



Creating an API with NestJS

Presenting an option to create backends using JavaScript/TypeScript in an organized and easy to maintain way

12.MAY/2021

SOFTWARE ENGINEERING

By [Alex Carneiro](#)

Introduction

Currently, to create frontend applications using JavaScript, the frameworks [Angular](#), [React](#) e [Vue](#) stand out for their maturity and popularity. In addition, there are many examples and use cases available for study, which allows developers to easily find examples to quickly create testable and extensible applications.

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.
To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

allows for high flexibility, but this can become a problem for beginners; or even when working with teams, since the code is in danger of becoming unstructured.

[NestJS](#) solves this problem by staying on a layer above Express, with a well-defined and ready-to-use architecture. Including the use of already consolidated and tested libraries, reducing decisions for beginners, mitigating inconsistencies in the code and facilitating the understanding of those who are looking at the project for the first time, thus allowing to create testable applications, scalable, loosely coupled and easy to maintain, using filters, pipes, interceptors, among others.

The architecture, syntax and components are inspired by [Angular](#). Despite being aimed at frontend applications, it has architectural concepts that can easily be used in backend solutions. NestJS also inherits several framework concepts that are widely used in the corporate environment, such as Spring Boot and .NET Core.

Among the advantages of using NestJS, we have some highlights:

- Well-structured architecture, easy to learn and master;
- Uses [TypeScript](#);
- A very detailed [Documentation](#);
- Uses Express as a standard HTTP server (you can replace it with [Fastify](#) or create an adapter);
- Monolithic applications as standard, but with a structure prepared for [microservices](#) (including support for [gRPC](#), [NATS](#) and [Kafka](#), among others);
- Built-in support for [GraphQL](#) and [OpenAPI](#) (Swagger);
- Authentication via [Passport](#), [rate limiting](#) and other security features;
- Several ready-made “recipes”: [Mongoose](#), [Sequelize](#), [generator CRUDs](#),

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.
To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

- [Nest CLI](#) to create applications, update dependencies and facilitate the daily routine;
- [Open source](#).

Prerequisites

Before we start creating our sample API, to follow all the steps as described in this guide, you need to have installed:

- [Node.js](#), version 10.13 or higher;
- [Nest CLI](#) (optional, but recommended);
- [PostgreSQL](#), version 9 or higher (can be via Docker).

Creating our API

In this guide, we will show you the steps for creating a RESTful shopping list API, including service definitions via Swagger. Let's start?

With [Node.js](#) (≥ 10.13), the first step is to install [Nest CLI](#) and create our project, running in the terminal:



```
1 npm i -g @nestjs/cli
2 nest new nest-shopping-list
3
```

If you don't want to install the CLI globally, you can run it via npx (present in $\text{npm} \geq 5.2$). In this guide, however, the examples will be demonstrated with the CLI installed:



```
1 npx @nestjs/cli new nest-shopping-list
```

In the sequence, you'll have the list of files that were created and you'll have to choose the package manager of your preference (npm or yarn). Throughout this

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.
To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

```

4 CREATE nest-shopping-list/.eslintrc.js (631 bytes)
5 CREATE nest-shopping-list/.prettierrc (51 bytes)
6 CREATE nest-shopping-list/README.md (3339 bytes)
7 CREATE nest-shopping-list/nest-cli.json (64 bytes)
8 CREATE nest-shopping-list/package.json (1980 bytes)
9 CREATE nest-shopping-list/tsconfig.build.json (97 bytes)
10 CREATE nest-shopping-list/tsconfig.json (339 bytes)
11 CREATE nest-shopping-list/src/app.controller.spec.ts (617 bytes)
12 CREATE nest-shopping-list/src/app.controller.ts (274 bytes)
13 CREATE nest-shopping-list/src/app.module.ts (249 bytes)
14 CREATE nest-shopping-list/src/app.service.ts (142 bytes)
15 CREATE nest-shopping-list/src/main.ts (208 bytes)
16 CREATE nest-shopping-list/test/app.e2e-spec.ts (630 bytes)
17 CREATE nest-shopping-list/test/jest-e2e.json (183 bytes)
18
19 ? Which package manager would you ❤️ to use? (Use arrow keys)
20 > npm
21   yarn
22

```

Confirm your choice with [Enter], wait for the installation of the packages and you'll see the confirmation of the creation of the project, including the instructions to start the application.



```

1 ✓ Installation in progress... 🍪
2
3 🛠️ Successfully created project nest-shopping-list
4 📖 Get started with the following commands:
5
6 $ cd nest-shopping-list
7 $ npm run start
8

```

The nest-shopping-list folder was created with the basic structure of your new project. Navigate to it, and initialize to test:



```

1 cd nest-shopping-list
2 npm run start:dev
3

```

In a few moments, you should see something like this:



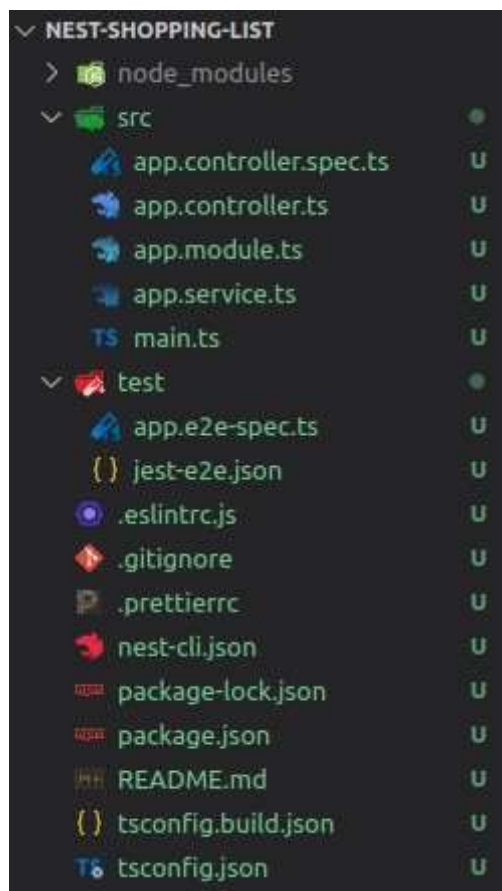
We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.
To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

With that, you can now test the server by opening <http://localhost:3000> in your browser to check out the good old “Hello World!”.

As we started with `npm run start:dev`, the server was initiated with the `--watch` option, monitoring changed files and reloading the application automatically. Check out more details and options in the [documentation](#).

Open the project in your favorite code editor, such as [VSCode](#), and you will see this structure:



In `main.ts` we have the `bootstrap()` function that performs the initialization of our application and starts listening on port 3000.



```
1 import { NestFactory } from '@nestjs/core';
2
3 import { AppModule } from './app.module';
```

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.
To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

You can open the `src/app.service.ts` file and update the return of the `getHello()` function to whatever you want, and save. By refreshing the browser page, you can check the updated API response.

After this initial test, finish the execution by pressing `Ctrl/CMD + C` on the terminal.

Now, we're all set, and we can effectively start creating your API!

Creating the structure for the entity

For our shopping list, we'll have the `Item` entity, with name, description (optional) and quantity of each item on the list.

We will use NestCLI's [CRUD](#) to automatically create our entity, its module, REST controller, service and DTOs, as well as the test `.spec` files.



```
1 nest generate resource item
```

At this point, we must choose the transport layer for our resource: REST, GraphQL, microservice or websocket gateway. Choose the REST API and for the question "Would you like to generate CRUD entry points? (Y / n) ", type Y and Enter.

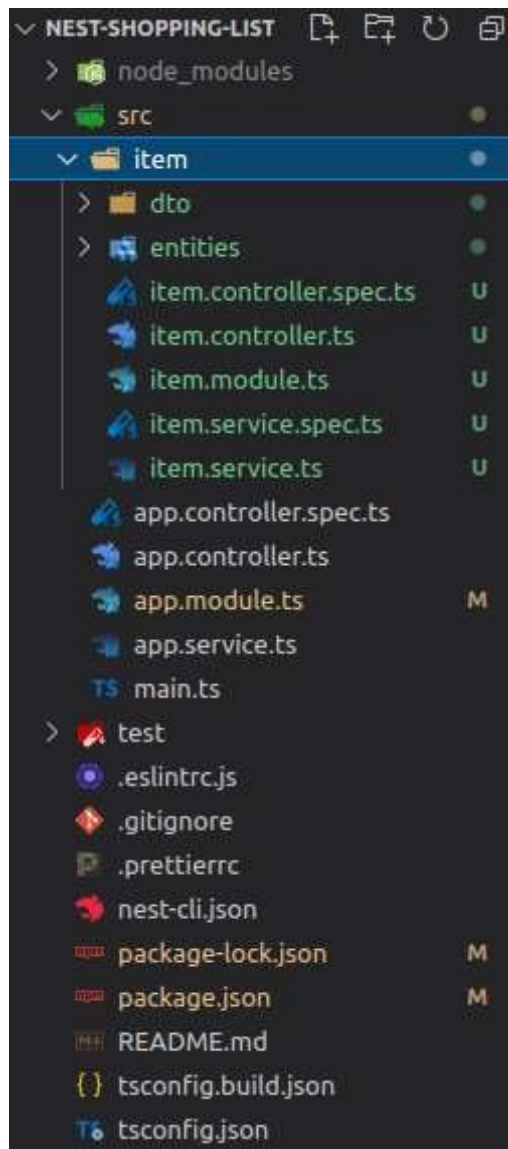


```
1 $ nest generate resource item
2 ? What transport layer do you use? REST API
3 ? Would you like to generate CRUD entry points? Yes
4 CREATE src/item/item.controller.spec.ts (556 bytes)
5 CREATE src/item/item.controller.ts (883 bytes)
6 CREATE src/item/item.module.ts (240 bytes)
7 CREATE src/item/item.service.spec.ts (446 bytes)
8 CREATE src/item/item.service.ts (607 bytes)
9 CREATE src/item/dto/create-item.dto.ts (30 bytes)
10 CREATE src/item/dto/update-item.dto.ts (169 bytes)
11 CREATE src/item/entities/item.entity.ts (21 bytes)
12 UPDATE package.json (2013 bytes)
13 UPDATE src/app.module.ts (308 bytes)
14 Packages installed successfully.
```

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.

To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept



- **module.ts:** our module, which specifies the controllers and providers that are necessary for the functioning of this module, and will be available in its scope. [Documentation](#);
- **controller.ts:** controller responsible for the REST service, with the methods memorized with `@Post()`, `@Get()`, `@Patch()` and `@Delete()`, all calling the `itemService`. [Documentation](#);
- **service.ts:** provider responsible for the Item business rule and calls to data sources. This service can also be exposed so that other application modules can use it, in case they need to interact with Item. [Documentation](#);
- **entities/item.entity.ts:** our Item entity, which for now is empty.

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.
To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

Before starting to change the code, let's install the [TypeORM](#) and PostgreSQL dependencies to manage our database. One of the interesting points of TypeORM is that it can keep our entity in sync with the structure in the database.



```
1 npm install --save @nestjs/typeorm typeorm pg
```

We'll also add the dependency of [ConfigModule](#), responsible for loading the declarations from the .env file in the environment variables and not maintaining the connection properties of the database and other sensitive information (or customized) directly in the code.



```
1 npm install --save @nestjs/config
```

Internally, @nestjs/config uses the popular library [dotenv](#).

After the installation is complete, we will start by creating our .env file at the root of the project. Update the example if your PostgreSQL database settings are different.



```
1 SERVER_PORT=3000
2 MODE=DEV
3 DB_HOST=127.0.0.1
4 DB_PORT=5432
5 DB_USERNAME=postgres
6 DB_PASSWORD=
7 DB_DATABASE=shopping_list
8 DB_SYNCHRONIZE=true
```

Important

Don't forget to add a line with .env to the .gitignore file, so that it doesn't accidentally stay in the repository.

From this point on, the files referenced in the guide will always be based on the

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.

To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept



```
1 import { Module } from '@nestjs/common';
2 import { ConfigModule } from '@nestjs/config';
3 import { TypeOrmModule } from '@nestjs/typeorm';
4 import { AppController } from './app.controller';
5 import { AppService } from './app.service';
6 import { ItemModule } from './item/item.module';
7
8 @Module({
9   imports: [
10     ConfigModule.forRoot(),
11     TypeOrmModule.forRoot({
12       type: 'postgres',
13       host: process.env.DB_HOST,
14       port: parseInt(process.env.DB_PORT),
15       username: process.env.DB_USERNAME,
16       password: process.env.DB_PASSWORD,
17       database: process.env.DB_DATABASE,
18       entities: [__dirname + '/*/*.entity{.ts,.js}'],
19       synchronize: (process.env.DB_SYNCHRONIZE === 'true'),
20     }),
21     ItemModule,
22   ],
23   controllers: [AppController],
24   providers: [AppService],
25 })
26 export class AppModule { }
27
```

Important

The TypeOrmModule's synchronize property was enabled in the .env file and is useful in the development phase, as it performs the automatic synchronization of the database with the specifications of the application entities. But it **should not be used in production**, as it can cause data loss!

Extra

What about using the SERVER_PORT env on main.ts, instead of leaving it fixed on port 3000? Check out a tip in the ConfigModule documentation:

<https://docs.nestjs.com/techniques/configuration#using-in-the-maints>

Defining the Item

Now, we can finally define our Item entity. Open the item/entities/item.entity.ts file and update the content to:

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.

To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

```
8 @UpdateDateColumn({ name: 'updated_at', type: 'timestampz', default: () =>
  'CURRENT_TIMESTAMP' })
9 updatedAt: Date;
10
11 @Column({ name: 'name', type: 'varchar', length: 50 })
12 name: string;
13
14 @Column({ name: 'description', type: 'varchar', nullable: true, length: 255 })
15 description?: string;
16
17 @Column({ name: 'quantity', type: 'int' })
18 quantity: number;
19 }
20
```

Note that the definitions of the mapping of the relational object of TypeORM are all made by annotations in the classes and attributes.

The default mapping for the string type is a varchar(255) column and for the number type it's an integer (depending on the database). Check out more information in the [documentation](#).

In the example of the description column that we declared, we could only use `@Column({ nullable: true })`, but we fill it all in for more detail. For `updatedAt`, the decorator `@UpdateDateColumn` was used, which updates the value automatically, whenever an update is made to the registry. There are also `@CreateDateColumn`, `@DeleteDateColumn` and `@VersionColumn`. TypeORM calls them [Special Columns](#).

Our entity is now ready for use. 🍷

If we start the application now, the table will be created automatically in the database, with the structure defined above (due to the `synchronize: true` property defined in the `AppModule`).

But we have not yet updated `item.service.ts` to use TypeORM.

Before that, how about we create our DTOs?

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.
To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

When using DTOs, we may not directly expose our internal model (of the database) to those who consume the API, but rather a representation of the data with the relevant (or allowed) attributes for external use. Having greater control over the data and the possibility of a better performance (consulting only the necessary table columns) are some of the advantages of using DTOs. So, let's define them.

Open `item/dto/create-item.dto.ts` and replace the content with:



```
1 import { IsInt, IsNotEmpty, IsOptional, IsString, Min } from 'class-validator';
2
3 export class CreateItemDto {
4   @IsString()
5   @IsNotEmpty()
6   name: string;
7
8   @IsOptional()
9   @IsString()
10  description: string;
11
12  @IsInt()
13  @Min(0)
14  quantity: number;
15 }
16
```

Got an error out there?

It was because we did not install the dependency [class-validator](#), responsible for ensuring that the values received in our DTO are in accordance with what we expect. We also need a [class-transformer](#) for validation to be performed automatically. Install both with:



```
1 npm install --save class-validator class-transformer
```

You can check the full list of validators, as well as more details about them in the [documentation](#).

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.
To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

Here `extendsPartialType(CreateItemDto)` does the job by taking the properties of `CreateItemDto` and reusing them automatically; however, it now turns all attributes as optionals.

We will update `main.ts` to perform automatic validation, adding the line:



```
1 app.useGlobalPipes(new ValidationPipe());
```

The file will look like this:



```
1 import { ValidationPipe } from '@nestjs/common';
2 import { NestFactory } from '@nestjs/core';
3 import { AppModule } from './app.module';
4
5 async function bootstrap() {
6   const app = await NestFactory.create(AppModule);
7   app.useGlobalPipes(new ValidationPipe());
8   await app.listen(3000);
9 }
10 bootstrap();
11
```

You can see more details (and settings) of the validator in the [documentation](#).

Implementing the ItemService

As we mentioned earlier, `item/item.service.ts` is called by the controller (`ItemController`) and is responsible for the business logic. The code generated by the CRUD Generator contains the basic methods we need, but they're not implemented (they only return explanatory messages).

Using the Repository pattern, we create an abstraction for the way we obtain data for the entity: Whether it's from the application's standard or a secondary database, an external API via REST or any other way.

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.

To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

```

2 import { InjectRepository } from '@nestjs/typeorm';
3 import { Repository } from 'typeorm';
4 import { CreateItemDto } from '../dto/create-item.dto';
5 import { UpdateItemDto } from '../dto/update-item.dto';
6 import { Item } from '../entities/item.entity';
7
8 @Injectable()
9 export class ItemService {
10   constructor(@InjectRepository(Item) private readonly repository: Repository<Item>) { }
11
12   create(createItemDto: CreateItemDto): Promise<Item> {
13     const item = this.repository.create(createItemDto);
14     return this.repository.save(item);
15   }
16
17   findAll(): Promise<Item[]> {
18     return this.repository.find();
19   }
20
21   findOne(id: string): Promise<Item> {
22     return this.repository.findOne(id);
23   }
24
25   async update(id: string, updateItemDto: UpdateItemDto): Promise<Item> {
26     const item = await this.repository.preload({
27       id: id,
28       ...updateItemDto,
29     });
30     if (!item) {
31       throw new NotFoundException(`Item ${id} not found`);
32     }
33     return this.repository.save(item);
34   }
35
36   async remove(id: string) {
37     const item = await this.findOne(id);
38     return this.repository.remove(item);
39   }
40 }
41

```

Since we're using an external provider (TypeORM's Repository), we need to declare its module as an import of item.module.ts, leaving the file like this:



```

1 import { Module } from '@nestjs/common';
2 import { TypeOrmModule } from '@nestjs/typeorm';
3 import { Item } from '../entities/item.entity';
4 import { ItemController } from '../item.controller';
5 import { ItemService } from '../item.service';
6
7 @Module({

```

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.
To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

Finally, `item.controllers.ts` should now show errors in the last three methods (`findOne()`, `update()` and `remove()`) due to calls to `itemService` that now expect an `id` of type `string` (it was a number). Replace ``+ id`` with just ``id`` to remove the [cast](#) and ...

Voilà!

Now our API is implemented and it's possible to perform operations of create, read, update and delete (CRUD) of items! 🙌

To perform all operations, you need to send REST requests via [Postman](#), [Insomnia](#), [curl](#) or as you prefer to the following addresses (according to `item.controllers.ts`):

Method	URL	Body	Description
GET	<code>http://localhost:3000/item</code>		Returns all items
GET	<code>http://localhost:3000/item/<ID></code>		Returns the item with the specified ID
POST	<code>http://localhost:3000/item</code>	<pre>{ "name": "Banana", "quantity": 2 }</pre>	Create a new item from <code>CreateItemDTO</code>
PATCH	<code>http://localhost:3000/item/<ID></code>	<pre>{ "quantity": 5 }</pre>	Updates the item attribute with the specified ID
DELETE	<code>http://localhost:3000/item/<ID></code>		Removes the item with the specified ID

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.

To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

developers who are integrating to our API), using the [Swagger](#) module.

With a few steps, Swagger will be working. Begin installing the dependencies:



```
1 npm install --save @nestjs/swagger swagger-ui-express
```

Then update main.ts to look like this:



```
1 import { ValidationPipe } from '@nestjs/common';
2 import { NestFactory } from '@nestjs/core';
3 import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';
4 import { AppModule } from './app.module';
5
6 async function bootstrap() {
7   const app = await NestFactory.create(AppModule);
8   app.useGlobalPipes(new ValidationPipe());
9
10  const config = new DocumentBuilder()
11    .setTitle('Shopping list API')
12    .setDescription('My shopping list API description')
13    .setVersion('1.0')
14    .build();
15  const document = SwaggerModule.createDocument(app, config);
16  SwaggerModule.setup('api', app, document);
17
18  await app.listen(3000);
19 }
20 bootstrap();
21
```

Now, we just need to update our entity and the DTOs with decorators for Swagger to identify them:

In the item/dto/create-item.dto.ts file, add imports and `@ApiProperty()` to the attributes. In the case of description, which is optional, use `@ApiPropertyOptional()`. It's also possible to enter some options, such as example value and description. The file might look like this:



We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.
To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

```
11 @IsOptional()  
12 @IsString()  
13 description: string;  
14  
15 @ApiProperty({ example: 5, description: 'Needed quantity' })  
16 @IsInt()  
17 @Min(0)  
18 quantity: number;  
19 }  
20
```

There are additional decorators and options that you can also add to the controllers' methods and describe the functions. Check out the [sample code](#) of NestJS.

And for item/dto/update-item.dto.ts, it's simpler:



```
1 import { PartialType } from '@nestjs/swagger';  
2 import { CreateItemDto } from '../create-item.dto';  
3  
4 export class UpdateItemDto extends PartialType(CreateItemDto) {}  
5
```

It's basically the same, but changing the import {PartialType} from '@nestjs/mapped-types' to '@nestjs/swagger'.

Running the application

The big moment has arrived! Now the application is ready and with some basic documentation. We can start and see everything working with:



```
1 npm run start:dev
```

When you open <http://localhost:3000/api> in the browser, Swagger's page will be displayed with the detailed API, including the DTOs! You can even make test calls right there.

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.
To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

Shopping list API ^{1.0} ^{OAS3}

My shopping list API description

default



POST /item Create item

GET /item

GET /item/{id}

PATCH /item/{id}

DELETE /item/{id}

Schemas

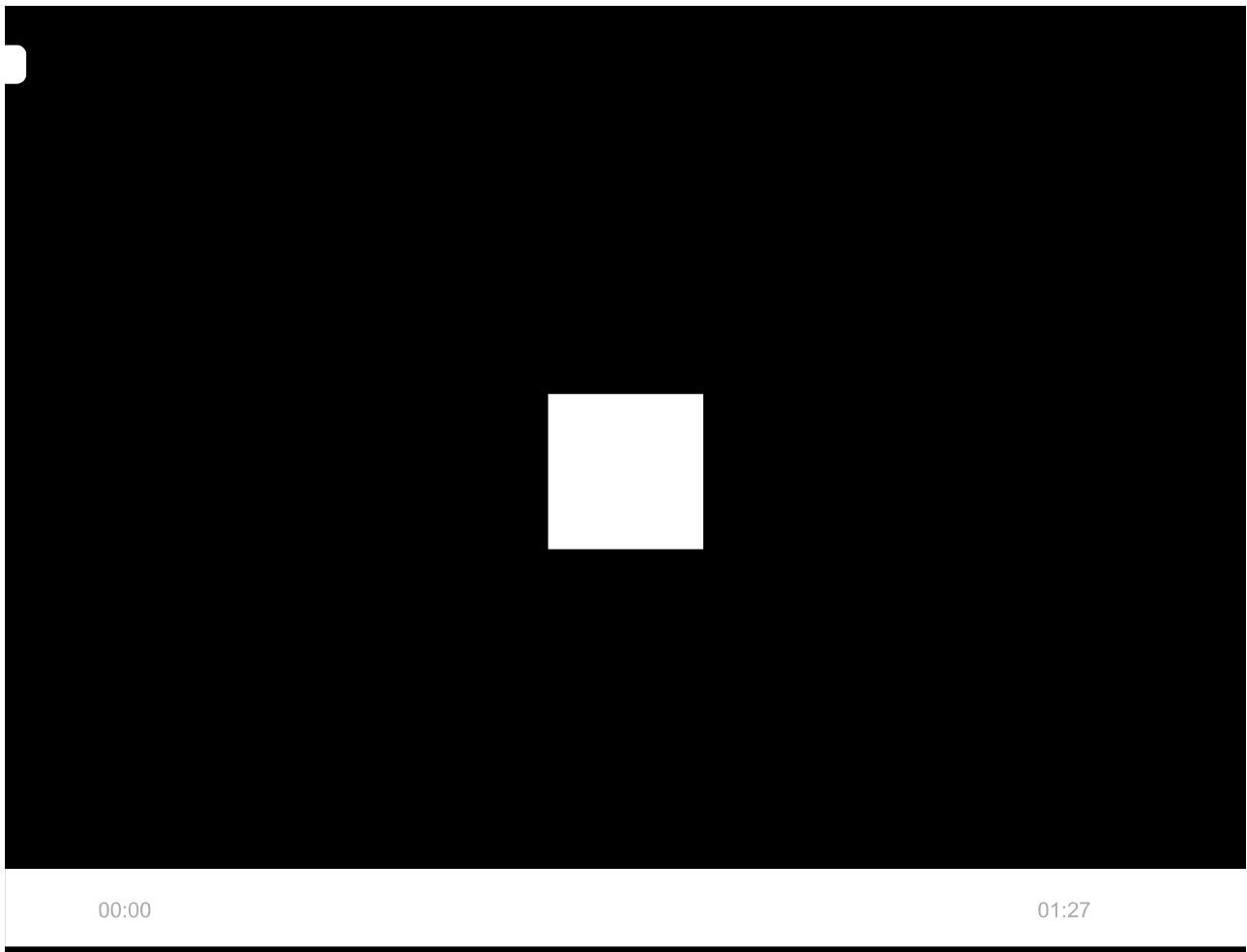


```
CreateItemDto {  
  name*      string  
              example: Bananas  
  description string  
              example: Cavendish bananas  
              Optional description of the item  
  quantity*  number  
              example: 5  
              Needed quantity  
}
```

UpdateItemDto >

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.
To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept



What about security?

Most of the time, we want our API to have user authentication in order to free access to available services. We do not cover these aspects in this guide, but the NestJS documentation has a specific section on the subject at <https://docs.nestjs.com/security/authentication>.

In addition to the authentication topic, we recommend the following readings and activations on your server:

- [Helmet](#), which adds HTTP headers to make the service a bit more secure;
- [CORS](#), to limit requests from browsers by domain;
- [CSRF](#), makes use of [csurf](#) to prevent attacks of [cross-site request forgery](#);

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.
To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

Important

These implementations don't make your application attack-proof, but they do help protect against some common and/or basic vulnerabilities.

Conclusion

As we said earlier, as NestJS has a predefined architecture and several plugins ready to use, the development of an API (especially with the help of NestCLI) is easy, with an organized code, well-structured by default, following good practices and easily documented via Swagger.

In this guide, we presented only a starting point. Continue to practice improving the code (how about creating the Category entity to [add](#) to Items?), [writing tests](#) and adding [authentication](#) to the API.

API, DEVELOPMENT, JAVASCRIPT, NESTJS

SIDECANNEL IS A
RING FROM

We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.
To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept

'TEMPEST'



We use cookies and other similar technologies to improve your experience. By using our system, you consent to such monitoring.
To learn more, including how to control cookies, see our [Privacy Policy](#)

Accept