# Module Introduction

## 

## In this module we will optimize the code both frontend and backend by applying some important concepts and below steps

### Using Controllers

### Separating Middleware

### Creating Angular Material Module

### Splitting the app into feature module(s)

### Fixing an Update bug

### Creating the Auth Module

### Adding Lazy Loading

### Fixing the Auth Guard

### Using the Global Angular Config

### Using the node Environment Variables

# Using Controllers (Server/backend optimization)

## THEORY: What we are trying to do

### In this section we will be optimizing the backend server by refactoring the code under **routes folder** and introducing a new layer called controller layer

### Under folder **backend/routes** we have 2 files to handle routing logic for all APIs

#### posts.js

#### user.js

### These files have 2 responsibilities combined

### Handling routing

### And also handle/process **requests received by routes**

### Let’s refactor the code and move the request handling logic to a separate layer(files) called controller layer by adding controllers for each of the respective routes

## PRACTICLE: Create User controller with signup and login functionality

### Create **“controller”** folder and **“user.js”** file under **backend** folder as shown below.

### 

### Copy below code into controllers/user.js. This code handles the singup functionality in route/user.js.

exports.createUser = (req, res, next) => {

  bcrypt.hash(req.body.password, 10).then((hash) => {

    const user = new User({

      email: req.body.email,

      password: hash,

    });

    user

      .save()

      .then((result) => {

        res.status(201).json({

          message: "User created!",

          result: result,

        });

      })

      .catch((err) => {

        res.status(500).json({

          message: "Invalid authentication credentials!",

        });

      });

  });

};

### Copy below code into controllers/user.js. This code handles the login functionality in route/user.js.

exports.userLogin = (req, res, next) => {

  let fetchedUser;

  User.findOne({ email: req.body.email })

    .then(user => {

      if (!user) {

        return res.status(401).json({

          message: "Auth failed"

        });

      }

      fetchedUser = user;

      return bcrypt.compare(req.body.password, user.password);

    })

    .then(result => {

      if (!result) {

        return res.status(401).json({

          message: "Auth failed"

        });

      }

      const token = jwt.sign(

        { email: fetchedUser.email, userId: fetchedUser.\_id },

        "secret\_this\_should\_be\_longer",

        { expiresIn: "1h" }

      );

      res.status(200).json({

        token: token,

        expiresIn: 3600,

        userId: fetchedUser.\_id

      });

    })

    .catch(err => {

      return res.status(401).json({

        message: "Invalid authentication credentials!"

      });

    });

}

### Move the below required import statements from route/user.js file into controllers/user.js file. Past the below code at the beginning for the file controllers/user.js

const bcrypt = require("bcrypt");

const jwt = require("jsonwebtoken");

const User = require("../models/user");

## PRACTICLE: Modify route/user.js and Observe the changes mentioned below

### Importing the controller

### Mapping each route for one of the controller functions

### Now routing file looks simple with only route logic

const express = require("express");

const UserController = require("../controllers/user");

const router = express.Router();

router.post("/signup", UserController.createUser);

router.post("/login", UserController.userLogin);

module.exports = router;

### Old code for reference

### 

## RUN THE APP: Verify the below functionality (sign-up, login)

### Login

### 

### 

### Sing-up

### 

## PRACTICLE: Create post controller with post API request handlers

### Create a file posts.js under controller folder as shown below

### 

### There are 5 individual routes in routes/posts.js with respective request handler functions. Move the request handler functions one by one into controllers/posts.js created in above step.

#### Create Post

#### Update Post

#### Get Posts

#### Get Post (Single Post)

#### Delete a Post

### Move create post by pasting below code into controllers/posts.js

exports.createPost = (req, res, next) => {

  const url = req.protocol + "://" + req.get("host");

  const post = new Post({

    title: req.body.title,

    content: req.body.content,

    imagePath: url + "/images/" + req.file.filename,

    creator: req.userData.userId

  });

  post

    .save()

    .then(createdPost => {

      res.status(201).json({

        message: "Post added successfully",

        post: {

          ...createdPost,

          id: createdPost.\_id

        }

      });

    })

    .catch(error => {

      res.status(500).json({

        message: "Creating a post failed!"

      });

    });

};

### Move Update post by pasting below code into controllers/posts.js

exports.updatePost = (req, res, next) => {

  let imagePath = req.body.imagePath;

  if (req.file) {

    const url = req.protocol + "://" + req.get("host");

    imagePath = url + "/images/" + req.file.filename;

  }

  const post = new Post({

    \_id: req.body.id,

    title: req.body.title,

    content: req.body.content,

    imagePath: imagePath,

    creator: req.userData.userId

  });

  Post.updateOne({ \_id: req.params.id, creator: req.userData.userId }, post)

    .then(result => {

      if (result.nModified > 0) {

        res.status(200).json({ message: "Update successful!" });

      } else {

        res.status(401).json({ message: "Not authorized!" });

      }

    })

    .catch(error => {

      res.status(500).json({

        message: "Couldn't udpate post!"

      });

    });

};

### Move Get Posts by pasting below code into controllers/posts.js

exports.getPosts = (req, res, next) => {

  const pageSize = +req.query.pagesize;

  const currentPage = +req.query.page;

  const postQuery = Post.find();

  let fetchedPosts;

  if (pageSize && currentPage) {

    postQuery.skip(pageSize \* (currentPage - 1)).limit(pageSize);

  }

  postQuery

    .then(documents => {

      fetchedPosts = documents;

      return Post.count();

    })

    .then(count => {

      res.status(200).json({

        message: "Posts fetched successfully!",

        posts: fetchedPosts,

        maxPosts: count

      });

    })

    .catch(error => {

      res.status(500).json({

        message: "Fetching posts failed!"

      });

    });

};

### Move Get Post (Single Post) by pasting below code into controllers/posts.js

exports.getPost = (req, res, next) => {

  Post.findById(req.params.id)

    .then(post => {

      if (post) {

        res.status(200).json(post);

      } else {

        res.status(404).json({ message: "Post not found!" });

      }

    })

    .catch(error => {

      res.status(500).json({

        message: "Fetching post failed!"

      });

    });

};

### Move Delete a Post by pasting below code into controllers/posts.js

exports.deletePost = (req, res, next) => {

  Post.deleteOne({ \_id: req.params.id, creator: req.userData.userId })

    .then(result => {

      console.log(result);

      if (result.n > 0) {

        res.status(200).json({ message: "Deletion successful!" });

      } else {

        res.status(401).json({ message: "Not authorized!" });

      }

    })

    .catch(error => {

      res.status(500).json({

        message: "Deleting posts failed!"

      });

    });

};

### Import required model by pasting below code at the beginning for the file controllers/posts.js

const Post = require("../models/post");

## PRACTICLE: Modify route/posts.js and observe the changes mentioned below

### Import post controller at the beginning

const PostController = require("../controllers/posts");

### Modify the below routes to use the newly imported controller

#### Create Post route modified

#### Update Post route modified

#### Get Posts route modified

#### Get Post (Single Post) route modified

#### Delete a Post route modified

router.post(

  "",

  checkAuth,

  multer({ storage: storage }).single("image"),

  PostController.createPost

);

router.post(

  "",

  checkAuth,

  multer({ storage: storage }).single("image"),

  PostController.createPost

);

router.get("", PostController.getPosts);

router.get("/:id", PostController.getPost);

router.delete("/:id", checkAuth, PostController.deletePost);

## RUN THE APP: Verify below functionality

### Create Post

### Update Post

### Get Posts

### Get Post (Single Post)

### Delete a Post

# Separating the middleware

## THEORY: Observe the routes/posts.js and route/user.js,

### It looks much cleaner compared to earlier as request handler functions are moved to controller. There is still scope for further refactor

### Multer logic to handle multi part file upload can be moved a separate middleware

## PRACTICLE: Create a new middleware to include multer logic into a separate file

### Create a new file called file.js under middleware folder as shown below. Observer the code (highlighted as 1,2,3,4) which needs to be included in file.js

### 

### Code to paste into file.js

const multer = require("multer");

const MIME\_TYPE\_MAP = {

  "image/png": "png",

  "image/jpeg": "jpg",

  "image/jpg": "jpg"

};

const storage = multer.diskStorage({

  destination: (req, file, cb) => {

    const isValid = MIME\_TYPE\_MAP[file.mimetype];

    let error = new Error("Invalid mime type");

    if (isValid) {

      error = null;

    }

    cb(error, "backend/images");

  },

  filename: (req, file, cb) => {

    const name = file.originalname

      .toLowerCase()

      .split(" ")

      .join("-");

    const ext = MIME\_TYPE\_MAP[file.mimetype];

    cb(null, name + "-" + Date.now() + "." + ext);

  }

});

module.exports = multer({ storage: storage }).single("image");

## PRACTICLE: Import the middleware into route/posts.js

### observe code changes as shown in below screenshot

### 

### Code to paste in posts.js

const express = require("express");

const multer = require("multer");

const PostController = require("../controllers/posts");

const Post = require("../models/post");

const checkAuth = require("../middleware/check-auth");

const extractFile = require("../middleware/file");

const router = express.Router();

router.post("", checkAuth, extractFile, PostController.createPost);

router.put("/:id", checkAuth, extractFile, PostController.updatePost);

router.get("", PostController.getPosts);

router.get("/:id", PostController.getPost);

router.delete("/:id", checkAuth, PostController.deletePost);

module.exports = router;

# Creating Angular material module (FRONT END OPTIMIZATION)

## THEORY: What we are trying to do

### There is a scope to improve the code in app.module.ts from coding perspective as well as performance

### From coding perspective, we can move related modules into one single sub module

### From performance perspective

#### So far, we have only one module in the current project

#### In a large project we can not have one single module. We need to have many logical modules so that we can load them selectively or whenever required.

#### Example: In our project we can have separate modules for login/signup feature and separate module for posting data

## THEORY: Creating separate modules for all angular material modules

### In app.module.ts we have a list of angular material modules imported from angular materials as shown below

### 

### The above highlighted modules will be moved into a separate new sub module angular-material.module.ts

### The newly created module will be included in the main app.module.ts

## PRACTICLE: Creating angular material module

### Create a file angular-material.module.ts under root folder as shown in below screenshot

### 

### NOTE: Observe the @NgModule

### Paste the below code into angular-material.module.ts and remove them from app.module.ts

import { NgModule } from "@angular/core";

import { MatInputModule } from "@angular//material/input";

import { MatCardModule } from "@angular/material/card";

import { MatButtonModule } from "@angular/material/button";

import { MatToolbarModule } from "@angular/material/toolbar";

import { MatExpansionModule } from "@angular/material/expansion";

import { MatProgressSpinnerModule } from "@angular/material/progress-spinner";

import { MatPaginatorModule } from "@angular/material/paginator";

import { MatDialogModule } from "@angular/material/dialog";

@NgModule({

  imports: [

    MatInputModule,

    MatCardModule,

    MatButtonModule,

    MatToolbarModule,

    MatExpansionModule,

    MatProgressSpinnerModule,

    MatPaginatorModule,

    MatDialogModule,

  ],

  exports: [

    MatInputModule,

    MatCardModule,

    MatButtonModule,

    MatToolbarModule,

    MatExpansionModule,

    MatProgressSpinnerModule,

    MatPaginatorModule,

    MatDialogModule,

  ],

})

export class AngularMaterialModule {}

### Import / export repeats the same list and that can be optimized using just export because of angular optimization technique. The below code can be removed from file angular-material.module.ts

  imports: [

    MatInputModule,

    MatCardModule,

    MatButtonModule,

    MatToolbarModule,

    MatExpansionModule,

    MatProgressSpinnerModule,

    MatPaginatorModule,

    MatDialogModule,

  ],

## PRACTICLE: Modify app.module.ts to include newly created sub module

### Remove the material module reference from app.module.ts and Include the newly created module into main module(app.module.ts) with below code changes ( highlighted)

### 

### Code not provided, modify as highlighted in the above screenshot

## RUN THE APP: After the above code change verify the application functionality once

# Splitting the app into feature module(s)

## THEORY: What we are trying to do

### Similar to material module create in previous section lets create post module

### We have below 2 components in app.module.ts related to POST functionality that can be moved to a separate module called postmodule

#### PostCreateComponent

#### PostListComponent

#### 

### The above highlighted components along with respective imports will be moved into a separate new sub module posts.module.ts

### The newly created post module will be included in the main app.module.ts

## PRACTICLE: Creating POST module

### Create a file posts.module.ts under post folder as shown in below screenshot

### 

### Move the 2 components to new file by pasting the below code into posts.module.ts and also remove them from app.module.ts

import { NgModule } from "@angular/core";

import { PostCreateComponent } from "./post-create/post-create.component";

import { PostListComponent } from "./post-list/post-list.component";

@NgModule({

  declarations: [PostCreateComponent, PostListComponent],

  imports: [],

})

export class PostsModule {}

### Observe the below points

#### The @NgModule and the skeleton code to create any module

#### Declaration section declaring the components

### As post component uses reactive forms component, move ReactiveFormsModule from app.module.ts to posts.module.ts. Observe the highlighted code in the imports

#### Paste the below code in posts.module.ts to make the change

import { NgModule } from "@angular/core";

import { PostCreateComponent } from "./post-create/post-create.component";

import { PostListComponent } from "./post-list/post-list.component";

import { ReactiveFormsModule } from "@angular/forms";

@NgModule({

  declarations: [PostCreateComponent, PostListComponent],

  imports: [ReactiveFormsModule],

})

export class PostsModule {}

## PRACTICLE: Modify app.module.ts to include newly created sub module

### Remove the POST components from app.module.ts and Include the newly created module into main module(app.module.ts) with below code changes ( highlighted)

### 

### Code not provided, modify as highlighted in the above screenshot

## RUN THE APP: Error when adding a new POST

### After running the app and trying to add a new post we get below error when we run the app and

### 

### There are 3 errors

### Error 1 is because of angular material modules dependency required to post forms

### Error 2 and 3 are because of dependency on default angular feature to recognize ngIf syntax which is not enabled by default.

### ngIf kind of syntax is enable in root module app.module.ts because of importing BrowserModule

### we can get ngIf enabled in post module by including another module called CommonModule

### Also, the post create component (form) routes to post list component after saving posts using routerLink which will work only if we include RouterModule

## PRACTICLE: Include AngularMaterialModule, CommonModule, RouterModule in post module

### As post components require material modules to run the post forms, include already created angular-material.module in posts.module.ts

### Include CommonModule and RouterModule as highlighted below for the reasons stated in the previous step

import { NgModule } from "@angular/core";

import { ReactiveFormsModule } from "@angular/forms";

import { CommonModule } from "@angular/common";

import { RouterModule } from "@angular/router";

import { PostCreateComponent } from "./post-create/post-create.component";

import { PostListComponent } from "./post-list/post-list.component";

import { AngularMaterialModule } from "../angular-material.module";

@NgModule({

  declarations: [PostCreateComponent, PostListComponent],

  imports: [

    CommonModule,

    ReactiveFormsModule,

    AngularMaterialModule,

    RouterModule

  ]

})

export class PostsModule {}

## RUN THE APP: We should able to add a new POST successfully

### 

# Fixing an Update Bug

## RUN THE APP: Edit a post and see below UnAuthorized error

### 

### 

### 

### The error is because of the below code which we need to fix - Explain the code

### 

## THEORY: WHY THE BUG

### In the above code result.nModified will be greater than zero if the DB object was modified upon saving the post.

### When we save an existing post without modifying it the above value will not be greater than zero

### In order to fix the issue change result.nModified to result.n

## PRACTICLE: Fix the issue with below code changes

### fix the issue change result.nModified to result.n

## RUN THE APP: Edit post should SUCCEED

# Creating Auth Module

## THEORY: What we are trying to do

### Similar to other feature module creation in previous section lets create auth module to move authentication features into a separate sub module

### We have below 2 components in app.module.ts related to Authentication functionality that can be moved to a separate module called AuthModule

#### LoginComponent

#### SignupComponent

#### 

### The above highlighted components along with respective imports will be moved into a separate new sub module auth.module.ts

### The newly created auth module will be included in the main app.module.ts

## PRACTICLE: Creating Auth module

### Create a file auth.module.ts under auth folder as shown in below screenshot

### 

### Move the 2 components to the new file by pasting the below code into auth.module.ts and also remove them from app.module.ts

import { NgModule } from "@angular/core";

import { CommonModule } from "@angular/common";

import { FormsModule } from "@angular/forms";

import { LoginComponent } from "./login/login.component";

import { SignupComponent } from "./signup/signup.component";

import { AngularMaterialModule } from "../angular-material.module";

@NgModule({

  declarations: [LoginComponent, SignupComponent],

  imports: [CommonModule, AngularMaterialModule, FormsModule],

})

export class AuthModule {}

### Observe the below points

#### The @NgModule and the skeleton code to create any module

#### Declaration section declaring the components

#### As auth component uses forms component it should import the dependent module called FormsModule

#### As auth component uses angular material components it should import the dependent AngularMaterialModule

## PRACTICLE: Modify app.module.ts to include newly created sub module

### Remove the authentication related components from app.module.ts and Include the newly created auth module into main module(app.module.ts) with below code changes ( highlighted)

### 

### Code not provided, modify as highlighted in the above screenshot

## RUN THE APP: Verify the signup and login functionality

# Adding Lazy Loading

### Splitting the modules into multiple smaller modules is good, now our root module which app.module.ts looks leaner.

### lets use this for performance benefits by loading the modules on demand using lazy loading technique provided by angular

### Provide an example of lazy loading technique and explain the theory ( NOTE : opposite of lazy loading is eager loading which is by default)

### lets decide to implement Lazy loading of user.module.ts with below steps

### If we observe app-routing.module.ts (this is the only routing module for entire app) we have 2 routes for components inside user module.

### 

### As we want to lazy load a user module, let’s move routes to login and signup components inside the user module to another child routing module called auth-routing.module

### Create a file as shown below and paste the provided code

### 

### Code to be pasted in auth-routing.module.ts

import { NgModule } from "@angular/core";

import { Routes, RouterModule } from "@angular/router";

import { LoginComponent } from "./login/login.component";

import { SignupComponent } from "./signup/signup.component";

const routes: Routes = [

  { path: "login", component: LoginComponent },

  { path: "signup", component: SignupComponent },

];

@NgModule({

  imports: [RouterModule.forChild(routes)],

  exports: [RouterModule],

})

export class AuthRoutingModule {}

### observe we are using RouterModule.forChild() , this is different from RouterModule.forRoot() function in the main routing module angular-material.module.ts

### Eventually this child route will be merged with main route.

### Next step is to include the new created auth-routing.module.ts into auth.module.ts not into main route or main module

### Update auth-module.ts with highlighted code below

import { NgModule } from "@angular/core";

import { CommonModule } from "@angular/common";

import { FormsModule } from "@angular/forms";

import { LoginComponent } from "./login/login.component";

import { SignupComponent } from "./signup/signup.component";

import { AngularMaterialModule } from "../angular-material.module";

import { AuthRoutingModule } from "./auth-routing.module";

@NgModule({

  declarations: [LoginComponent, SignupComponent],

  imports: [CommonModule, AngularMaterialModule, FormsModule, AuthRoutingModule]

})

export class AuthModule {}

### next step is to connect root router into child router using auth module

### make below code changes to app-routing.module.ts

### 

### Code provide below for app-routing.module.ts

import { NgModule } from "@angular/core";

import { RouterModule, Routes } from "@angular/router";

import { PostListComponent } from "./posts/post-list/post-list.component";

import { PostCreateComponent } from "./posts/post-create/post-create.component";

import { AuthGuard } from "./auth/auth.guard";

import { AuthModule } from "./auth/auth.module";

const routes: Routes = [

  { path: "", component: PostListComponent },

  { path: "create", component: PostCreateComponent, canActivate: [AuthGuard] },

  {

    path: "edit/:postId",

    component: PostCreateComponent,

    canActivate: [AuthGuard],

  },

  {

    path: "auth",

    loadChildren: () => import("./auth/auth.module").then((m) => m.AuthModule),

  },

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule],

  providers: [AuthGuard],

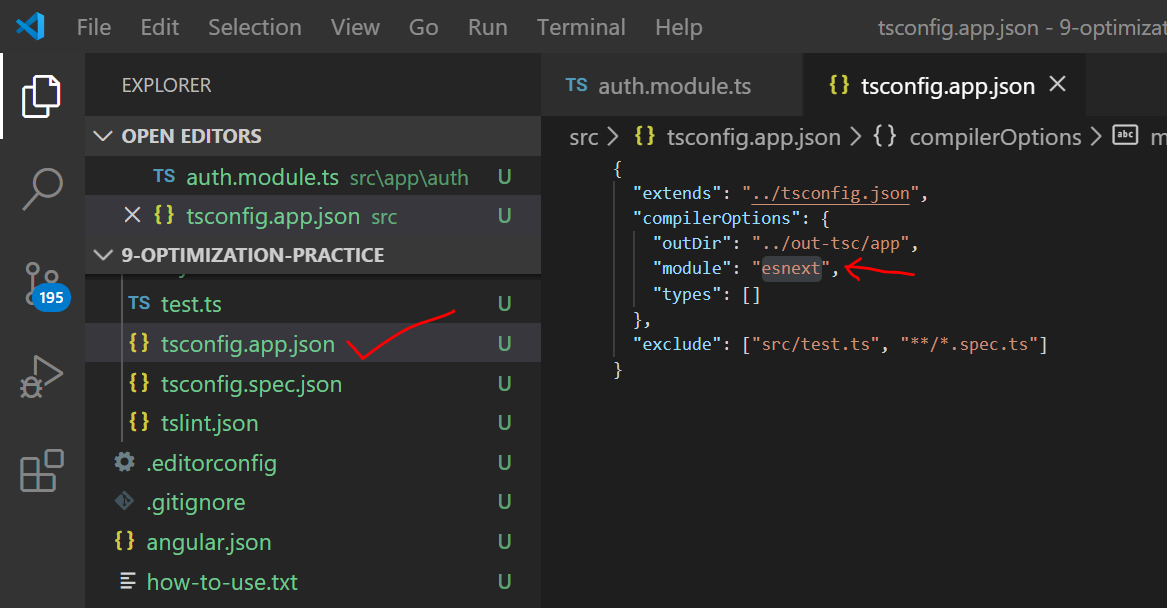
})

export class AppRoutingModule {}

### After making above changes we might see an issue like below

### 

### This can be resolved by making below change in tsconfig.app.json



### 3 more change is required, in app.module.ts we have directly include UserModule, lets remove that

### remove the highlighted code

### 

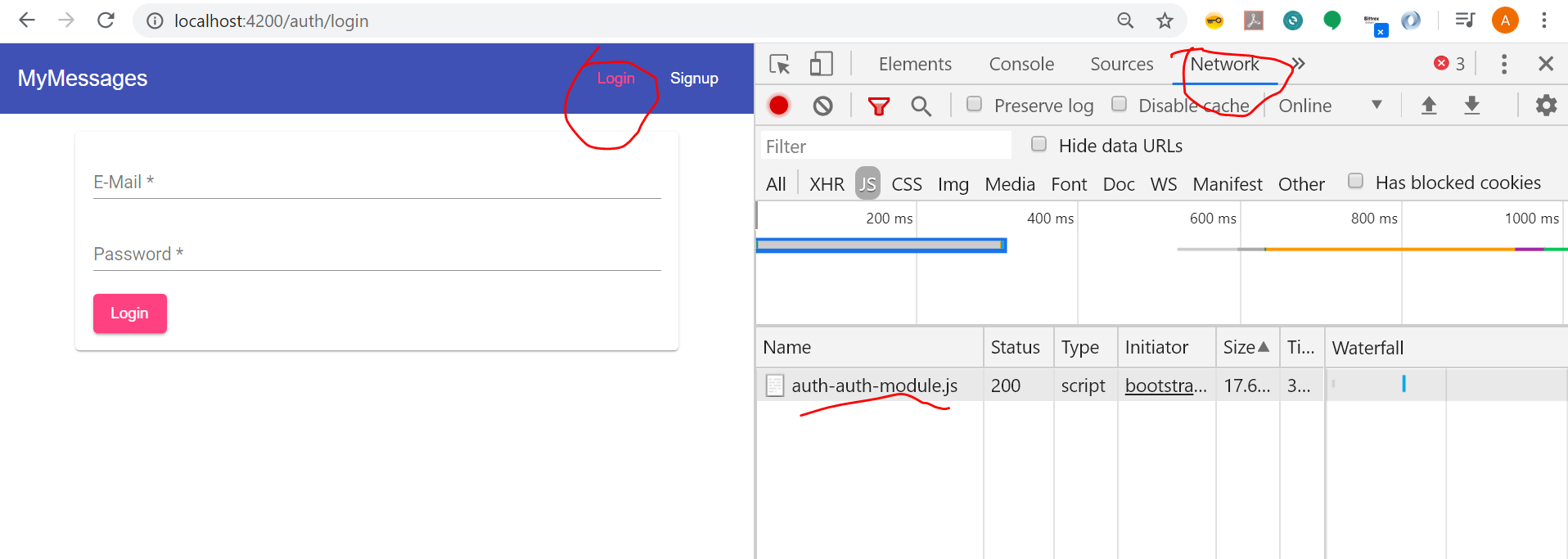
### Change the routes in header.component.html as shown below

### 

### Fix the auth guard

### 

### RUN THE APP and observe that auth module is lazily loaded in the network tab of browser



# Using a global angular config

### In posts.service.ts we have below URL repeated for each API

### <http://localhost:3000/api/posts>

### In auth.sevice.ts we have below URL repeated for each API

### <http://localhost:3000/api/user/signup>

### <http://localhost:3000/api/user/login>

### the above highlighted part of URL is common for all APIs in our application which can be moved global configuration file called environment.ts which is already added to the project by angular cli when we created the project

### Explain how it varies across environment

### 

### After this lets refactor the service code as below

### 

### Code for posts.service.js

import { environment } from "../../environments/environment";

const BACKEND\_URL = environment.apiUrl + "/posts/";

### similarly change for user.service.js using below code

import { environment } from "../../environments/environment";

const BACKEND\_URL = environment.apiUrl + "/user/";

### run the app and everything should work except delete

### for delete api add an extra “/” in the service route to fix the issue

## Using Node environment variables

### For backend we need to inject global variables to the running node app using hosting provider .

### In our demo case as we are using nodemon as hosting provider lets add below file and place the environment variables there

### In our case mongo DB password and JWT token are the 2 items we can include as global configuration items

### 

### Code

{

  "env": {

    "MONGO\_ATLAS\_PW": "QuBqs0T45GDKPlIG",

    "JWT\_KEY": "secret\_this\_should\_be\_longer"

  }

}

### Now we can use below string in code to get those values at runtime

### process.env.MONGO\_ATLAS\_PW

### process.env.JWT\_KEY

### make the code changes and restart the server as we change the configuration