

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ  
Санкт-Петербургский национальный исследовательский университет  
информационных технологий, механики и оптики  
Мегафакультет трансляционных информационных технологий  
Факультет информационных технологий и программирования

**Лабораторная работа №5**  
**По дисциплине «Web-программирование»**  
**Создание пользовательских представлений и дальнейшего связыванием их с**  
**интерфейсом**

**Выполнили студент группы М33081**  
**Аль Даббагх Харит Хуссейн**

**Проверил**  
**Приискалов Роман Андреевич**

**САНКТ-ПЕТЕРБУРГ**

**2022**

## СОДЕРЖАНИЕ

Updating the API .....	2
Validation .....	2
Exception filters .....	3
CRUD Operations .....	3
Dealing with the frontend.....	5

## UPDATING THE API

Ongoing with our API development, we are required to actually implement the business logic for our formerly created API structure. During the implementation comes the real realization of what actually is required to be done. Mainly implementation of the CRUD methods on the controller side.

### Validation

For this lab work we were requested to work with validation, for that first we need to install a couple of things:

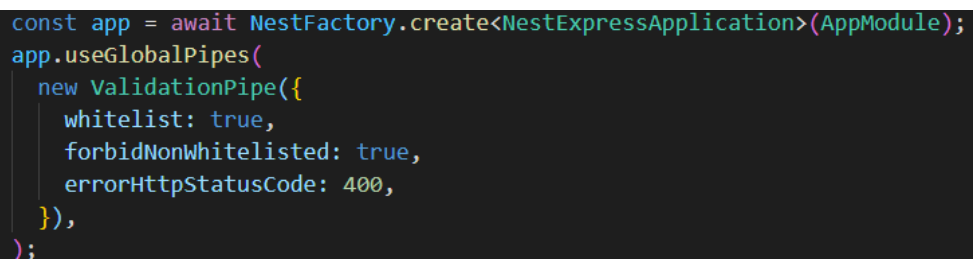
```
npm i --save class-validator class-transformer
```

In the NestJS documentation they call these mechanisms as Guards. And they are mainly added as decorators to our models. We can't add decorators to our schema directly, the best we to do so is by creating a DTO or Data Transfer Object, an example of which is my implementation for the user DTO:



```
TS user-dto.ts M X
src > user > dto > TS user-dto.ts > ...
1  import { IsNotEmpty, IsString, IsEmail } from 'class-validator';
2
3  export class UserDto {
4      @IsNotEmpty()
5      @IsString()
6      @IsEmail()
7      email: string;
8
9      @IsNotEmpty()
10     @IsString()
11     name: string;
12 }
```

But there are other types of validation that need to be checked before actually the incoming data on the application level. We implement that by adding the following lines to our main.ts file:



```
const app = await NestFactory.create<NestExpressApplication>(AppModule);
app.useGlobalPipes(
    new ValidationPipe({
        whitelist: true,
        forbidNonWhitelisted: true,
        errorHttpStatusCode: 400,
    }),
);
```

Each of these has it's own meaning which can be found in the documentation [here](#).

## Exception filters

NestJS comes with a built-in exceptions layer which is responsible for processing all unhandled exceptions across an application. We can create custom check and then throw our own HTTP Exception which will right away send a reply to the requester with a status code and message of our own, an example is I used inside the implementation of the service for the userID:

```
if (!+id) throw new HttpException('User ID Provided is not a number!', 400);
```

+id is a hack to convert id from string to number

!+id just means that if the ID is not a number (A non number value will result in NaN and thus !NaN will be true)

## CRUD Operations

All the Create, Read, Update and Delete are implemented in the controllers and services for the user and article models. It's really inconvenient to include all of it as screenshots in the report, so it's better to look directly in the Github repository. But here is an example for user creation, deletion and update methods:

```
async createUser(CreateUserDto: UserDto): Promise<User> {
  const { email, name } = CreateUserDto;
  const user = await prisma.user.create({
    data: {
      email: email,
      name: name,
    },
  });
  return user;
}

async updateUser(id: number, CreateUserDto: UserDto): Promise<User> {
  const { email, name } = CreateUserDto;
  const user = await prisma.user.update({
    where: {
      id: +id,
    },
    data: {
      email: email,
      name: name,
    },
  });
  return user;
}

async deleteUser(id: number): Promise<void> {
  const user = await this.getUser(id);
  if (user) {
    await prisma.user.delete({ where: { id: +id } });
  }
}
```

I also updated the decorators for the controllers to show the best documentation in Swagger:

GET	/	✓
GET	/index.html	✓
GET	/schedule.html	✓
GET	/todolist.html	✓
GET	/fetch.html	✓
GET	/fetchFromOwnAPI.html	✓
GET	/video.html	✓
GET	/users Get All Users	✓
GET	/users/{id} Get user by ID	✓
POST	/users/create Create user	✓
POST	/users/{id}/update Update user by ID	✓
DELETE	/users/{id}/delete Delete user by ID	✓
GET	/articles Get All Articles	✓
GET	/articles/{id} Get article by ID	✓
GET	/articles/user/{id} Get articles created by user using his ID	✓
POST	/articles/create Create article	✓
POST	/articles/{id}/update Update article by ID	✓
DELETE	/articles/{id}/delete Delete article by ID	✓

## DEALING WITH THE FRONTEND

For the frontend side of the lab, I've implemented just so little to show the methods work (Testing before was done using both Swagger and definitely Postman).

I created a new page in my site called "Fetch Own API". It contains two forms and a load button. Wrote a couple of javascript files and included them in the pages.

The first and second forms are responsible for creating users and articles, respectively.

The load button loads a table with the articles of a random user based on their ID (default is first 5).

Below is an example and confirmation of how these work.

The screenshot displays the 'Fetch Own API' frontend. At the top, a teal header contains the text 'Certifications' and 'Fetch Own API'. Below the header, the page is divided into two main sections. The first section, titled 'Create new User', features two input fields labeled 'Email' and 'Name', followed by a teal button labeled 'Create User'. The second section, titled 'Create new Article', features three input fields labeled 'Title', 'Content', and 'AuthorID', followed by a teal button labeled 'Create Article'. At the bottom of the page, there is a large teal button labeled 'Load'.

## Create new User

Email

Name

## Create new Article

Title

Content

AuthorID

User ID	Article ID	Title	published?
4	10	Lab5 Title	false