INSTITUTO POLITÉCNICO
DO CÁVADO E DO AVE
ESCOLA SUPERIOR DE TECNOLOGIA

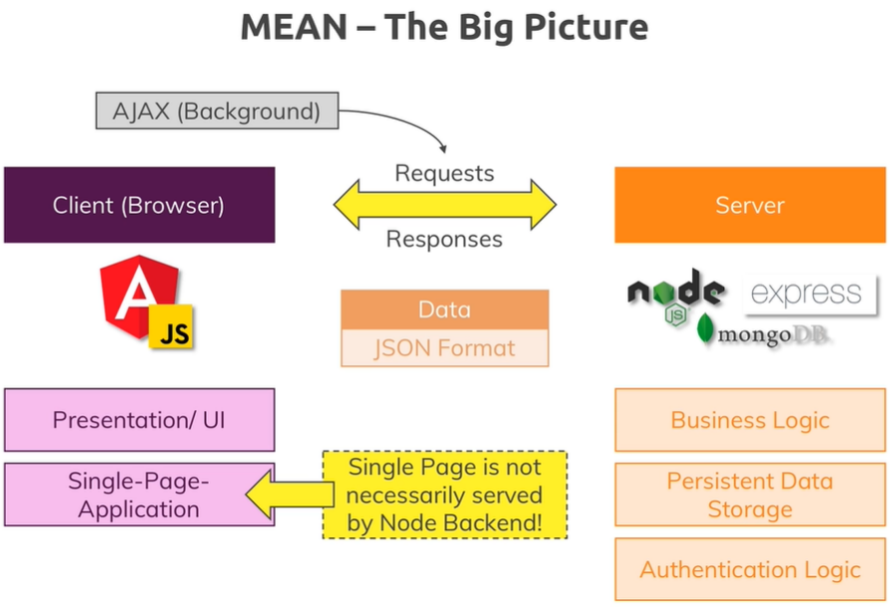
Web Programming

5th Activity - Node and Angular

2020/2021

1. Adding NodeJS

The image below is the big picture of what we want to try to achieve. We already had an introduction to angular and now we will be decing into node and express. Both these technologies are part of the MEAN stack.



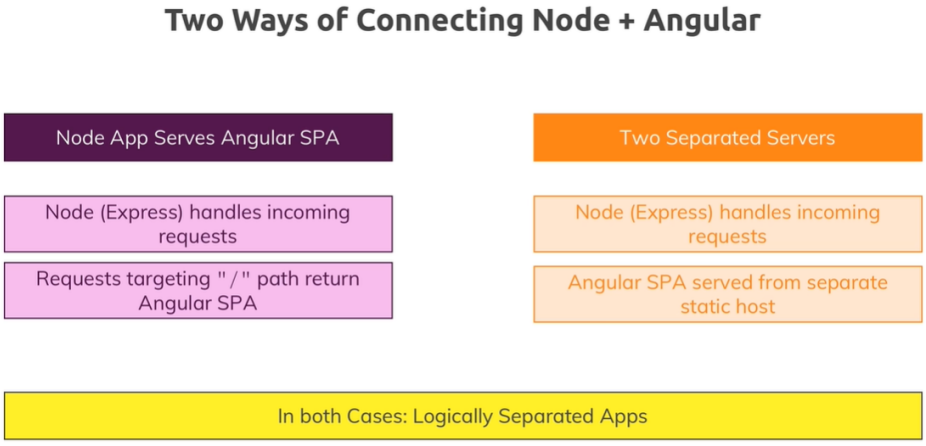
By now we have already used the command ngServe to start our angular application, and behind the scenes, what we have is a node server, but it is a development server, not a production development server. ngServe is nice to develop our angular application but it is not ideal for our node application.

Connecting Node and Angular

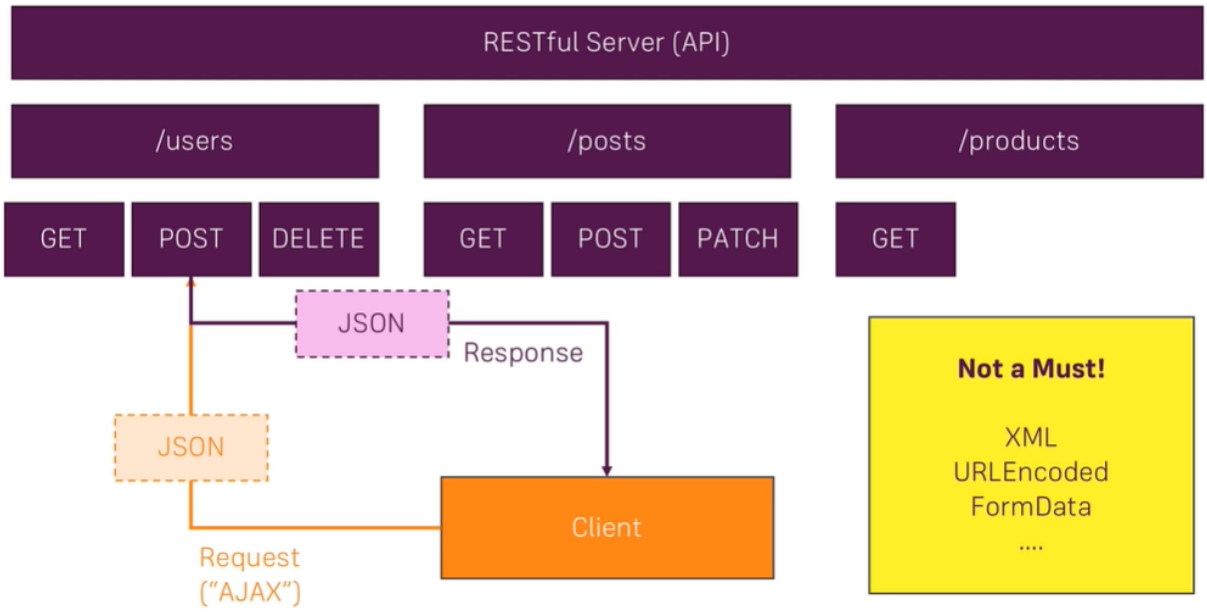
We have two ways of connecting our angular applications. We can have one node application serving both frontend and backend, or have two separate servers serving both frontend and backend separately.

In both cases we have logically separated applications and this will only make any difference when deploying our application to production.

We can refer to the image below to better understand what was just explained.



Our nodeJs backend will handle all data management and we will communicate with it using REST. We will define several verbs that will represent our endpoint and we will send and receive data using JSON. Sending URLEncoded and FormData is also possible but not a must. We will see how to handle those cases.



Adding Node Backend

To start with the next solution we can create a new project or use the existing one. It will be easier to manage all the files and projects in the same solution, so we will use the current solution for that.
Let's create the following file in the root of the solution, and add in the following code.

server.js

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

const server = http.createServer((req, res) => {
    res.end('This is my first response');
});
g
server.listen(3000);
```

Since we are not using **typescript** anymore and we are not using the angular framework, our syntax around imports and language is slightly different.

The server.listen will listen to the defined environment port, which is the one defined in production environments, or it will listen to the port 3000. Locally this will be running on the port 3000.

To start our sever we should run the following command

```
node server.js
```

Now, simply navigate to http://localhost:3000/ and look at the output.

We should take into consideration that for now, our server will not automatically update when we make changes. We will be adding that into this project later on.

Adding Express Framework

Let's now add express, a framework for NodeJs to make NodeJs a little bit easier. To install express we should run the following command

```
npm install --save express
```

Let's create a new **folder** in the root of our application called **backend** with a file called **app.js**

App.js file will hold the express app. Which is still a nodeJS server side app, but taking advantage of express features. The express app is just a big chain of middlewares we apply to the incoming request, like a funnel.
Add the following code to

app.js

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

app.use((req, res, next) => {
    console.log('First middleware');
    next();
});

app.use((req, res, next) => {
    res.send('Hello from express');
});

module.exports = app;
```

server.js

```
const http = require('http');
const app = require('./backend/app.js');


const port = 3000;


app.set('port', port);
const server = http.createServer(app);
```

```
server.listen(port);
```

An improvement we can do is to add nodemon. Nodemon is a powerful development tool which allows the server to rebuild each time we make a change to it, thus, preventing us from stopping it and starting it manually every time.
To install nodemon we should use the following command

Mac: `sudo npm install nodemon -g`
Windows em modo admin: `npm install nodemon -g`

We can now either run nodemon from the terminal or we can create a command for it. To create a command to run nodemon we should go to package.json and under scripts we should add the following

package.json

```
"scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e",
    "start-server":"nodemon server.js"
  },
```

After demonstrating how the basic functionality in the server and app work, we should now progress into having an improved server with error handling and invalid port handling. We should replace the contents of server.js with the following.

To run your node app, try on terminal:
nodemon server.js

Server.js

```
const app = require("./backend/app");
const debug = require("debug")("node-angular");
const http = require("http");

const normalizePort = val => {
  var port = parseInt(val, 10);

  if (isNaN(port)) {
    // named pipe
    return val;
  }

  if (port >= 0) {
    // port number
    return port;
  }

  return false;
};

const onError = error => {
  if (error.syscall !== "listen") {
    throw error;
  }
  const bind = typeof port === "string" ? "pipe " + port : "port " + port;
  switch (error.code) {
    case "EACCES":
      console.error(bind + " requires elevated privileges");
      process.exit(1);
      break;
    case "EADDRINUSE":
      console.error(bind + " is already in use");
      process.exit(1);
      break;
    default:
      throw error;
  }
};

const onListening = () => {
  const addr = server.address();
  const bind = typeof port === "string" ? "pipe " + port : "port " + port;
  debug("Listening on " + bind);
};

const port = normalizePort(process.env.PORT || "3000");
app.set("port", port);

const server = http.createServer(app);
server.on("error", onError);
server.on("listening", onListening);
server.listen(port);
```

Note:
We can configure different ways to initialize our app. For example if we set in package.json a line:
“Start”:”nodemon server.js”
We could start the server just by running
`npm run start-server`

Fetching Posts

We should start by going into our app.js file and adding the following code

`app.js`

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

app.use('/api/posts', (req, res, next) => {

const posts = [
         { id: '23hr23r8', title: 'First server-side post', author: 'catzilla', content: 'This
is coming from the server'},
         { id: 'fd7yfdyf', title: 'Second server-side post', author: 'dogge', content: 'This is
also coming from the server'}
    ];
    res.status(200).json({
        message: 'Post sent with success!',
        posts: posts
    });
});

module.exports = app;
```

If we look carefully into the new code, we can see that we now added a path as a middleware to our app. Now, when we go to that endpoint, we will receive the resource requested.

We can also see that we are assigning a status code, which is optional for this to work, but should always be there. We can return a json object, and it can have any given format, as we can see.

If we save the file and navigate to that end point on the browser, we will now see the json content. http://localhost:3000/api/posts

We should now focus on our angular application. We want to import the http client module to be able to call our api. For that, we should add the following to the file

app.module.ts

```
…
import { HttpClientModule } from '@angular/common/http';
...
imports : [
    …,
    HttpClientModule
]
```

We now want to inject the HttpClient into our post service in the constructor.

posts.service.ts

```
import { Post } from './post.model';
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';
import { HttpClient } from '@angular/common/http';

@Injectable({providedIn: 'root'})
export class PostsService {
    posts: Post[] = [];
    private postsUpdated = new Subject<Post[]>();

    constructor(private httpClient: HttpClient){}

    getPosts() {
        this.httpClient.get<{message:string, posts: Post[]}>('http://localhost:3000/api/posts')
        .subscribe((postData) => {
            this.posts = postData.posts;
            this.postsUpdated.next([...this.posts]);
        });
    }

    getPostUpdateListener() {
        return this.postsUpdated.asObservable();
    }

    addPost(title: string, author: string, content: string) {
        const post: Post = { id: null, title: title, author: author, content: content };
        this.posts.push(post);
        this.postsUpdated.next([...this.posts]);
    }
}
```

this.httpClient.get<data we receive>(url).subscribe(what we do we the data we receive)

post.model.ts

```
export interface Post {
    id: string,
    title: string,
    author: string,
    content: string
}
```

There are actually some more changes in there. We define what type object we want to receive in the get, we are saying that we are receiving a string and an array of posts, which is what we defined in the backend. We are also subscribing to that endpoint and when we receive the results, we set the posts in the service and notify the observable that we have new posts. We also have to define the id to null, since we are not handling it right now.
One last thing we have to do is to go into the post-list component and fix an error that we now have, since the getPosts no longer returns a set of posts.

Post-list.component.ts

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Subscription } from 'rxjs';
import { Post } from '../post.model';
import { PostsService } from '../posts.service';

@Component({
  selector: 'app-post-list',
  templateUrl: './post-list.component.html',
  styleUrls: ['./post-list.component.css']
})
export class PostListComponent implements OnInit, OnDestroy {

  posts: Post[] = [];
  private postsSub: Subscription;

  constructor(public postsService: PostsService){

  }

  ngOnInit(){
    this.postsService.getPosts();
    this.postsSub = this.postsService.getPostUpdateListener().subscribe((posts: Post[]) => {
      this.posts = posts;
    });  }
```

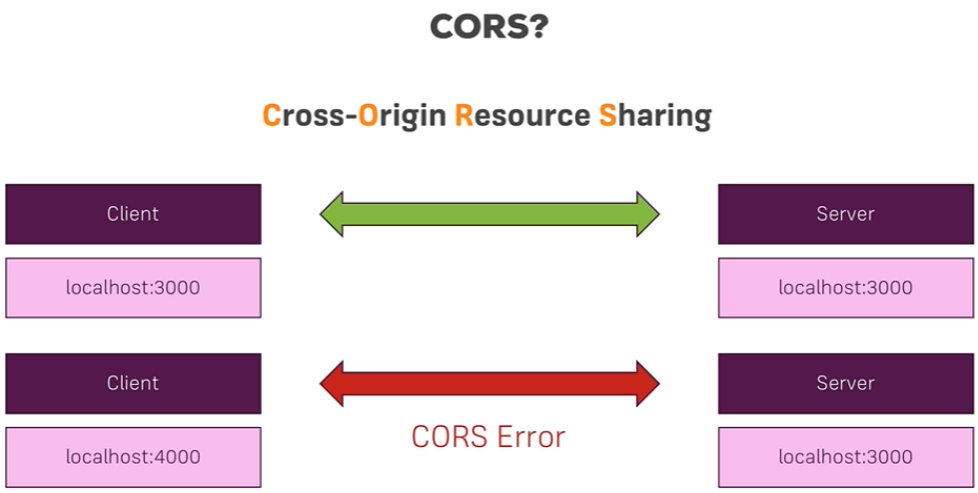
```
  ngOnDestroy(){
    this.postsSub.unsubscribe();
  }

}
```

We can not run both servers, but we will get a CORS error that will be explained in the next chapter as well as how to overcome it.

Understanding CORS

Cross-Origin Resource Sharing is a security mechanism that generates an error when we try to access a resource that is not on the same server and port. The problem is that for our setup and for many modern web applications, we want this. This is not an error, it is a wanted behaviour. Se we will need to disable this behavior by setting the correct headers in our application.



To solve this, we need to add a new middleware before handling our requests. To do that, let's edit our app.js

app.js

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

```
app.use((req, res, next) => {
    res.setHeader("Access-Control-Allow-Origin", "*");
    res.setHeader("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type,
Accept");
    res.setHeader("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, PATCH, OPTIONS");
    next();
});

app.use('/api/posts', (req, res, next) => {
    const posts = [
         { id: '23hr23r8', title: 'First server-side post', author: 'catzilla', content: 'This
is coming from the server'},
         { id: 'fd7yfdyf', title: 'Second server-side post', author: 'dogge', content: 'This is
also coming from the server'}
    ];
    res.status(200).json({
        message: 'Post sent with success!',
        posts: posts
    });
});

module.exports = app;
```

The setHeader method has a key and a value. The first parameter is the key and the second is the value that we want to give to that header.

- The header Access-Control-Allow-Origin with the value star (*) means that no matter which domain the app domain is sending the request, it is allowed to access our resources.
- The header Access-Control-Allow-Header means that we allow headers with the specified headers.
- The header Access-Control-Allow-Methods defines which http verbs we allow to send requests.

We can now run both servers and see that we are retrieving the list of posts.

Adding Posts

Body Parser

Adding posts has the same logic, but now we need to fetch data from de FORM, and node has a package - bodyparser - that helps us to grab data sent from a FORM. To use it we need to install it

npm install --save body-parser

And use it as a middleware to parse data from forms:

// parse application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: false }))

and json
// parse application/json
app.use(bodyParser.json())

Now we'll use **req.body** to access the data that came from the http request sent by angular.

app.js

```
const express = require('express');
const app = express();
const bodyParser = require("body-parser");

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));


app.use((req, res, next) => {
    res.setHeader("Access-Control-Allow-Origin", "*");
    res.setHeader("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type,
Accept");
    res.setHeader("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, PATCH, OPTIONS");
    next();
});


app.post("/api/posts", (req, res, next) => {
  const post = req.body;
  console.log(post);
  res.status(201).json({
    message: "Post added",
  });
```

```
});

app.get('/api/posts', (req, res, next) => {
    const posts = [
         { id: '23hr23r8', title: 'First server-side post', author: 'catzilla', content: 'This
is coming from the server'},
         { id: 'fd7yfdyf', title: 'Second server-side post', author: 'dogge', content: 'This is
also coming from the server'}
    ];
    res.status(200).json({
        message: 'Post sent with success!',
        posts: posts
    });
});

module.exports = app;
```

Returns middleware that only parses json and only looks at requests where the Content-Type header matches the type option. This parser accepts any Unicode encoding of the body and supports automatic inflation of gzip and deflate encodings.
Returns middleware that only parses urlencoded bodies and only looks at requests where the Content-Type header matches the type option. This parser accepts only UTF-8 encoding of the body and supports automatic inflation of gzip and deflate encodings.

Post Http Request

post.service.ts

```
import { Post } from "./post.model";
import { Injectable } from "@angular/core";
import { Subject } from "rxjs";
import { HttpClient } from "@angular/common/http";

@Injectable({ providedIn: "root" })

export class PostsService {
  private posts: Post[] = [];
  private postsUpdated = new Subject<Post[]>();

  constructor(private httpClient: HttpClient) {}

  getPosts() {
    this.httpClient
      .get<{ message: string; posts: Post[] }>(
        "http://localhost:3000/api/posts"
      )
      .subscribe((postData) => {
        console.log(postData);
```

```
        this.posts = postData.posts;
        this.postsUpdated.next([...this.posts]);
      });
  }

  getPostUpdateListener() {
    return this.postsUpdated.asObservable();
  }

  addPost(title: string, content: string) {
    const post: Post = {
      id: null,
      title: title,
      content: content,
    };

    this.httpClient
      .post<{ message: string }>("http://localhost:3000/api/posts", post)
      .subscribe((responseData) => {
        console.log(responseData.message);
        this.posts.push(post);
        this.postsUpdated.next([...this.posts]);
      });
  }
}
```

We perform the post, subscribe to the answer, and when we are obtaining the answer we add to the post array and notify every subscriber of the postUpdated observable we have an update.

The post is similar to the get, the basic syntax is:

this.http.post<type of message to send>(url, postdata).subscribe(what to do with the data)