these tools I can affect the user's filesystem from inside code and get the path of any files no matter the system they are using. Next up I tested out the OS and URL modules. The OS one can gather information about the user's system for system specific applications and processes. And URL is just used for handling URL's and getting different parts of the URL easily. Last demo that I did today was for events. With eventemitters I can create events about anything and listeners to watch for those events to fire. For me this feels like the most important module since events are what usually drives applications, at least with mobile development onClickListeners and such were very important.

8.6.2021
Today I continued with the Node tutorial. I started out by creating a logger that uses the EventEmitter I learned about previously, combined with uuid in order to create message logs with unique id's. I tested it out with index.js with a few different messages and it worked like a charm, it created message objects that contained the msg and the id it generated.
Next up I created my first barebones local http server. The webserver only contained the text Hello world, but it worked. It was done with the Http module from Node. It listened on port 5000.
Lastly, I created two versions of a node server. First one was a bit clumsy with all of the different request urls in their separate if statements. When the site would grow larger this would grow bloated. The second solution was more elegant by holding the filepath of the requested url in a variable and checking whether it is index.html for just a slash and otherwise the filepath iss built straight from the requested URL. After that I just check the content type with a switch statement and then read the file and write the content on the page as a response. This time the port is also changed into working dynamically by using the environment variable. That was about it! A web page with multiple pages running on a server. After that I deployed the site to

Heroku. This was a bit more complicated for me since it was a bit different for me from the tutorial video. The difference was that my folder for the server was not the root folder of the repository since I have the project files and other subjects for the course. In order to push it I tried a few different methods, I tried using Heroku buildpacks but I couldn't find one that would be compatible with my project. The solution was to use git subtrees, and push from there. This was new to me and a pleasant way to learn more about git. With subtrees it worked! The site was deployed at: https://radiant-lake-31418.herokuapp.com/ and requesting the about page worked and requesting for an unknown page took it to the 404 page. Node section complete!

2. MONGODB

9.6.2021
Today I started the MongoDB crash course. The course started by introducing MongoDB which is a no-SQL database. This means that it isn't a relational database. This is great for evolving databases since you don't need to specify all of the tables, rows and relations of the database initially, and you can just add them as you go. This seemed like a much more intuitive and usable way of doing it since I've had many frustrations before with SQL when I forget to add one little thing and I have to drop the databases and create it again.

After the introductory bit ended, the tutorial went to show shell commands that are used in handling the documents. First off were showing how collections and connections work inside databases. After the creation of a collection, simple insert statements were made to get data into the documents. After that I learned about querying the data. Everything felt familiar from SQL and other programming languages like Java, since almost all the commands in Mongo can be chained like in Java.

I then learned about sorting, counting, limiting, and finding specific rows. These are all useful tools to be used with querying the data you want. Next up I learned two ways of updating the document, one replacing all the data and the better one in my opinion replacing only the wanted data. Next up was modifying the tables themselves, which is the most annoying part of SQL in my opinion. In Mongo renaming fields and deleting certain rows comes with no problems since the structure is not as set in stone as in SQL.

The next topic was a bit more advanced, embedding subdocuments inside documents. This was done by editing the 'comments' section of a collection and adding data for different things about the comment, like the user that posted it, the date it was posted on and the comment itself. Working with subdocuments continued with finding them, it worked intuitively with similar embedding to the table itself and searching for specific commenters for example is possible. After that I created an index to the title of the posts and used that with the search function. Next was greater than and less than and they work as you would expect. Memorizing how all of these commands are formatted would prove rather difficult at first and that’s why the cheat sheet provided in the description will probably prove useful in the future working with Mongo.

Lastly the tutorial went through the graphical interface for Mongo which is called Compass. This will probably be useful for having a quick overview of the collections or the data and seeing the structure in a more organized manner. For working with the databases though I believe that the shell will be more efficient. Finally, MongoDB Atlas was looked at, which is a cloud-based database solution. This can be useful in larger scale projects, but I think for this course at least using the local databases will suffice.

3. EXPRESS JS

30.6.2021
Express JS is the third tutorial in the series. Express is a fast unopinionated and minimalist web framework for Node. Express is used in the back end of development. With Express building Node web applications becomes much easier and it is very light and fast. The basic syntax for an Express server needs route handlers with express get function where a response is sent back to the user, and then just listen on a port. Handling routes works with express functions get(), post() etc. With these you can manipulate the data going in and out of the site. With get for example you can fetch from databases, load different html pages and return JSON’s. Lastly the slideshow covered Middleware functions, which are functions that have access to the req/res objects.

I already had Node.JS so the only requisite software I downloaded was postman, which is an API platform.

Next up I created the simplest form of a web server using the things mentioned in the slideshow. I used app.get to get the request and response objects and I used res.send to send back data to the user reaching the server. I sent just a simple hello world. For developing and deploying the site I just copied the Nodemon scripts from the Node crash course since they were just the same. Next I just edited the res.send to send a file instead, and I sent index.html. Then I changed the current system to just use middleware with app.use, I used the static function from express to make the public folder static. This does all of the previous stuff just in one line automatically, so I don’t have to specify each page with its own response.

The next thing I learned about was using Express to create middleware and handle JSON’s to create apis. To create my own middleware, I just created a simple logger with the req and res and the next parameters. And then I just used Express’s use function to init the middleware. The second thing I did was an api to access a list of js objects and return them as a json to the server. For this I used just used Router to specify when the user gets from the api route. Sending the full list of JS objects is simple, I just respond with the json version of the file. Getting a single entry for example by id is not that hard as well, I just need to use some method to filter only the wanted items. This can be done with JavaScript’s filter function. I filter the items by comparing them to the desired id and if it is found, I send the json for that item. Otherwise I send a 400 bad request status and a msg telling the item was not found.

I then worked with API’s a little more. I created methods for creating new items into the json and updating or deleting existing ones. Creating these methods didn’t require much new Express knowledge, just some JS code and utilizing uuid for generating id’s for new members. Then I tested out these functions with postman, everything worked fine! After that I prettied out the website with handlebars to create layouts. I created these into their own views folder. Using the handlebars came new to me but so far they haven’t seemed too hard. I needed to use set templates and variables and a for loop to loop through the member items. Visually I just created a simple form to add new members using the functions I already did and listing the members below the form. I also created a button to visit the API JSON file.

That was the final thing for the express tutorial! Next up on to Angular.

4. ANGULAR JS

7.7.2021
Angular is an application design framework for creating efficient single-page apps. This tutorial seems to work a little bit differently from the previous ones, as it doesn’t have a video. This time I will be following along a text-based tutorial. The introduction for this crash course covers what I’m going to do during the tutorial, and it sounds really fun! I will be creating a ‘tour of heroes’ application, or whatever else I want it to be. The introduction also covered the topics of Angular I’m going to learn about, including directives, components, data binding, pipes and routing.

I started off by installing the Angular cli using npm. After that I created an Angular workspace using the angular cli command ng new and launched the application with the serve command. The first thing I did with the project was creating my first component. Components are what Angular is built out of and creating new ones happens with the Angular CLI command generate. I then learned about how the metadata properties are generated and what they do and how ngOnInit() is a lifecycle hook that is called shortly after creating a component. I then created my first hero and made it visible from the html part of the component, As I did this I started to understand how a component is formed out of a typescript file, html file, css file and a typescript test file, simple enough. Then I created a class or “interface” for a hero, this one just contained an id and a name and I learned how to display this too, just like in other object-oriented languages. I then learned about piping when I piped the name of a hero to the uppercase function in the html. Lastly for editing my hero I needed a two-way binding which was done with ngModel, for this to work I needed to import formsmodule into the main app module typescript file.

Tutorial part two started off by creating a mock list of heroes to be displayed. I just created Its own file for it and imported the list to places where it is used, mainly heroesComponent which is the only component I’ve created. Here I just saved it to a property with a declaration. In order to display a list in html I need to use *ngFor in a <li> container, which is just a for loop for html made possible with angular. Then I added some styling to the heroes. In order to know what the selected hero is I need to add an onSelect(hero) function to the <li> container. Then I implemented the onSelect function in the typescript file to make this.selectedHero = hero, which is the hero that is sent as a parameter from html. After that I just added the html to display the data of the selected hero and to hide it with *ngIf when no hero is selected. Lastly I added some stylizing and part 2 of the tutorial was done!

13.7.2021

Today I’m going to learn about feature components, which means creating components for different features of the application. My first feature component will be the heroDetailComponent which will contain the detail things I created last time. Creating the component happens the same way as the main component. I then just copy pasted the detail section from the Hero component to this one. This way of working will keep the files much more manageable, and it is a necessity in larger scale projects. I changed the heroes html as well to show the herodetail by adding its own container. These changes were mostly moving stuff around to better accommodate for future development.

14.7.2021

Today’s plan is to create services to handle data access in order for components to focus on just displaying data. Firstly I created the service Hero and made it providedIn root so every component can use it if the wish to. I then created a getHeroes() function to move the data fetching to the service from the component. Since I'm still using the mock heroes this was very easy to do by just returning the heroes list. To use this service I import it into the heroes component and inject the service by adding it to the constructor. Then I created a getHeroes method which uses the service to fetch the data. This method is then called on ngOnInit to call it everytime the component is created. To make my getHeroes function Http request friendly I need to make it asynchronous. In order to do this I use an observable that emits the array as an observable. To make this work on the component side I need to change the getHeroes function to use the subscribe(), this is needed for server applications. The subscribe function waits for the observable to emit the array, which can take an arbitrary amount of time. After it emits, the function just sets the components heroes property as the emitted array.

Next up is creating a message log for the application, this will log all activity of fetching data. For this we need a component and service called message. We display this component similarly to the heroes component in the main html file. The message service will hold and edit the message array accordingly and the component will display it. For the service we need a push method to push new messages to the array and a clear method to clear the message array. This time we inject the messageService to the heroService, so it can use the service, and we can also inject it further into components and services that use heroService. We use the messageService on getHeroes to add a message that we fetched heroes. The messages are shown from the messages component's html file. It contains a familiar *ngFor loop to loop through the messages, presenting each one. I also added a message to be sent when a hero is selected, the message contains the id of the selected hero.

The fifth part of the tutorial contains navigation. I'll create dashboard to present the topheroes and navigation elements to navigate through these different views. First I need to create a module for the app-routing. This will handle all the different routing from view to view. I changed the heroes view with router-outlet to show what the router gives it. This works already and if I go to localhost:4200/heroes, it shows the normal heroes page. But this still needs some work since navigating with only the url bar is very cumbersome. Next I create the second view, dashboard. This works similarly to the normal one but just shows a slice from 1 through 5 so I only get 4 heroes. Then I implement the dashboard's route in the routing module and add a default route to the dashboard so when the url is only / it will redirect to the dashboard. I created and stylized some buttons to navigate to dashboard and heroes, they work well! Next I will move the hero details from heroesComponent to their own heroDetail component so they can be viewed from both dashboard and heroes view. Then I needed to add the route of details into the routing module, this time I needed to use a placeholder for the id with the ":" sign, this way every id has its own url. Then I made the links in heroescomponent and dashboard take the user to the detail view with the parameterized route. Then I removed some code that was no longer used. Heroservice needed some changing to display the details of a selected hero, since I need to fetch a specific hero now instead of just selecting it. For this I created a gethero function that finds the wanted hero by id from the array and adds a message that the hero was fetched. Lastly I created a back button to go back from the detail view by calling this.location.back().

Last part of the tutorial makes the application get data from a server rather than a mock hardcoded array. For this I need to enable HTTP services, and since I'm not using a real data server I will need to simulate one. I will do this with angular-in-memory-web-api. I create its own service for the "server" and add the heroes array there. I also made a generate ID function so that new heroes added get an id added that is one higher than the last hero. Now the heroservice that fetches the data needs to use HTTP to do so we need to import some HTTP modules. Also as a side note I made messagingservice its own method log() since it is used so often. Then I changed the getHeroes and getHero to use HTTP instead of just returning from the array. It was simple to do with just a http get. For server data fetching, error handling is necessary so I pipe the getHeroes functions into a catchError function that does basic error handling. For updating heroes I added a save method that calls updateHero to the detail component, and then implemented the function again in heroService which worked very similarly to the fetching but this time I just used HTTP put. Next up I implemented adding new heroes, since the "server" generates id's for new heroes automatically, a new hero only needs a name. Then I added an add function in heroes that is called from the html side to call for heroService to add a hero. I needed to remember to check that the input field isn't empty so that nameless heroes can't be born. Deleting a hero works almost exactly the same, I just add a button to every hero and make it call a delete function that again calls heroService to delete the hero with HTTP delete.
The last thing I implemented was the search. I again created a function to fetch the matching heroes from the server with heroService and display the search component below the dashboard. The search component just consist of a search box and the list of heroes that match the search. The server searches on every keystroke, but to prevent too much load on the server I added a 300ms timer to check if the user keeps writing before fetching. Otherwise very many requests could be made. I also added a distinctUntilChanged() method that checks if the filter text changes from the last one. That concludes the Angular tutorial, tour of heroes is complete!

5. MEAN STACK PROJECT

16.7.2021
After learning about the parts of the mean stack individually, it is now time to use them together in a full-fledged mean stack application. This application is going to be simple, it will just have basic authentication and a few pages, but all parts of the mean stack will be used. For this tutorial I will not be writing down everything I do, since I have already covered many of the things used here in the previous sections. Rather I will be documenting new things I learn about and interesting interactions between the different parts of the stack.

The first video was only an introductory one, presenting the final product and going through what will be used to achieve it. The second video also just set up express and the routes for the application, these were done similarly to before. One new thing in the second tutorial was using mongoose to access our mongo database. I created a config file where I designated the path and port for the database and then used this path with mongoose.connect(). I then created an error and connection function that sends logs whenever the app connects to the database or encounters an error.

During the third video I created a mongodb user schema for my project using mongoose. This was the simplest way of setting up a database I have encountered thus far on my coding adventures. I'm really starting to like Mongo over SQL. I created some functions for finding users and adding new ones. For adding new ones, I needed to encrypt their passwords. This was done with bcrypts salt generation and hash functions, they did most of the work for me. Of course, I needed to do the jus side also for these functions to send something back to the user informing if the registration completed or not. To login my comparePassword function needs the hash created in the encrypting, and the candidate for the password.

20.7.2021
Next up we continue on with the back end of the site. I'm creating authentication for the website's protected elements. I achieved this with Jwt's Passport. Jwt has different "strategies" for the authentication and different ones can be found on their site. This authentication is fairly standardized, and I didn't deviate from the most common strategy. Basically, the client gets created a token when they login and using this token as a header for authorization they can access protected parts of the site. I had to make a few different changes to the video since they were outdated. For example the opts jwt fromrequest worked with a different way of extracting it. Also the way I presented user in the sign part needed to be in JSON format. Next up on to front end with angular!

Working on the front end angular was familiar from the Tour of Heroes application, I created components for all individual sections of the site.

After creating the components I needed to route them, this happened familiarly with Routermodule and a list of the routes for the page. Then I just took Some bootstrap html for the navbar and made the links routermodule friendly by setting them in the following format:

```html
<li class="nav-item" [routerLinkActive]="['active']">
        <a class="nav-link" [routerLink]="['/dashboard']">Dashboard</a>
</li>
```

As I was creating the registration form, I needed some way to inform the user if their input is correct or if something is missing. For this I used flash messages, I just added its own container in the html and used the show function to show positive or negative messages to the user. To check if the email is correct I used a ready-made regex line. Everything else was familiar from the Angular tutorial, on to the next!

29.7.2021
Registration still needs the back end; this was handled into the authService angular service. The subscribing to the observables is a nice way to visualize how the logic works, since it works just like subscribing to a Youtube channel for example. The "consumer" of the data coming from the observable is notified every time the observable uploads, so the code in the consumer doesn't run before it has gotten data, which is useful so that the data values are always set. After the consumer receives the data, I just check if the data was successful (If it passed through the email and data checks) and redirect the user to the login screen.

For logging I use similar techniques to the registration, with subscriptions and angular functions. After the user is logged in, they need to use the token which they got from logging in to access the profile and dashboard. For this I need to store the token somewhere, this is done on the local storage of the user, so that only they have the token saved, this helps with security. Using the local storage was easy. To logout I just created a button which nulls the user, the token and clears the local storage and takes the user to the login page. Now I needed to use the local storage in the profile/dashboard pages to use the token. For this I created the loadToken function. The profile html uses the *ngIf again to check if the user exists before loading the page. This way no html data is displayed

The last video was about launching the site to Heroku again. I'm glad I already made one Heroku app before on this course. This time I needed to do some changes; all the localhost URLs had to go. They were simply changed to just not have the localhost part. The other change was for the MongoDB. I couldn't use a local database server and port, so I used Mongo Atlas. This varies from the tutorial video since it uses mLab, which isn't used anymore. I needed to delete my other Mongo Atlas database since you can only have one free cluster. I then added the link of the Atlas cluster to my config file holding the database URL. After that I again used git subtrees to push to Heroku and voila! The site was online with functioning registration and login at: https://auth-app-mean.herokuapp.com/. That concludes the course, Thanks for reading!