

Software Development Skills Full Stack, Online course

Author: Kim Venho | 000430285

Lappeenrannan teknillinen yliopisto

Table of Contents

Learning Diary	1
1. NodeJS.....	1
2. MongoDB	4
3. Express JS.....	5
4. Angular.....	12
5. MEAN Stack.....	22

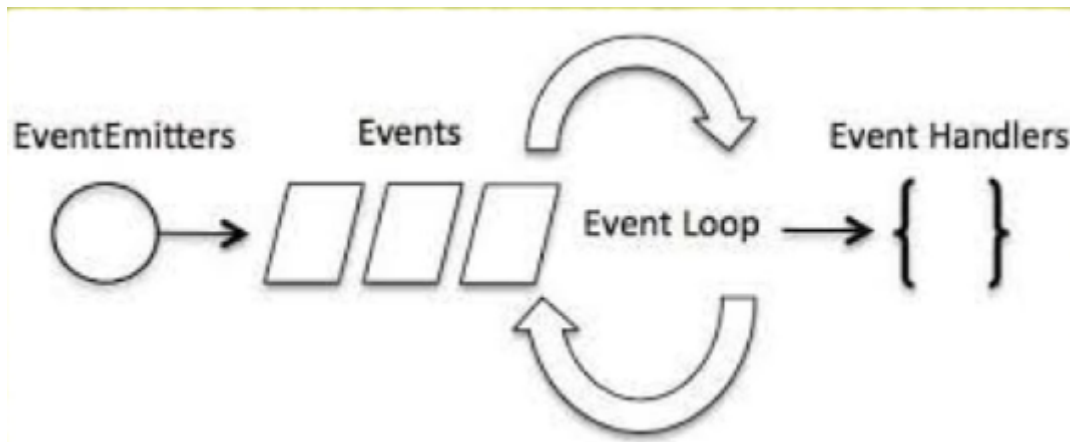
Learning Diary

1. NodeJS

20.05.2023

Today's topic is Node.JS. I learned that it is a javascript runtime. It allows you to run javascript on server side. It is built on V8 JavaScript Engine, which I've heard is a powerful one.

Node is single threaded, and uses non blocking I/O calls. It uses event loop to cycle through its tasks:



... due to these facts, it is not good for heavy calculations (if those are needed to be done, you should call it with node and then fetch the result when it's ready). Basically it is very good on anything that is not CPU intensive.

Node Package Manager (NPM) is as the name says, package manager for Node. It can be used to download packages, which are then installed in node_modules folder. The package.json file that comes when you initialize the NPM for your project holds information about dependencies of your project. With that you can share your project dependencies easily without actually sending copies of them (NPM can download all required dependencies before starting your application).

We started our Node journey by initializing the Node with `npm init`. We created our first javascript files, and tested that we can use them together. It is done via export & require:

```
const person : {age: number, name: string} = {  
  name: 'Cat Dog',  
  age: 7,  
}  
  
module.exports = person;
```

```
const person : {...} | {...} = require('./person');  
  
console.log(person);
```

We then went through some of the core modules of node.

path module can be used to modify paths without having to worry about which operating system you are using. It can also be used to get information about the file that is currently being operated.

fs (file system) module can be used to create, delete, read and write to files.

os (operating system) module can be used to gather information about the computer that the script is being run on. You can for example check how much memory you have free, which platform you are on or how many cores the CPU has.

url module is used to handle website urls. For example lets say we have 'https://madeupsite.com:8000/hello.html?id=100&status=active' url. We can easily grab the site (madeupsite.com) or the requested page (hello.html). We can also split the rest of the data to key/value pairs, which in this case would be id=100 and status=active.

events module is used to pass information from place to place. You can basically create a listener, then emit a message somewhere else and all the listeners will get that message.

http module can be used for powerful stuff, like creating a server! It is very low level stuff, so you have to do a lot of stuff manually like setting status response codes.

Next I learned that while you can use `node index.js` to launch that file, you can use `nodemon index.js` to automatically update the live application to latest changes in your code. You can make it even better, if you write this:

```
"scripts": {
  "start": "node index.js",
  "dev": "nodemon index.js"
},
```

... to your package.json, you can run whichever version you want with `npm run xxx!`

After learning the basics of these core modules, we got to work on our first actual website:

```
const server : Server<IncomingMessage, ServerResponse> = http.createServer( request
  if (req.url === '/') {
    res.writeHead(200, { 'Content-Type': 'text/html' }); // 200 = success
    res.end('<h1>Home</h1>');
  }
});
```

... that's it? Yes, that's how simple it is to create a site with node. Of course let's not forget that a lot of magic happens behind those libraries that we use.

We then made our big beautiful website a bit better, by using what we learned before. We used **fs** to load the webpage from our disk:

```

if (req.url === '/') {
  fs.readFile(path.join(__dirname, 'public', 'index.html'), { callback: (err : ErrnoException | null , content
    if (err) throw err;

    res.writeHead(200, { 'Content-Type': 'text/html' }); // 200 = success status response code
    res.end(content);
  });
} else if (req.url === '/about') {

```

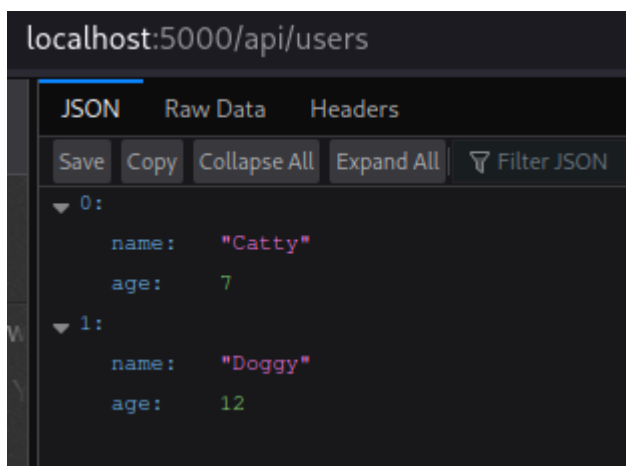
Lastly I learned how to answer to api requests!

```

} else if (req.url === '/api/users') {
  const users : [{name: string, age: number}, ...] = [
    { name: 'Catty', age: 7 },
    { name: 'Doggy', age: 12 }
  ];
  res.writeHead(200, { 'Content-Type': 'application/json' });
  res.end(JSON.stringify(users));
}

```

... that api site result looks like this:



... which looks familiar, I have definitely received and read those before, but never actually sent them myself! Cool stuff.

2. MongoDB

21.05.2023

In this tutorial we learn about MongoDB. It is NoSQL database, which means that the mechanism for data storage and retrieval is other than the tabular relation used in relational databases (so no need to create tables and relations beforehand).

I had some issues trying to install the MongoDB. Apparently they have lately changed their licence, and MongoDB could not been redistributed in official Arch Linux packages. I tried to compile the latest version myself, but there were some compile errors. I decided to just use older version of MongoDB, version 5.0 to be exact (Newest is version 6.0, tutorial is using version 4.0).

After installing MongoDB, I launch it with `systemctl start mongod.service`. As I'm using version 5.0, I can access the mongo shell with command `mongo` (In version 6.0 it is changed to `mongosh`).

I have only used MySQL and SQLite in the past, so it will be interesting how MongoDB compares to them. While watching and following through the tutorial, all the commands that are we did are quite simple. It just takes some time to remember them - gotta have a cheat-sheet of commands nearby. Overall its quite interesting how you chain the commands together.

The chaining of commands works like this: `db.posts.find().limit(10).pretty()`. This will grab all the items in posts database, limit the results to 10 and print them in prettier way.

I'm not sure what else to write in this section, since the video is short and the commands itself are very similar to what I've used with other database languages. What is different is how the data is stored and managed behind the scenes(what Mongo actually does), but that goes out of the scope of this tutorial.

3. Express JS

22.05.2023

Today's topic is Express Js. It is fast, minimalist **server-side** framework for NodeJS. You have a lot of freedom of choice when using Express. As it is NodeJS framework, it also uses javascript as its language.

Before starting any code, we went through some slides and I learned that the req object will have all the data about the site request, and res will contain everything about the response. 'Middleware functions' are functions that have access to these req and res objects.

I downloaded the postman that the tutorial suggested. I quickly checked if there are any better alternatives (just out of habit), and nothing popped out. After deciding to use postman, I downloaded and launched it. What surprised me positively was that postman allowed me to use their application without forcing me to create user and signing in. Not that it would be any requirement to actual usage of the software, but usually companies like to force you to create accounts in order to gather and sell your information.

We started with `npm init` as usual, but this time we used -y flag, and I learned that by using it, the command does not ask you anything and just generates the package.json with default values. We then installed express with `npm i express`.

Next we made the most basic version of server possible:

```
const express = require('express');

const app = express();

app.get('/', (req: Request<P, ResBody, ReqBody, ReqQuery, LocalsObj>, res: Response<ResBody, LocalsObj>) : void => {
  res.send({ body: 'Hello world' });
});

const PORT: string | number = process.env.PORT || 5000;

app.listen(PORT, () => console.log(`Server started on port ${PORT}`));
```

... as you can see, it does not differ that much from our basic node server. But I'm sure we'll get to the goodness of Express soon.

As I guessed, the next thing we did was using static folder for express. I learned that it can be used to serve static files. By using the following:

```
// Set static folder
app.use(express.static(path.join(__dirname, 'public')));
```

... any file we put in that folder, will be automatically loaded when someone requests it. For example if we put about.html inside that folder, then anyone trying to load that about page will be able to. So no need to manually set every possible path to point to those files, express can do that for us.

Next we created our own middleware. Simple showcase of middleware is our first creation:

```
const logger = (req, res, next) : void => {
  console.log('Hello!');
  next();
};

// Init middleware
app.use(logger);
```

... as you can see, straightforward syntax. We create a method that has request, response and next method as parameters. Inside the method we then have access to those, and can do whatever we want to do with them. At the end of the method we need to call the next method on the stack that was given as a parameter (otherwise the request will be left hanging if it does not end the request-response cycle <https://expressjs.com/en/guide/using-middleware.html>).

Next small thing I learned was the moment node package. With that you can easily get current time and date. But after googling and learning more about moment, I decided that I will not use it in my projects, since even moment developers recommend against using it in new projects: "... but we would like to discourage Moment from being used in new projects going forward" (<https://momentjs.com/docs/>). This seems to be mostly because new better alternatives for time and date exists, and moment's file size can get quite large for what it's role is. Which seems to be very true, the NodeJS itself can give you everything you need about times or dates. With `Date().toString()` you can get current time. While researching this, I found out interesting thing about using timings in browser. For example if using firefox, the Date functions current time will be rounded to 2ms. This is because using exact times, someone can fingerprint and identify you based on how long exactly it took to run certain commands (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/now#reduced_time_precision). Interesting stuff!

After that we learned how HTTP GET, POST, PUT and DELETE methods works. Those methods are something that the client (website user) will send to the server. When client sends a GET request, it means that they want some data from server, and server responds accordingly. We made the following middleware handler for that:

```
// Get single member
router.get( {path:('/:id'), handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Response<ResBody, LocalsObj> ) : void => {
  const found : boolean = members.some(member : {email: string, id: number, name: string, status: string} => member.id === parseInt(req.params.id));

  if (found) {
    res.json(members.filter(member : {email: string, id: number, name: string, status: string} => member.id === parseInt(req.params.id)));
  } else {
    res.status(400).json( { body: { msg: `No member with the id of ${req.params.id}` } });
  }
} });
```

... basically if someone sends a GET request with specific url, we will return specific data from our json user list.

POST request means that someone wants to send us some data. For our POST request we made the following:


```

router.post( path: '/', handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Response<ResBody>) => {
  const newMember : {email: string, id: string, name: string, status: string} = {
    id: uuid.v4(),
    name: req.body.name,
    email: req.body.email,
    status: 'active'
  }

  if (!newMember.name || !newMember.email) {
    return res.status( code: 400 ).json( body: { msg: 'No name or email included' } );
  }

  members.push(newMember);

  res.json(members);
});

```

... if someone sends us POST request that contains name and member variables in json format, we will add that user to our users list.

PUT request means that the sender wants to update some specific data. Our PUT request looks like the following:

```

router.put( path:('/:id)', handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Response<ResBody>) => {
  const found : boolean = members.some(member : {email: string, id: string, name: string, status: string} => {
    return member.id === req.params.id;
  });

  if (found) {
    const updateMember : ReqBody = req.body;

    members.forEach(member : {email: string, id: string, name: string, status: string} => {
      if (member.id === parseInt(req.params.id)) {
        member.name = updateMember.name ? updateMember.name : member.name;
        member.email = updateMember.email ? updateMember.email : member.email;

        res.json( body: { msg: "Member updated", member } );
      }
    });
  } else {
    res.status( code: 400 ).json( body: { msg: `No member with the id of ${req.params.id}` } );
  }
});

```

... now if someone sends us PUT request which contains either name or email variable in json format, we will update our system with that data.

Lastly, DELETE request means that the sender wants to delete something. Our handler looked like this:

```
router.delete( path:('/:id'), handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Res
  const found : boolean = members.some(member : {email: string, id: number, name: string, status: string} =>

  if (found) {
    res.json( body: { msg: 'Member deleted', members: members.filter(member : {email: string, id: nu
  } else {
    res.status( code: 400).json( body: { msg: `No member with the id of ${req.params.id}` });
  }
});
```

... we basically just filter through our list excluding the target user.

Next we moved on to rendering stuff server side. We started it by installing node package express-handlebars with `npm i express-handlebars`. After that we created few files and folders similar way as the GitHub page of express-handlebars instructs:

```
├─ app.js
├─ views
│   └─ home.handlebars
│   └─ layouts
│       └─ main.handlebars
```

... after that we simply imported the library and set up the middleware:

```
// Handlebars middleware
app.engine( ext: 'handlebars', expshbs({ defaultLayout: 'main' }));
app.set('view engine', 'handlebars');
```

... but this was not working. If we look at the GitHub page of this part:

app.js:

Creates a super simple Express app which shows the basic way to register a Handlebars view engine using this package.

```
import express from 'express';
import { engine } from 'express-handlebars';

const app = express();

app.engine('handlebars', engine());
app.set('view engine', 'handlebars');
```

... notice that we are actually including **engine**, not the express-handlebars as a whole. So the fix was simple:

```
// Handlebars middleware
app.engine( ext: 'handlebars' exphbs.engine( config: { defaultLayout: 'main' }));
app.set('view engine', 'handlebars');
```

While reading through the GitHub documentation about express-handlebars, there was a good reminder:

Danger 🔥

Never put objects on the `req` object straight in as the data, this can allow hackers to run XSS attacks. Always make sure you are destructuring the values on objects like `req.query` and `req.params`. See

... gotta keep that in mind.

Then I learned how to use variables in views. They are used with double curly-braces:

```
<h1>{{title}}</h1>
```

... I'm sure there are many ways to assign them, but the first way we assigned them was when we created our middleware:

```
app.get('/', (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res
  title: 'Member App'
}));
```

Shortly after that, I learned how to go through every child of an object in our view:

```
{{#each members}}
  <li class="list-group-item">{{this.name}}: {{this.email}}</li>
{{/each}}
```

Lastly we made a quick page with the following content:

Member App

Name

Name

Add Member

Members

Kitty: kitty@email.meow

Doggy: doggy@email.wuf

New Addition: new@addition.com

Second Addition: second@addition.com

Visit API

... with this you can add new members to our users list. The code looked like this:

```

<h1 class="text-center mb-3">{{title}}</h1>

<form action="/api/members" method="POST" class="mb-4">
  <div class="form-group">
    <label for="name">Name</label>
    <input type="text" name="name" class="form-control">
  </div>
  <div class="form-group mb-1">
    <label for="email">Name</label>
    <input type="text" name="email" class="form-control">
  </div>
  <input type="submit" value="Add Member" class="btn btn-primary btn-block">
</form>

<h4>Members</h4>
<ul class="list-group">
  {{#each members}}
    <li class="list-group-item">{{this.name}}: {{this.email}}</li>
  {{/each}}
</ul>

<a href="/api/members" class="btn btn-dark mt-4">Visit API</a>

```

... this was a nice simple example of a way how to use what we learned and created before. Especially that form method="POST" part, that was new for me.

4. Angular

23.05.2023


The angular part begin with downloading a base project. I unzipped it and launched that folder on my WebStorm. We ran the good old `npm install` to install all the dependencies. After that we tested that everything works by building and launching the server with `ng serve`.

I learned that `src/app/app.component.ts` describes our app-root component, which is the top-level component in angular. Component is basic building block of Angular application. It holds information about that particular component, like HTML template, what styles to use and the component's code.

We then learned about the basic folder and file structure of angular app. For example `angular.json` describes the Angular app to the app building tools and `.angular` folder contains files that are required to build the Angular application.

We then created our first own component. I stopped the server, and run `ng generate component Home --standalone --inline-template --skip-tests`, and restarted the server. Then we imported our new component, both as in file (`import { HomeComponent } from './home/home.component';`) and as import to our component (`imports: [HomeComponent],`). We modified our template html. Our first component works!

We then created another component similar like the first one. But this time we added angular interface to it. This interface will define properties of location of a house, like city and state where it is located. To use this, we included that data in our home component:



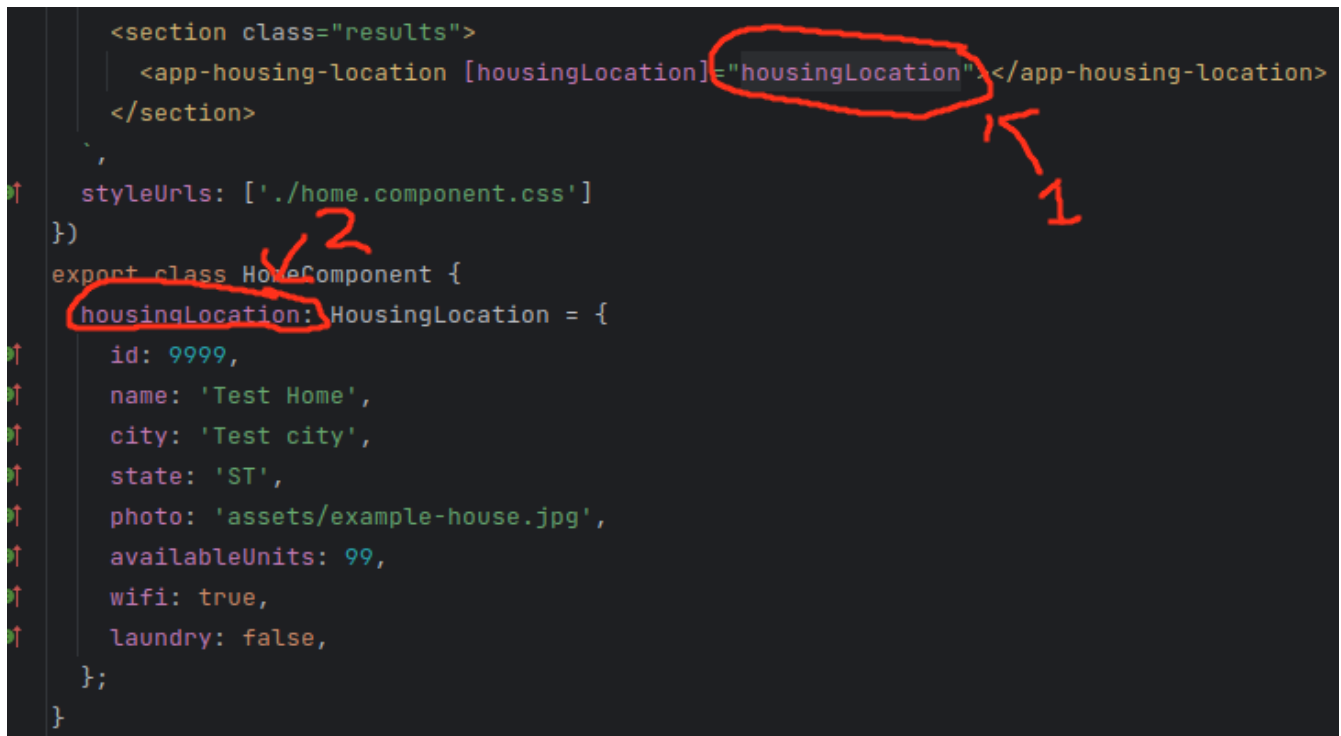
```
export class HomeComponent {
  housingLocation: HousingLocation = {
    id: 9999,
    name: 'Test Home',
    city: 'Test city',
    state: 'ST',
    photo: 'assets/example-house.jpg',
    availableUnits: 99,
    wifi: true,
    laundry: false,
  };
}
```

... if there would be something wrong, our IDE would give us errors at this point.

Next we will learn how to pass data from parent component to child component. This is done via `@Inputs`. To actually use it, you first need to import it from angular core (`import { Input } from '@angular/core';`). Then you need to tell what you want to input. In our case, we want to input `HousingLocation`, so we add `@Input() housingLocation!: HousingLocation;` to our class export (don't forget to import the `HousingLocation` [`import { HousingLocation } from '../housinglocation';`]).

Now it's time to add data bindings so we can send data from our `HomeComponent` to the `HousingLocationComponent`. It was simple as adding `[housingLocation="housingLocation"` to our

app-housing-location in our home.component.ts file like so:

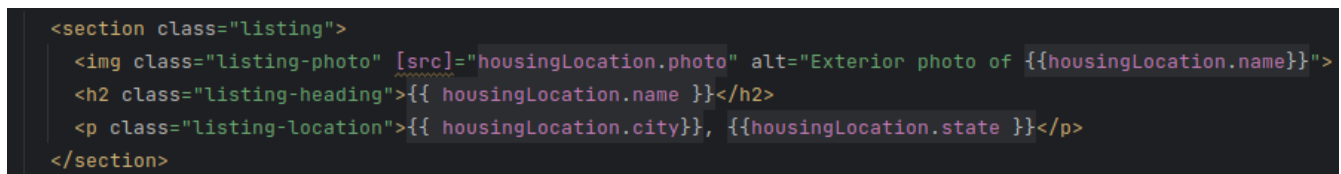


```
<section class="results">
  <app-housing-location [housingLocation]="housingLocation"></app-housing-location>
</section>

styleUrls: ['./home.component.css']
})
export class HomeComponent {
  housingLocation: HousingLocation = {
    id: 9999,
    name: 'Test Home',
    city: 'Test city',
    state: 'ST',
    photo: 'assets/example-house.jpg',
    availableUnits: 99,
    wifi: true,
    laundry: false,
  };
}
```

... the power of IDE's again shine in here, we can make sure that the 1st houseLocation is actually same as the 2nd by control clicking the first one. That brings our caret to the 2nd houseLocation, meaning that they are in fact the same. While doing this, I learned that using the [] inside html means that it will be treated as property from component and not a string value.

Next we added some html code that will show more of the properties that we pass from our parent component:

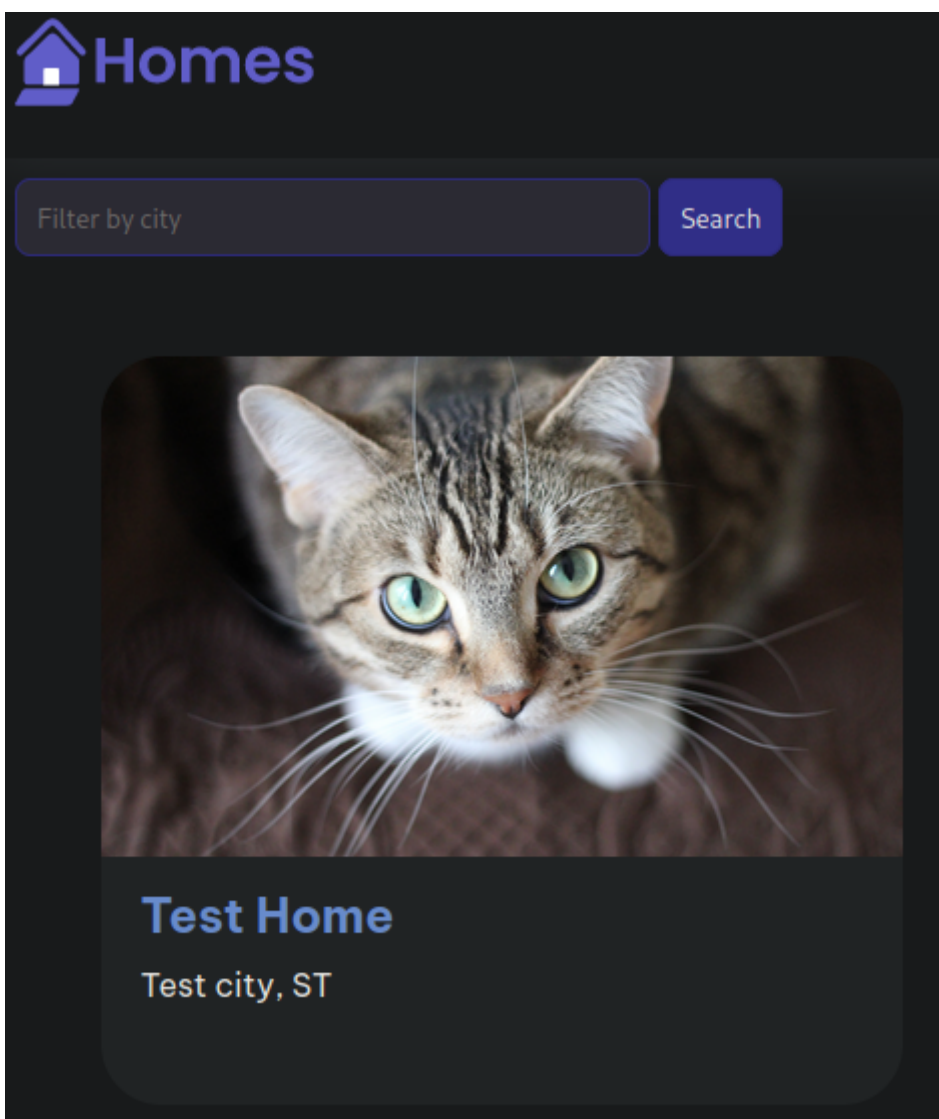


```
<section class="listing">
  <img class="listing-photo" [src]="housingLocation.photo" alt="Exterior photo of {{housingLocation.name}}">
  <h2 class="listing-heading">{{ housingLocation.name }}</h2>
  <p class="listing-location">{{ housingLocation.city}}, {{housingLocation.state }}</p>
</section>
```

... while doing this, I learned that you can access the parameters similar to the Express JS — with double curly-braces. After following the tutorial, my site should have image displaying right now, but it didn't. After few seconds of debugging, I noticed that our photo was pointing to file that were not there:

```
export class HomeComponent {
  housingLocation: HousingLocation = {
    id: 9999,
    name: 'Test Home',
    city: 'Test city',
    state: 'ST',
    photo: 'assets/example-house.jpg',
    availableUnits: 99,
    wifi: true,
    laundry: false,
  };
}
```

... so I fixed that by importing a cat and everything was on track again:



In next section we learn about ngFor. It is basically a for loop so you can loop through arrays for example. We added more houses to be passed on the child component, and made the into array. After that we used the ngFor to loop through all of them:


```

<app-housing-location
  *ngFor="let housingLocation of housingLocationList"
  [housingLocation]="housingLocation">
</app-housing-location>

```

... as you can see, when we use ngFor, you need to prefix it with '*'. Also remember how I told about the control click IDE feature? Well now we use the same variable name, but it will move the caret to the line above, to the 'let houseLocation'! But back to angular again, this ngFor seems like very handy way to repeat html code that you usually have had to manually type, even if you had repeated code.

Now it's time to learn about services. Angular services are some function, value or feature that the application needs. It can be shared so it can be used anywhere within your application. The service can be created with `ng generate service housing --skip-tests`. We then moved our house array list to the new service that we created, and added few methods so we can get all the houses, or some specific house. To actually get the house information from this service to where we need it, we need to import inject from angular core. After that we create an empty array which we will then fill in our constructor, like this:

```

export class HomeComponent {
  housingLocationList: HousingLocation[] = [];
  housingService: HousingService = inject(HousingService);

  no usages
  constructor() {
    this.housingLocationList = this.housingService.getAllHousingLocations();
  }
}

```

... and there we go, we are using a service! That service can be used anywhere, and it could have whatever functions you would like. Very modular.

Next we will learn about routing in angular. Usually routing means that you can move from webpage to webpage. But as angular is single page application, only part of the page will be updated. We started again by generating new component: `ng generate component details --standalone --inline-template --skip-tests`. After that we created routes.ts file, which will contain information about our routes. For now we made 2 different routes:

```
const routeConfig: Routes = [
  {
    path: '',
    component: HomeComponent,
    title: 'Home page'
  },
  {
    path: 'details/:id',
    component: DetailsComponent
    title: 'Home details'
  }
];
```

But to actually implement the routing functionality to our app, we needed to do few things. First our main.ts needed these imports:

```
import { provideRouter } from '@angular/router';
import routeConfig from './app/routes';
```

... and also provide the routes:

```
bootstrapApplication(AppComponent,
  options: {
    providers: [
      provideProtractorTestingSupport(),
      provideRouter(routeConfig)
    ]
  }
);
```

... then our app.component.ts needed this import:

```
import { RouterModule } from '@angular/router';
```

... and of course we needed to include the import in the component metadata:

```
imports: [
  HomeComponent,
  RouterModule
],
```

... that should give the basic routing functionality to our app. Now we only need to actually use the routing:

```

template:
<main>
  <a [routerLink]="['/']">
    <header class="brand-name">
      
    </header>
  </a>
  <section class="content">
    <router-outlet></router-outlet>
  </section>
</main>

```

... as you can see, the first `routerLink="['/']"` initiates navigation to our root page. The `<router-outlet>` is where it will paste the content that our router provides.

Next we learn how we can use parameters with our routing. It will allow including dynamic information as part of our route URL. We added `<a [routerLink]="['/details', housingLocation.id]">Learn More` to our housing-location template. That `routerLink` works kinda like the path we used in NodeJS, it will combine the arguments it gets into a path. In this case it will be `oursite/details/OUR_DYNAMIC_VARIABLE`. We then started working on the `details.component.ts` to get our detail page working. The end result looked like this:

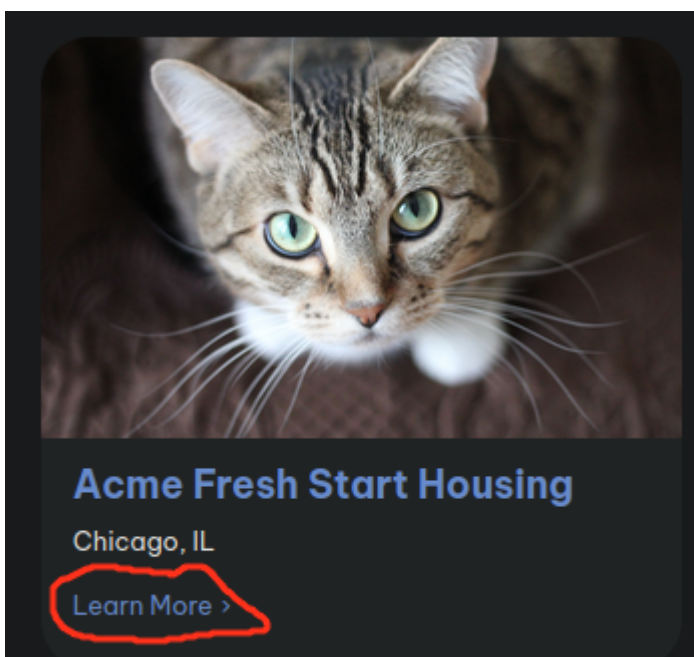
```

import { Component, inject } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ActivatedRoute } from '@angular/router';
import { HousingService } from '../housing.service';
import { HousingLocation } from '../housinglocation';

2 usages
@Component({
  selector: 'app-details',
  standalone: true,
  imports: [CommonModule],
  template: `
    <p>
      details works! {{ housingLocationId }}
    </p>
  `,
  styleUrls: ['./details.component.css']
})
export class DetailsComponent {
  route: ActivatedRoute = inject(ActivatedRoute);
  housingLocationId : number = -1;
  no usages
  constructor() {
    this.housingLocationId = Number(this.route.snapshot.params['id']);
  }
}

```

... as you can see, the template is using our id, which it will get from the router! Lets see it in action, by clicking on the first link:



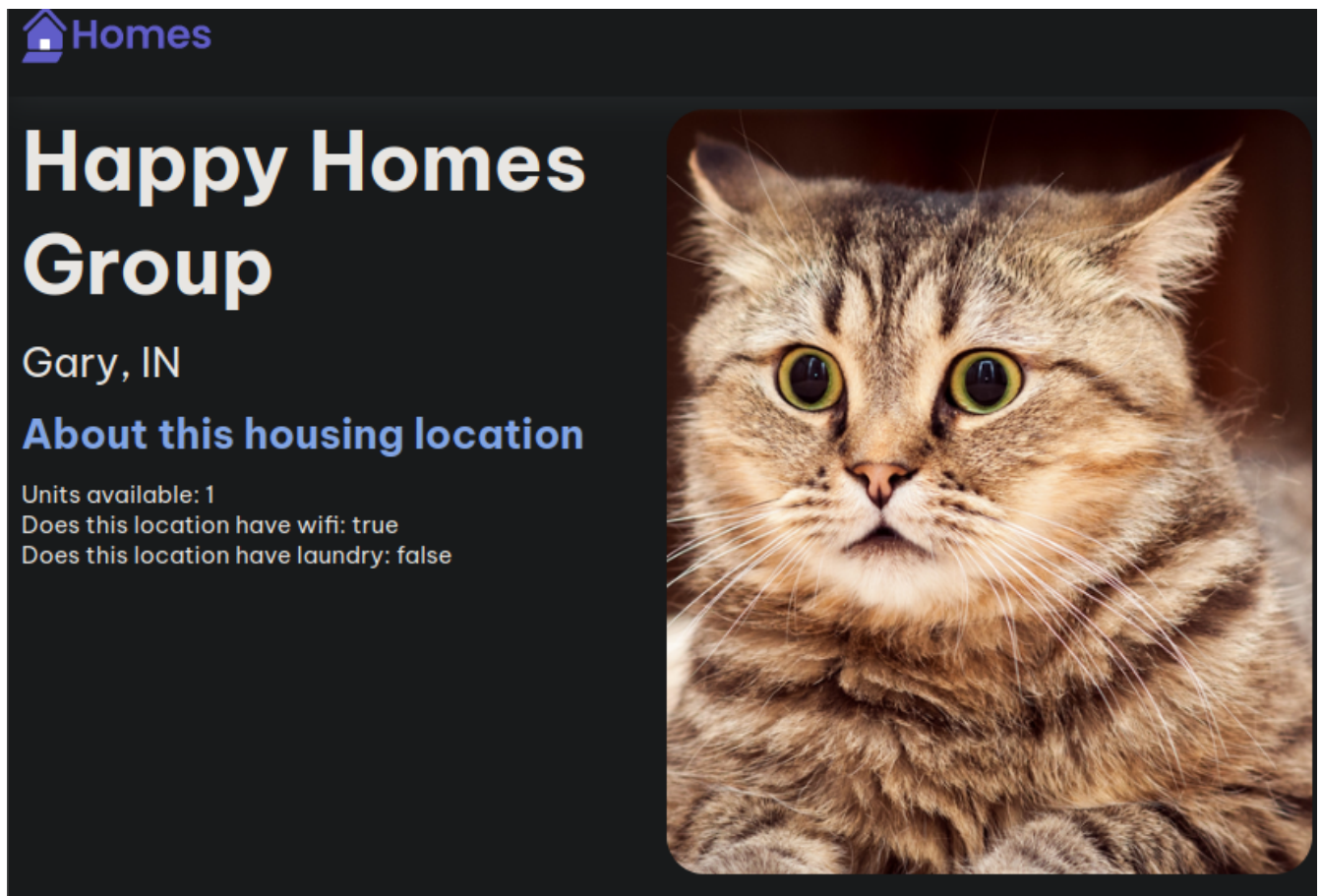
... the result site:

details works! 0

... it works! Let's prove by clicking the second link and see the results:

details works! 1

... it is dynamic! Of course the page is hideous, just a proof of concept. After few modifications, our details page looks beautiful:



Next we will learn about angular forms. With those we can get user input and use it in our application. This seems to be fairly simple. As always, we first import what we need:

```
import { FormControl, FormGroup, ReactiveFormsModule } from '@angular/forms';
```

... then we create new form group and few form controls, these allows us to build forms:

```
applyForm : FormGroup<{firstName: FormCont... = new FormGroup( controls: {  
  firstName: new FormControl( value: '' ),  
  lastName: new FormControl( value: '' ),  
  email: new FormControl( value: '' )  
});
```

... then we add code for our upcoming submit button:

```
submitApplication() : void {
  this.housingService.submitApplication(
    firstName: this.applyForm.value.firstName ?? '',
    lastName: this.applyForm.value.lastName ?? '',
    email: this.applyForm.value.email ?? ''
  );
}
```

... lastly we modify our template to include the form that we want:

```
template: `
<article>
  <img class="listing-photo" [src]="housingLocation?.photo"
    alt="Exterior photo of {{housingLocation?.name}}"/>
  <section class="listing-description">
    <h2 class="listing-heading">{{housingLocation?.name}}</h2>
    <p class="listing-location">{{housingLocation?.city}}, {{housingLocation?.state}}</p>
  </section>
  <section class="listing-features">
    <h2 class="section-heading">About this housing location</h2>
    <ul>
      <li>Units available: {{housingLocation?.availableUnits}}</li>
      <li>Does this location have wifi: {{housingLocation?.wifi}}</li>
      <li>Does this location have laundry: {{housingLocation?.laundry}}</li>
    </ul>
  </section>
  <section class="listing-apply">
    <h2 class="section-heading">Apply now to live here</h2>
    <form [formGroup]="applyForm" (submit)="submitApplication()">
      <label for="first-name">First Name</label>
      <input id="first-name" type="text" formControlName="firstName">

      <label for="last-name">Last Name</label>
      <input id="last-name" type="text" formControlName="lastName">

      <label for="email">Email</label>
      <input id="email" type="email" formControlName="email">
      <button type="submit" class="primary">Apply now</button>
    </form>
  </section>
</article>
`
```

... and that's it! We have working form, that currently outputs the data to browser console whenever the 'Apply now' button is pressed:

FIRST NAME
My Name

LAST NAME
My Last

EMAIL
Email.com

Apply now

```
Homes application received: firstName: My Name, lastName: My Last, email: Email.com.
```

Finally, at the last part of the angular tutorial, we will learn how to include http communication. Our goal is to get data from webserver as json format. We will be using simple json server, which we installed with `npm install json-server`. Next we created file at our project root folder which stores our application data. Then we launched the json-server with `json-server --watch db.json`. This will launch a server at localhost:3000. With the `--watch` tag it will reload itself when we modify the data source file. Next we needed to actually load the content for our website from our newly created server. Currently our data is hardcoded to our angular components. We needed to remove the variables that were holding them, and just load the data from the server when called. As they are loaded from server, we need to use asynchronous functions. If you remember, we created a service that was responsible for getting this data, so we had to modify the methods in there:

```
async getAllHousingLocations(): Promise<HousingLocation[]> {
  const data : Response = await fetch(this.url);
  return await data.json() ?? [];
}

1 usage
async getHousingLocationById(id: number): Promise<HousingLocation | undefined> {
  const data : Response = await fetch( input: `${this.url}/${id}` );
  return await data.json() ?? {};
}
```

... the url of being variable for site <http://localhost:3000/locations>. You can also see that after the url we use the `${id}`, which of course is dynamic, and will result in site content that we actually wanted to see.

And that was it for the angular tutorial. What an awesome tight packet of information!

5. MEAN Stack

24.05.2023

Apparently we are starting our project with creating authentication app, doing the back end first with NodeJS and Express JS. We started by installing many dependencies, including bcryptjs, body-parser, cors, express, jsonwebtoken, mongoose, passport and passport-jwt. We then created a simple server, just like we did in earlier tutorials. The only new thing we did was that we used mongoose to create to MongoDB instance that is running on my system. We did this with:

```
// Connect to database
mongoose.connect(config.database);

// On Connection
mongoose.connection.on('connected', () : void => {
  console.log('Connected to database ' + config.database);
})

// On Error
mongoose.connection.on('error', (err) : void => {
  console.log('Database error: ' + err);
})
```

... the config.database just holds the database location (mongodb://localhost:27017/meanauth), and the secret key we will use to authenticate ourselves.

Next we started creating user system for us. I had some major problems with the following addUser function:

```
module.exports.addUser = function(newUser, callback) : void {
  bcrypt.genSalt( rounds: 10, seed_length: (err : Error , salt : string ) : void => {
    bcrypt.hash(newUser.password, salt, (err, hash) : void => {
      if (err) throw err;

      newUser.password = hash.toString();
      newUser.save(callback);
    });
  });
}
```

... after some frustrating debugging, I found the solution. In the `newUser.save(callback);` you cannot anymore give the callback argument. The tutorial is old enough that is used to work, and even latest comments are 2 years old. Nowhere was any mentions about this not working anymore. But some time later I finally found a way to fix this:


```

module.exports.addUser = function(newUser, callback) : void {
  bcrypt.genSalt( rounds: 10, seed_length: (err : Error , salt : string ) : void => {
    bcrypt.hash(newUser.password, salt, async (err, hash) : Promise<any> => {
      if (err) throw err;

      newUser.password = hash.toString();
      newUser.save();
      return await callback.call();
    });
  });
};

```

... I made the function asynchronous and returned an await call to the callback. Let's hope this fix won't break something in the future.

After adding all of the password management, I was ready for bugs and crashes since I glanced over the comments and saw tons of errors. Well the second error I hit was the following:

```

Error: Login sessions require session support. Did you forget to use `express-session` middleware?

```

... this was simply fixed by installing the `express-session`, importing it: `const session = require('express-session');` and pasting the following before our other passport codes:

```

8 // Passport middleware
9 app.use(session( options: {
10   secret: 'keyboard cat',
11   resave: false,
12   saveUninitialized: true,
13   cookie: { secure: true }
14 }));
15
16 app.use(passport.initialize());
17 app.use(passport.session());

```

... then next error:

```

>MongooseError: Model.findOne() no longer accepts a callback<br>

```

... which I fixed with the same trick that I used last time on this error:

```

module.exports.getUserByUsername = async function(username, callback) : Promise<void> {
  const query : {username: any} = {username: username};
  callback(false, await User.findOne(query));
}

```

... now everything seems to work again .. for now.

Well I speak too soon. After doing the authentication, we got hit by our old friend:

```
>MongooseError: Model.findById() no longer accepts a callback<br>
```

... the fix was changing `User.findById(id, callback);` to `callback(false, User.findById(id));`.

... new error..

```
>TypeError: Converting circular structure to JSON<
```

- a. the problems seems to be that my passport authentication does not contain information about the user, other than the token. There was a fix suggested in moodle page, and also many suggestions in comments, but none of them worked. After hours and hours of debugging, I finally figured out a way to get the data. First I made second version of `getUserById()` function:

```
module.exports.getUserByIdNoCallbacks = async function (id, callback) : Promise<Query<...>> {  
  const query : {_id: any} = {_id: id};  
  return User.findOne(query);  
}
```

... and the following for the router.get:

```
router.get( path: '/profile', passport.authenticate( strategy: 'jwt', options: { session: false, },), async (req :  
  let userID = req.user._conditions._id;  
  
  User.getUserById(userID, callback: async (err, user) : Promise<void> => {  
  
    let workingUser : Query<any, any, unknown, any, "findOne"> = await User.getUserByIdNoCallbacks(userID);  
    res.json( body: {  
      name: workingUser.name,  
      email: workingUser.email,  
      username: workingUser.username  
    });  
  });  
});
```

... I figured out this by checking everything the req parameter contains. I saw that the user ID was the only other data that was related to our target user. Then after many tries on how to actually get the data to variable, I found the `req.user._conditions._id` to work. After that I had to create few asynchronous methods that eventually ends up at my custom version of `getUserByIdNoCallbacks` function. That was quite a journey.

25.05.2023

After all that mess, we finally got the angular part. I had to install it with `npm install @angular/cli` as the other ones did not work. Then we created angular components that we will use (navbar, login, register, home, dashboard and profile). Then we included bootstrap into our project. The video host just included it into his index.html file, but I decided to add it to dependency of my project. That was done by first installing the bootstrap node package (`npm install bootstrap@v5.3.0`). After that I needed to include that to the angular.json build options with the following:

```
"styles": [  
  "node_modules/bootstrap/scss/bootstrap.scss",  
  "src/styles.css"  
],  
"scripts": [  
  "node_modules/bootstrap/dist/js/bootstrap.bundle.min.js"  
]
```

... with that, the bootstrap is ready to be used in my project.

Then we started creating our register page. We created the form which include name, username, password and email. We made validation checks for each value:

```

validateEmail(email : string) : RegExpMatchArray | null {
  return String(email)
    .toLowerCase()
    .match(
      matcher: /^((^[<>()[]\]\\.,;:\s@"]+(\.[^<>()[]\]\\.,;:\s@"])*))$/;
};

```

2 usages

```

validateRegister(user : any) : boolean {

  if (user.name === "") {
    console.log("Missing name");
    return false;
  }

  if (user.username === "") {
    console.log("Missing username");
    return false;
  }

  if (user.email === "") {
    console.log("Missing email");
    return false;
  }

  if (!this.validateEmail(user.email)) {
    console.log("Invalid email");
    return false;
  }

  if (user.password === "") {
    console.log("Missing password");
    return false;
  }

  return true;
}

```

... and now that we have a working validator, we need to inform end user about these validation messages when they happen. The video host used flash messages, which simply did not work for me. So I ended up writing some divs which has angulars ngIf checks whether to show them. The validator now looks like this:

2 usages

```
validateRegister(user : any) : {success: boolean, nameMissing...  {  
  
    let result : {success: boolean, nameMissing...  = {  
        success: true,  
        nameMissing: false,  
        emailMissing: false,  
        emailValid: true,  
        usernameMissing: false,  
        passwordMissing: false  
    }  
  
    if (user.name === "") {  
        result.nameMissing = true;  
        result.success = false;  
    }  
  
    if (user.username === "") {  
        result.usernameMissing = true;  
        result.success = false;  
    }  
  
    if (user.email === "") {  
        result.emailMissing = true;  
        result.success = false;  
    }  
  
    if (!this.validateEmail(user.email)) {  
        result.emailValid = false;  
        result.success = false;  
    }  
  
    if (user.password === "") {  
        result.passwordMissing = true;  
        result.success = false;  
    }  
  
    return result;  
}
```

... here we call the validator and set local variables:

2 usages

```
onRegisterSubmit() : void {
  const user : {name: String, email: String, ...} = {
    name: this.name,
    email: this.email,
    username: this.username,
    password: this.password
  }

  let validateResult : {success: boolean, nameMissing...} = this.validateService.validateRegister(user);

  if (validateResult.success)
  {
    this.authService.registerUser(user).subscribe( observerOrNext: data => {
      if (data.success) {
        console.log('successful register');
        this.router.navigate( commands: ['/login']);
      } else {
        console.log('failed to register');
        this.router.navigate( commands: ['/register']);
      }
    })
  } else {
    this.nameMissing = validateResult.nameMissing;
    this.emailMissing = validateResult.emailMissing;
    this.emailValid = validateResult.emailValid;
    this.usernameMissing = validateResult.usernameMissing;
    this.passwordMissing = validateResult.passwordMissing;
  }
}
```

... and this is how we show the alert divs:

```
<div class="form-group">
  <label for="email">Email</label>
  <input type="email" [(ngModel)]="email" name="email" class="form-control" id="email" aria-describedby="emailHelp" #refEmail="ngModel">
  <small id="emailHelp" class="form-text text-muted">We'll never share your email with anyone else.
  <div class="alert alert-danger" *ngIf="emailMissing">
    This field is required.
  </div>
  <div class="alert alert-danger" *ngIf="!emailMissing && !emailValid">
    Invalid email.
  </div>
</div>

<div class="form-group">
  <label for="password">Password</label>
  <input type="password" [(ngModel)]="password" name="password" class="form-control" id="password" #refPassword="ngModel">
  <div class="alert alert-danger" *ngIf="passwordMissing">
    This field is required.
  </div>
</div>
```

... and the end result looks like this:

Register

Name
Enter Name
This field is required.

Username
my nice username

Email
invalid email
Invalid email.

We'll never share your email with anyone else.

Password
●●●●●●●●

Submit

Then we started working on our authentication. We need to send the user login information to our backend server which in return tells whether the login information is correct. The javascript that the video host wrote did not seem to work, but after some digging and reading comments, the following code ended up working:

```
2 usages
registerUser(user: any): Observable<any> {
  let headers : Headers = new Headers();
  let httpOptions : {headers: HttpHeaders} = { headers: new HttpHeaders( headers: { 'Content-Type': 'application/json' }) };
  return this.httpClient.post<any>(url: 'http://localhost:5000/users/register', user, httpOptions);
}

2 usages
authenticateUser(user: any): Observable<any> {
  let headers : Headers = new Headers();
  let httpOptions : {headers: HttpHeaders} = { headers: new HttpHeaders( headers: { 'Content-Type': 'application/json' }) };
  return this.httpClient.post<any>(url: 'http://localhost:5000/users/authenticate', user, httpOptions);
}
```

... but I ran into an issue. The `angular2-jwt` what the video host was using did not work anymore. After some digging, using `@auth0/angular-jwt` was the choice. To get everything working, I first needed to include it via `import { JwtHelperService } from '@auth0/angular-jwt';`, then setting up the constructor:

```
export class AuthService {  
  
  jwtHelper: any;  
  authToken: any;  
  user: any;  
  
  2 usages  
  constructor(private httpClient:HttpClient) {  
    this.jwtHelper = new JwtHelperService( config: {  
      config: {  
        tokenGetter: this.getToken,  
      },  
    });  
  }  
}
```

... and the some of the methods also needed rework:


```

2 usages
getProfile() : Observable<Object> {
  this.loadToken();
  const headers : HttpHeaders = new HttpHeaders( headers: {
    'Content-Type': 'application/json',
    'Authorization': this.authToken
  });
  return this.httpClient.get( url: 'http://localhost:5000/users/profile', options: {headers: headers})
}

2 usages
storeUserData(token: any, user: any) : void {
  localStorage.setItem('id_token', token);
  localStorage.setItem('user', JSON.stringify(user));
  this.authToken = token;
  this.user = user;
}

1 usage
loadToken() : void {
  const token : string | null = localStorage.getItem( key: 'id_token');
  this.authToken = token;
}

6+ usages
isLoggedIn() : boolean {
  return !this.jwtHelper.isTokenExpired(this.getToken());
}

3 usages
getToken() : string | null {
  const token : string | null = localStorage.getItem( key: 'id_token');
  return token;
}

2 usages
logout() : void {
  this.authToken = null;
  this.user = null;
  localStorage.clear();
}

```

... and there we go, we can create users, authenticate to them, logout and repeat from our website!