**Bookmark Api**

**NestJs – FreeCodeCamp**

[**https://www.youtube.com/watch?v=GHTA143\_b-s**](https://www.youtube.com/watch?v=GHTA143_b-s)

1. Setup nest project  
   nest new bookmarkapi
2. Create auth module
3. Create auth controller
4. Create auth service  
   Graphical user interface, text

   Description automatically generated
5. Create routes for signin and signup in auth controller

Create equivalent functions in auth service

1. Set up prisma
   1. Install prisma and prisma client  
      *npm install -D prisma*  
      *npm install @prisma/client*
   2. *npx prisma init*
   3. Update database url in .env file
   4. Create model’s equivalent to tables in sql database in schema.prisma file

model User {

  id        Int      @id @default(autoincrement())

  createdAt DateTime @default(now())

  updatedAt DateTime @updatedAt

  email     String   @unique

  hash      String

  firstName String?

  lastName  String?

}

model Bookmark {

  id          Int      @id @default(autoincrement())

  createdAt   DateTime @default(now())

  updatedAt   DateTime @updatedAt

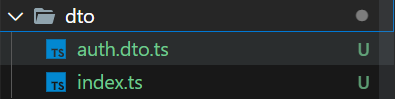
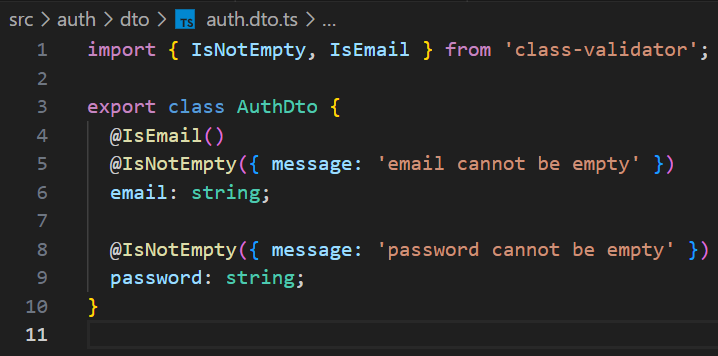
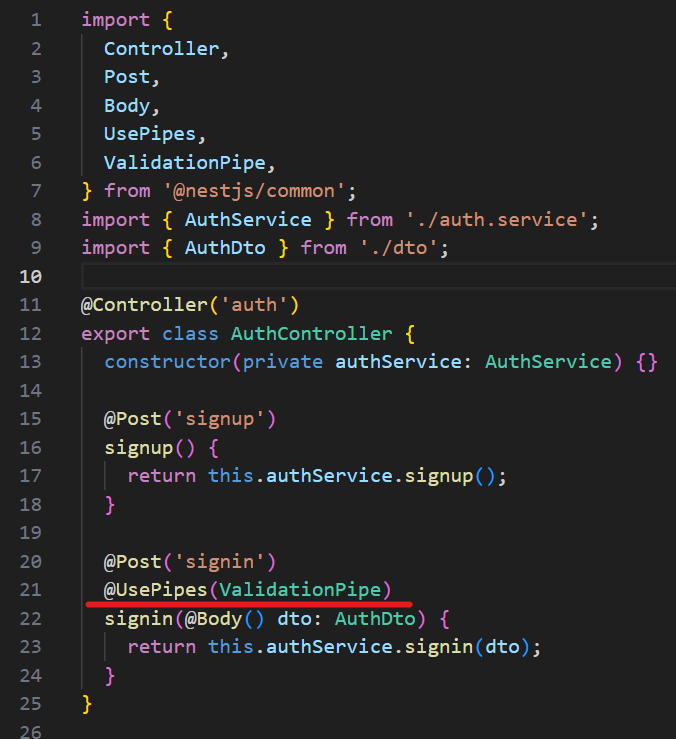
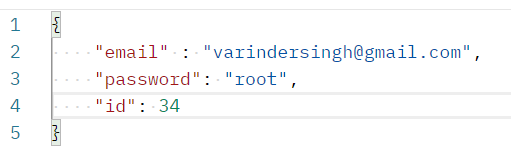
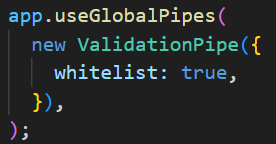
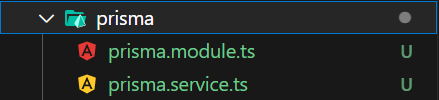
  title       String

  description String?

  link        String

}

* 1. Use following command to create tables in database from the models defined in schema.prisma  
     *npx prisma migrate dev*
  2. *npx prisma generate*  
     the above command takes the model/schema defined in schema.prisma (as shown in above picture) and converts them to a type in Typescript  
     for example type User, type Bookmark  
     and provides them through @prisma/client to be used in the application code.
  3. *npx prisma studio*

1. Validating incoming data to backend application from rest endpoints
   1. Install following libraries  
      *npm install class-validator  
      npm install class-transformer*
   2. Create auth.dto.ts
   3. Decorate the dto that you will be using to map the incoming data as shown below
   4. Now to make the validations work, you need to use pipes in nestjs as shown below
      1. Using validation pipe locally in a controller  
         
      2. Using validation pipe globally  
         Instead of decorating each endpoint with usePipe, we can use globalPipe in main.ts as shown below  
         
   5. ValidationPipe or Pipes in general can be used to modify the data or remove the redundant data that is received from user end, before it is actually passed to functions defined in a controller.  
      For example, say while signin flow the user is required only to send email and password but there is some extra data coming along that is not required as shown below  
        
      So in a situation like this we can use ValidationPipe as shown below to remove the redundant data  
      
2. Now to connect the database through the code follow the below steps
   1. Create a module and service named prisma  
      
   2. Write below mentioned code prisma.service.ts  
       PrismaService extends PrismaClient  
      PrismaClient contains the logic to connect to database  
      So PrismaService instantiates the PrismaClient by passing it the database url  
        
      PrismaService is exported through the PrismaModule

import { Injectable } from '@nestjs/common';

import { PrismaClient } from '@prisma/client';

@Injectable()

export class PrismaService extends PrismaClient {

  constructor() {

    super({

      datasources: {

        db: {

          url: 'postgresql://varinder:root@localhost:5432/bookmarkapi?schema=public',

        },

      },

    });

  }

}

* 1. Write below mentioned code in prisma.module.ts  
       
     PrismaModule is decorated as a global module so that every other module does not need to explicitly import it in order to use its service.

import { Global, Module } from '@nestjs/common';

import { PrismaService } from './prisma.service';

@Global()

@Module({

  providers: [PrismaService],

  exports: [PrismaService],

})

export class PrismaModule {}

1. Register a new user  
   update auth.service.ts for signup logic as shown below  
      
   Inject the PrismaService created above in the AuthService.  
   PrismaService will help to perform actions on the database  
   As we previously run the *prisma generate* therefore now the prismaService object has the reference for User.  
   \*\* Here I have used argon to hash the password.

import { Injectable } from '@nestjs/common';

import { PrismaService } from 'src/prisma/prisma.service';

import { AuthDto } from './dto';

import \* as argon from 'argon2';

@Injectable()

export class AuthService {

  constructor(private prismaService: PrismaService) {}

  async signup(dto: AuthDto) {

    // hash the user password

    const hash = await argon.hash(dto.password);

    // create new user in db

    try {

      const user = await this.prismaService.user.create({

        data: {

          email: dto.email,

          hash,

        },

      });

      // return the user

      return user;

    } catch (error) {

      // console.log(error);

      return { err: 'User registeration failed' };

    }

  }

}

1. Updating Model in schema.prisma  
     
     
   Run *npx prisma migrate dev*

model User {

  id        Int      @id @default(autoincrement())

  createdAt DateTime @default(now())

  updatedAt DateTime @updatedAt

  email     String   @unique

  hash      String

  firstName String?

  lastName  String?

  bookmarks Bookmark[]

  @@map("users")

}

model Bookmark {

  id          Int      @id @default(autoincrement())

  createdAt   DateTime @default(now())

  updatedAt   DateTime @updatedAt

  title       String

  description String?

  link        String

  userId Int

  user   User @relation(fields: [userId], references: [id])

  @@map("bookmarks")

}

1. Catch Unique Credentials Failed error

// create new user in db

    try {

      const user = await this.prismaService.user.create({

        data: {

          email: dto.email,

          hash,

        },

      });

      delete user.hash;

      // return the user

      return user;

    } catch (error) {

      if (error.code === 'P2002') {

        throw new ForbiddenException('Credentials aleady exists in database');

      } else {

        throw error;

      }

    }

1. Complete signin functionality

async signin(dto: AuthDto) {

  // get user with emailId

  const user = await this.prismaService.user.findFirst({

    where: { email: dto.email },

  });

  if (!user) {

    throw new ForbiddenException('User does not exists');

  }

  // console.log(user);

  // compare password with hash

  const isValid = await argon.verify(user.hash, dto.password);

  if (!isValid) {

    throw new ForbiddenException('Invalid Credentials');

  }

  delete user.hash;

  return user;

}

1. Access .env variables.
   1. install following package  
      *npm i @nestjs/config*
   2. Import ConfigModule in app.module.ts  
      and make it Global  
      Text

      Description automatically generated
   3. Using .env varialbes with ConfigService  
      Text

      Description automatically generated
2. JWT Authentication  
   Refer this [NestJS Chapter](https://docs.nestjs.com/recipes/passport)  
     
   1. Install following packages  
      *npm i @nestjs/passport passport @nestjs/jwt passport-jwt  
      npm install --save-dev @types/passport-jwt*
   2. Import JwtModule in auth.module.ts

import { Module } from '@nestjs/common';

import { AuthController } from './auth.controller';

import { AuthService } from './auth.service';

import { JwtModule } from '@nestjs/jwt';

@Module({

  imports: [JwtModule.register({})],

  controllers: [AuthController],

  providers: [AuthService],

})

export class AuthModule {}

* 1. Update the auth.service.ts  
     Create a function named **signToken** to create a jwt token from user credentials  
     and update the logic in **signin** function as shown below

async signin(dto: AuthDto) {

    // get user with emailId

    const user = await this.prismaService.user.findFirst({

      where: { email: dto.email },

    });

    if (!user) {

      throw new ForbiddenException('User does not exists');

    }

    // console.log(user);

    // compare password with hash

    const passwordMatches = await argon.verify(user.hash, dto.password);

    if (!passwordMatches) {

      throw new ForbiddenException('Invalid Credentials');

    }

    const authToken = await this.signToken(user.id, user.email);

    return authToken;

  }

  async signToken(

    userId: number,

    email: string,

  ): Promise<{ auth\_token: string }> {

    const payload = {

      sub: userId,

      email,

    };

    const token = await this.jwt.signAsync(payload, {

      secret: this.configService.get('JWT\_SECRET'),

      expiresIn: '15m', // expires in 15 minutes

    });

    return { auth\_token: token };

  }

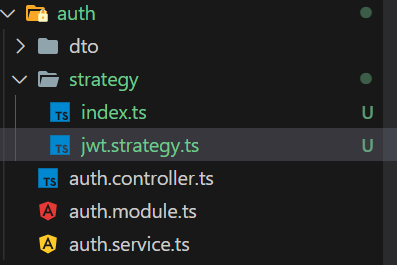
  async verifyToken(token: string) {

    return await this.jwt.verify(token, {

      secret: this.configService.get('JWT\_SECRET'),

    });

  }

1. Implement Strategy using passport  
   Refer this [NestJS Chapter](https://docs.nestjs.com/recipes/passport#implementing-passport-jwt)  
   1. Create a strategy class as shown below  
      
   2. Write the following code in jwt.strategy.ts  
        
      this class made as @Injectable() because it is going to be used by passport as a strategy to verify the jwt token

import { Injectable } from '@nestjs/common';

import { ConfigService } from '@nestjs/config';

import { PassportStrategy } from '@nestjs/passport';

import { ExtractJwt, Strategy } from 'passport-jwt';

@Injectable()

export class JwtStrategy extends PassportStrategy(

  Strategy,

  'jwt', // it sets 'jwt' name by default for this strategy because we used 'Strategy' from the passport-jwt, so we can skip

  // providing it here

  // otherwise we can provide a name we like such as 'jwt-verify-strategy'

) {

  constructor(configService: ConfigService) {

    super({

      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),

      ignoreExpiration: false,

      secretOrKey: configService.get('JWT\_SECRET'),

    });

  }

async validate(payload: any) {

        // we can perform any validation required here on the decoded jwt token

    // this will be appended to request object as user

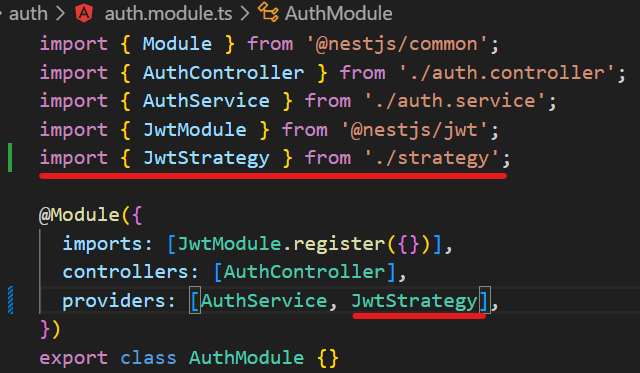
    // req.user

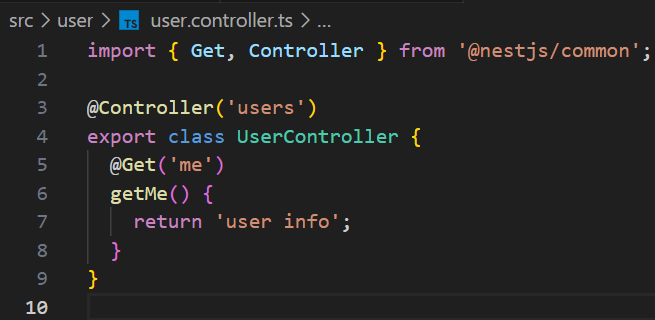
    return { userId: payload.sub, email: payload.email };

  }

}

* 1. Update auth.module.ts providers with JwtStrategy

So that the JwtStrategy is available to other modules that imports AuthModule  


1. NestJs Guards  
   Refer this [NestJs Chapter](https://docs.nestjs.com/guards)  
   First, let’s create a route that would require a valid jwt access token to access it.  
   To verify a jwt token the above created strategy will be used.
   1. Create User Controller  
      
   2. Create a route to Get user info  
      
   3. Let’s protect this route using NestJs Guards and passport as shown below

import { Get, Controller, UseGuards } from '@nestjs/common';

import { AuthGuard } from '@nestjs/passport';

@Controller('users')

export class UserController {

  @Get('me')

  @UseGuards(AuthGuard('jwt'))

  getMe() {

    return 'user info';

  }

}

We get the following Response without sending a valid jwt token

Graphical user interface, text, application, email

Description automatically generated

* 1. Update user controller

import { Get, Controller, UseGuards, Req } from '@nestjs/common';

import { AuthGuard } from '@nestjs/passport';

import { Request } from 'express';

@Controller('users')

export class UserController {

  @UseGuards(AuthGuard('jwt'))

  @Get('me')

  getMe(@Req() req: Request) {

    const user = req.user;

    return user;

  }

}

* 1. Create user.service.ts to get user info from database as shown below

import { Injectable } from '@nestjs/common';

import { PrismaService } from 'src/prisma/prisma.service';

@Injectable()

export class UserService {

  constructor(private prismaService: PrismaService) {}

  async getUserById(userReq) {

    const user = await this.prismaService.user.findUnique({

      where: { id: userReq.userId },

    });

    delete user.hash;

    return user;

  }

}

* 1. Update user controller

import { Get, Controller, UseGuards, Req } from '@nestjs/common';

import { AuthGuard } from '@nestjs/passport';

import { Request } from 'express';

import { UserService } from './user.service';

@Controller('users')

export class UserController {

  constructor(private userService: UserService) {}

  @UseGuards(AuthGuard('jwt'))

  @Get('me')

  async getMe(@Req() req: Request) {

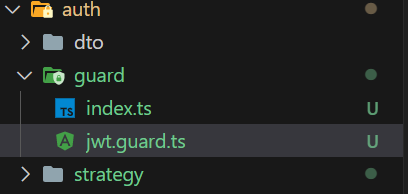
    const userReq = req.user;

    const user = await this.userService.getUserById(userReq);

    return user;

  }

}

* 1. Let’s extract the guard in separate file  
       
       
     write following code in jwt.guard.ts

import { AuthGuard } from '@nestjs/passport';

export class JwtGuard extends AuthGuard('jwt') {

  constructor() {

    super();

  }

}

* 1. Update user controller

import { Get, Controller, UseGuards, Req } from '@nestjs/common';

import { Request } from 'express';

import { UserService } from './user.service';

import { JwtGuard } from 'src/auth/guard';

@Controller('users')

export class UserController {

  constructor(private userService: UserService) {}

  @UseGuards(JwtGuard)

  @Get('me')

  async getMe(@Req() req: Request) {

    const userReq = req.user;

    const user = await this.userService.getUserById(userReq);

    return user;

  }

}

Continue tutorial = <https://youtu.be/GHTA143_b-s?t=7919>