Guide to NestJS sites

Week	
Files	
Read	☐
Unit/Module	
Property	
Date	

Step 1:

- Install Nest

```
npm install -g @nestjs/cli
```

- Create the project
  - Choose ‘npm’

```
nest new project-name
```

- cd into the new directory

Step 2: (Not necessary, associating project with a github repository)

- Create a repository
- Associate the repository with your local directory

```
git remote add origin url
```

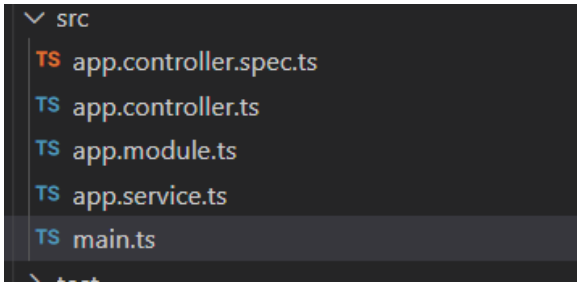
- Change the branch to main

```
git branch -M main
```

- Push your codes to the repository

N.B

main.ts is the program entry point



app.conroller.ts is the entry point for http requests. It makes calls to the **app.service.ts**

Step 3:

- Install nunjucks from your project's root

```
npm install nunjucks
npm install -D @types/nunjucks
```

- Edit the **main.ts** file to look like this

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { NestExpressApplication } from '@nestjs/platform-express'
import { join } from 'path'
import * as nunjucks from 'nunjucks'

async function bootstrap() {
  const app = await NestFactory.create<NestExpressApplication>(AppModule);
  const express = app.getHttpAdapter().getInstance();
  const views = join(__dirname, '..', 'views');
  console.log(views)
  nunjucks.configure(views, {express})
  await app.listen(3000);
}
bootstrap();
```

- Create a ***views*** directory - this is where all the html files to be displayed are stored
- Define the functions for the routes in your App.service.ts

```
function(): {} {
  return {variableName: variable};
}
```

👉 This means we're defining a function whose output is an object

- Define the methods in your app.controller.ts

```
@HTTPVerb
functionName(): {} {
```

```
  return this.appService.function();
}

e.g.
@Get()
  @Render('home.html')
  getHome(): {} {
    return this.appService.getHome();
  }
```

- In the notes, we had to define getHome and getAboutUs
- Add the relevant html files to the ***views*** directory
- Create a ***static*** directory - this is where static assets - css, pics etc. are stored.
- Here, we had to copy the ***public*** folder into it & then add this code to your ***main.ts***

```
const staticAssets = join(__dirname, '..', 'static');
app.useStaticAssets(staticAssets);
```

Step 4:

- Create a new module

```
nest g module modulename

e.g.

nest g module studentRegistration
```

- Launch a terminal from the newly created folder and run this

```
nest g resource
```

👉 We run this twice; to create the user resource then the student resource. We use the REST API & generate CRUD points

- Now it generates two folders - one for each resource. In this folder: we have entities and dto's. The entity is where you define the model. The dto is where you define how data is transferred.
- Now we need to install some packages

```
npm install @nestjs/mapped-types @nestjs/typeorm typeorm pg
```

Step 5: Connecting to a database

Step 5.1: Code side

- Create a file ***ormconfig.json*** and put this in, substituting the values to make it relevant

```
{
    "type": "postgres",
    "host": "localhost",
    "port": 5439,
    "username": "postgres",
    "password": "postgres",
    "database": "nestwafprimer",
    "entities": [
    "dist/**/*.entity{.ts,.js}"
    ],
    "synchronize": true
}
```

- Create a docker-compose.yml file with these things

```
version: '3'

services:
  database:
    container_name: examprep
    image: postgres:latest
    ports:
      - "5439:5432"
    env_file:
      - waf.env
    volumes:
      - /var/lib/postgresql/data
```

- Create an env file with the name referenced in env_file

```
POSTGRES_DB=nestwafprimer
POSTGRES_USER=postgres
POSTGRES_PASSWORD=postgres
```

- Run docker-compose on the file
- Edit your app.module.ts to look like this

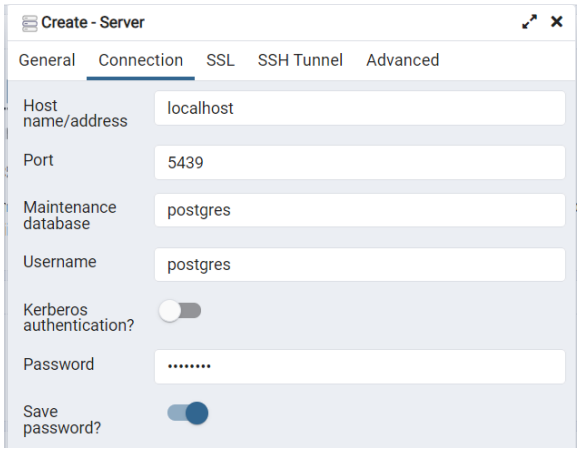
```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { StudentRegistrationModule } from './student-registration/student-registration.module';

@Module({
```

```
  imports: [TypeOrmModule.forRoot(), StudentRegistrationModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Step 5.2: Pgadmin

- Create a server with any name - I called mine tomilola
- In your connection settings, your hostname should be '*localhost*' and the port should be the one defined in your **docker-compose.yml** and **ormconfig.json**



👉 When you check the databases in this server, you should see the one defined in the **docker-compose.yml**

Step 5.3: Making the database visible to each module

- In the **users.module.ts**, add this

```
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';

@Module({
  imports: [TypeOrmModule.forFeature([User])],
  controllers: [UsersController],
  providers: [UsersService]
})
export class UsersModule {}
```

- In the **students.module.ts**, add this

```
import { Module } from '@nestjs/common';
import { StudentsService } from './students.service';
import { StudentsController } from './students.controller';
import { Student } from './entities/student.entity';
import { TypeOrmModule } from '@nestjs/typeorm';
```

```
import { User } from '../users/entities/user.entity';

@Module({
  imports: [TypeOrmModule.forFeature([Student, User])],
  controllers: [StudentsController],
  providers: [StudentsService]
})

export class StudentsModule {}
```

👉 How do we determine which one is *User* and which one is *Student*? since the codes in the modules are different

Step 5.4: Defining the entities and relationships

- In the entity file for the module, add '@Entity()'
- In the entity file for the module, define the columns you want

```
@Column()
columnName: datatype

// Inside the column's brackets, you can define properties like
// {Nullable: true} - meaning it's not compulsory
```

- In the service files, edit it to look like this

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';
import { User } from './entities/user.entity';

@Injectable()
export class UsersService {
  constructor(
    @InjectRepository(User)
    private usersRepository: Repository<User>
    ){}
}
```

👉 The injectrepository function for user allows you to assign the variable usersRepository to the actual repository

```
constructor(
    @InjectRepository(Student)
    private studentRepository: Repository<Student>,

    @InjectRepository(User)
    private userRepository: Repository<User>
  ) { }
```

👉 How do we know which one to add both to?

👉 At this point, when you restart the program, you'll see the table in pgAdmin

- In the user service, define the relevant functions to create a new user

```
async create(createUserDto: CreateUserDto) {
    const newUser: User = this.usersRepository.create(createUserDto)
    return this.usersRepository.save(newUser);
  }
```

- At this point, we edit the create-user.dto & create-student.dto to include the variables and datatypes

```
export class CreateEntityDto {
  readonly variablename: datatype;
  readonly variablename?: datatype;
}

// The ? shows that it's optional
```

- To note, in the users.controller.ts

```
@Post()
  create(@Body() createUserDto: CreateUserDto) {
    return this.usersService.create(createUserDto);
  }
```

@Body() indicates that we're getting this from the body & it initializes a variable of type CreateUserDto

Step 6: Front end

- In the ***views*** directory, create a new directory for the form, here i'm using ***users***
- Update your user controller to update the file

```
  @Get('create')
  @Render('users/create-user.html')
  createForm() {
  }
```

▼ If you're continuing from Pius' create-user.html

Copy the entire <script> block into the {% block body %} & {% endblock %}

Script Block Explanation

Step 7 : Implementing Relationships

- In the user entity, add this and include the relevant imports

```
@OneToOne(type => Student,student => student.user)
student: Student;
```

👉🏾 @Joincolum is used to specify the attribute that joins them together

Deconstructing OneToOne & JoinColumn

the JoinColumn is used on the side that has the foreign key, cascade lets you save it with a single call. The second parameter is the links it with the relationship defined in the **student.entity.ts**

- In the student entity, add this and include the relevant imports

```
@JoinColumn()
@OneToOne(type => User, user => user.student, {cascade:true})
user: User;
```

- In the student service, add these functions (these let you link and unlink records manually)

```
async setUserById(studentId: number, userId: number) {
   try {
     return await this.studentRepository.createQueryBuilder()
     .relation(Student, "user")
     .of(studentId)
     .set(userId)
   } catch (error) {
     throw new HttpException({
       status: HttpStatus.INTERNAL_SERVER_ERROR,
       error: `There was a problem setting user for student: ${error.messsage}`
     }, HttpStatus.INTERNAL_SERVER_ERROR)
   };
 }

// This links studentId to a user Id. This builds a query and sets the user Id to be
// the same as the student Id

async unsetUserById(studentId: number) {
  try {
    return await this.studentRepository.createQueryBuilder()
    .relation(Student, "user")
    .of(studentId)
    .set(null)
  } catch (error) {
    throw new HttpException({
      status: HttpStatus.INTERNAL_SERVER_ERROR,
      error: `There was a problem unsetting user for student: ${error.messsage}`
```

```
    }, HttpStatus.INTERNAL_SERVER_ERROR)
  };
}
```

- In your student controller, add these to define the routes for linking and unlinking manually

```
@Patch('studentId/user/userId')
  setUserbyId(@Param('studentId') studentId: number, @Param('userId') userId: number) {
    return this.studentsService.setUserById(studentId, userId)
  }

  @Delete('studentId/user/userId')
  unsetUserbyId(@Param('studentId') studentId: number) {
    return this.studentsService.unsetUserById(studentId)
  }
```

Step 7.1 : Making relationships in one HTML request

- Add the user field to the CreateStudentDto and the type should be the CreateUserDto
- Add this function to the students service

```
async create(createStudentDto: CreateStudentDto) {
    const newStudent = this.studentRepository.create(createStudentDto);

    if(createStudentDto.user){
      const newUser = this.userRepository.create(createStudentDto.user);
      const user: User = await this.userRepository.save(newUser)
      newStudent.user = user;
    }
    return this.studentRepository.save(newStudent);
  }
```

When this function is called, it creates a new student - if there’s no user, it creates a new user

Step 7.2: Updating the HTML to allow you pass everything