# Nest.js Step-by-Step: Part 2

By **Bilal Haidar**
Published in: **CODE Magazine: 2019 - September/October**
Last updated: March 18, 2021

In the first part of this series (https://www.codemag.com/Article/1907081/Nest.js-Step-by-Step), you were introduced to Nest Framework and started building the To Do REST API using mock data.

Mock data for development and testing isn't sufficient to make a realistic and ready-to-launch app. Using a database to store your data is part process and is mandatory for making a great launch.

In this article, I'll use an instance of PostgreSQL database running locally on a Docker container. To access the database and execute the querie mutations, I'll make use of TypeORM, which is one of the most mature Object Relational Mapping (ORM) tools in the world of JavaScript. Nest with built-in support for TypeORM.



The source code for this article series is available on this GitHub repository: https://github.com/bhaidar/nestjs-todo-app/ and online at CODE associated with this article.

I'll start by introducing TypeORM and its features, then explore how Nest.js integrates with TypeORM. Finally, the step by step demonstration s how to convert the code from Part 1 into code that is database-aware and eliminates the use of any mock data.

> Nest.js can deal with a rich variety of databases ranging from relational databases to NoSQL ones.

## Related Articles

- NestJS Step-by-Step (Part 1)
- Nest.js Step-by-Step: Part 3 (Users and Authentication)
- NestJS Step-by-Step: Connecting NestJS with Angular (Part 4)

# What is TypeORM?

TypeORM is a JavaScript library that's capable of connecting to several database engines, including PostgreSQL, Microsoft SQL Server, and Mo just to name a few. By hiding the complexity and specificity of connecting to different database engines, TypeORM enables the communicatio your application and the back-end database of your choice.

TypeORM is built on top of TypeScript decorators that allow you to decorate your entities and their corresponding properties so that they ma database table with columns.

TypeORM supports both the Active Record and Data Mapper patterns. I won't be touching on these topics, but you can read more about them following this link/.

In general, I prefer using the Data Mapper pattern and specifically using Repositories to access the database. TypeORM supports the repositor pattern, so each entity has its own `Repository` object. These repositories can be obtained from the database connection itself.

In addition, TypeORM allows you to create your own custom repository by letting you extend the standard base repository and add any custo functions that you need.

Here's a quick summary of TypeORM features that I'm going to use in the application you're about to build:

- Entity Decorator to mark a JavaScript as an entity in the database
- Column Decorator to customize the mapping between a JavaScript object property and the corresponding column in the database. Custon includes specifying column data type, length, allow null or not, and other useful settings.
- A repository object per entity by auto-generating them. You can inject those objects in your Nest.js services and start accessing the databa
- Table relationships including one-to-one, one-to-many, and many-to-many relationships. In the course of this series, I'll be using mostly or many and many-to-many relationships.

I strongly advise you give it a try if you're serious about using TypeORM in your professional projects. Access everything TypeORM at: **https://TypeORM.io**.

# How Nest Framework Integrates with TypeORM

The **@nestjs/TypeORM** package is a Nest.js module that wraps around the TypeORM library and adds a few service providers into the Nest.js Dependency Injection system. The following is a list of services that are added by default:

- TypeORM database Connection object
- TypeORM EntityManager object (used with data mapper pattern)
- TypeORM Repository object per entity (for each entity defined in the application)

Every time a service or controller in your application injects any of the above services, Nest.js serves them from within its Dependency Injectior

You can check the source code for this module by following this URL: https://github.com/nestjs/TypeORM.

# Demo

"The proof of the pudding is in the eating!"

This is a famous saying that has always haunted me since I started software development. To best understand the concepts that I'm going to e recommend that you grab the source code from Part 1 of this series, and follow along to apply it as you go. Another saying is practice makes p let's start!

## Setup Docker & PostgreSQL database

**Step 1:** Open the application in your favorite editor and create a new git branch to start adding database support.

```
git checkout ?b add-db-support
```

**Step 2:** Install the NPM packages that are required to run the application and connect to the database by issuing the following command:

```
yarn add @nestjs/TypeORM TypeORM pg
```

This command installs three NPM packages:

- **@nestjs/TypeORM** is the Nest.js module wrapper around TypeORM.
- **TypeORM** is the official NPM package for TypeORM library.
- **pg** is the official library connector for PostgreSQL database.

**Step 3:** Setup a docker-compose file to run an instance of PostgreSQL on top of Docker. Docker is a prerequisite you have to have on your co order to run this step.

```
version: '3'
services:
db:
container_name: todo_db
image: postgres:10.7
volumes:
- ./db/initdb.d:
/docker-entrypoint-initdb.d
ports:
- '5445:5432'
```

The docker-compose file you'll use for the application is simple. The file defines a single service called `db`. The Docker file defines the followir for the db service:

- **container_name:** The name Docker assigns to the new container
- **image:** The image that Docker needs to download and instantiate a new container based on that image
- **volumes:** Docker creates a container volume by mapping a local directory in your application into the /docker-entrypoint-initidb.d directo the Postgres container.
- **ports:** Port mapping allows Docker to expose the PostgreSQL service running inside the container on port 5432 to the host computer on p

To run any initialization script when Docker first creates the container, you can place the script file inside the `/db/initidb.d` directory and D automatically runs it upon creating the container.

> By defining a Docker volume mapped to a local directory, you have more control to your Docker container

**Step 4:** Create a new bash script file inside the `/db/initdb.d directory`. **Listing 1** shows the content of this file.

Listing 1: Bash file

```
#!/usr/bin/env bash

set -e

psql -v
ON_ERROR_STOP=1
--username "$POSTGRES_USER"
--dbname "$POSTGRES_DB" <<-EOSQL
CREATE USER todo;
```

```
CREATE DATABASE todo ENCODING UTF8;
GRANT ALL PRIVILEGES ON DATABASE todo TO todo;

ALTER USER todo WITH PASSWORD 'password123';
ALTER USER todo WITH SUPERUSER;
EOSQL
```

The script starts by including a shebang line that makes it an executable script. The script is straightforward. It starts by creating a new databas todo and a new database todo. It also makes the todo user a SuperUser and assigns it the password of `password123`.

Read more about customizing the initialization phase that docker uses to instantiate a new Postgres container here: https://docs.docker.com/samples/library/postgres/#initialization-scripts.

**Step 5:** If you're working on a Windows computer, I recommend converting the script file above to a UNIX format by issuing the following con

```
dos2unix init-users-db.sh
```

If you can't find the dos2unix utility on your computer, you can download it from this URL: https://sourceforge.net/projects/dos2unix/.

**Step 6:** Create the Docker container and build your database. Let's start by adding a new NPM script to the package.json file to make the task up Docker and creating the container an easier task. Add the following script:

```
"run:services": "docker-compose up && exit 0"
```

To start up the Postgres container, you simply run the following command:

```
yarn run "run:services"
```

This command creates the Postgres container, a new PostgreSQL database, and a database user.

**Step 7**: Verify that the database is up and running by opening a new command line session and running the following docker command:

```
docker exec -it todo_db bash
```

The command above starts an interactive docker session on the container instance that you've just created. By default, running this command bash session for you to run and execute your commands. To verify that the database exists, run the following commands:

- Connect to Postgres engine: psql ?U postgres
- List the existing database: \l

You should be able to see an entry for the `todo` database.

You can read more about docker exec command by following this URL: https://docs.docker.com/engine/reference/commandline/exec/.

## Configure TypeORM

**Step 1:** Initialize TypeORM by creating a new `ormconfig.json` file at the root of your application. Simply paste the JSON content in **Listing** JSON file:

Listing 2: ormconfig.json file

```
{
     "name": "development",
     "type": "postgres",
     "host": "localhost",
     "port": 5445,
     "username": "todo",
     "password": "password123",
     "database": "todo",
     "synchronize": false,
     "logging": true,
     "entities":
     ["src/**/*.entity.ts", "dist/**/*.entity.js"], "migrations":
     ["src/migration/**/*.ts", "dist/migration/**/*.js"], "subscribers":
     ["src/subscriber/**/*.ts", "dist/subscriber/**/*.js"],
     "cli": {
        "entitiesDir": "src",
        "migrationsDir": "src/migration",
        "subscribersDir": "src/subscriber"
        }
}
```

TypeORM expects this configuration file to create a connection to the database, deal with database migrations, and all things related. TypeOR supports a variety of means to store the connection options including JSON, JS, XML, and YML files, and also environment variables.

You can read more about the TypeORM connection options by following this URL: https://github.com/TypeORM/TypeORM/blob/master/docs ormconfig.md.

The most important connection options that you need for the application are:

- **Name:** The name of the configuration settings. You can have one named "development" and another named "production". You need one s configurations per running environment.
- **Type:** The type of the database that TypeORM is connecting to. In this case, the type is "postgres".
- **Host, port, username, password, and database:** These settings resemble the details of the connection string that TypeORM uses to conn underlying database.
- **Synchronize:** A value of **true** means that TypeORM will auto-synchronize to the application code and the database structure every time th application runs. This is good for development but you should avoid using it in production. Preferably, you should create database migrati each and every change you make in the application code, even when in development. Make it a habit. It'll guarantee a smooth transition w are ready to deploy your app and database to production.
- **Logging:** If you enable this setting, TypeORM will emit some logging messages on the application's console when it's running. This is help development phase.
- **Entities:** This is the path where TypeORM finds the entities your application is using and maps to a table in the database.
- **Migrations:** This is the path where TypeORM finds the migrations you create and runs them against the database.

> With Nest.js, there's a variety of ORMs to use in your application. You can even develop your own integration with any other mc
> and make it work inside Nest.js.

**Step 2:** Amend the `tsconfig.json` file of your application to look like the content of **Listing 3**.

Listing 3: tsconfig.json file

```
{
    "compilerOptions": {
        "target": "es5",
        "module": "commonjs",
        "moduleResolution": "node",
        "outDir": "./dist",
        "baseUrl": "./",
        "removeComments": true,
        "emitDecoratorMetadata": true,
        "experimentalDecorators": true,
        "sourceMap": true,
        "lib": ["es2015"],
        "paths": {
            "@todo/*": ["src/todo/*"],
            "@shared/*": ["src/shared/*"]
            }
        },
    "exclude": ["node_modules"]
}
```

The next few steps will make heavy use of the **@todo/** path, as you will see.

**Step 3:** Change all references to `TodoEntity` and `TaskEntity` in the application code to match the ones here:

```
import { TodoEntity } from '@todo/entity/todo.entity';
import { TaskEntity } from '@todo/entity/task.entity';
```

Later, when you start creating migrations, TypeORM requires having the same exact reference path to all entities stored in the database. For in
the application code references the same entity in two different places with two different paths, TypeORM assumes that these are two differen
To make sure that TypeORM is happy, use the same path for all entities all over the application code.

**Step 4:** I'm making use of a few helper methods that wrap TypeORM API calls. Create a new **/src/shared/utils.ts.** Copy the functions from **Lis**

Listing 4: /shared/utils.ts file

```
import { getConnectionOptions, getConnection } from 'TypeORM';
export const getDbConnectionOptions = async (connectionName: string = 'default',
) => {
    const options = await getConnectionOptions(process.env.NODE_ENV || 'development', );
    return {
        ...options,
        name: connectionName,
    };
```

```
    };

export const getDbConnection = async (connectionName: string = 'default') => {
    return await getConnection(connectionName);
 };

export const runDbMigrations = async (connectionName: string = 'default') => {
    const conn = await getDbConnection(connectionName);
    await conn.runMigrations();
};
```

The `getDbConnectionOptions()` function reads the TypeORM configuration settings from the `ormconfig.json` file based on the current environment. Remember back in Step 8, when you assigned a name for the configuration `settings` object as development? This is a conven that proves helpful, especially when you move the application to production. So you can easily add another configuration object with the nam `production`.

The `getDbConnection()` function retrieves a connection from TypeORM.

The `runDbmigrations()` function runs the pending migrations using the active database connection.

**Step 5:** Let's change the AppModule and make it return a `DynamicModule` by accepting TypeORM connection settings.

Locate the **/src/app.module.ts** file and replace its content with the code in **Listing 5**.

The module now defines a `forRoot()` method that accepts the connection options and returns a DynamicModule.

The DynamicModule is a normal Nest.js module that you return and customize the way you want. You can read more about DynamicModule c https://docs.nestjs.com/modules.

**Step 6:** Import the TypeORMModule by replacing the content of AppModule with the code in **Listing 6**.

Listing 6: /src/app.module/ts ? Import TypeORM

```
@Module({})
export class AppModule {
    static forRoot(
    connOptions: ConnectionOptions): DynamicModule {
        return {
        module: AppModule,
        controllers: [AppController],
        imports: [TodoModule, TypeORMModule.forRoot(connOptions)],
        providers: [AppService],
        };
    }
}
```

You simply import the TypeORMModule into the `imports` section of the AppModule and passing the connection options as follows:

```
TypeORMModule.forRoot(connOptions)
```

Now that the root module of your app imports TypeORMModule, let's continue and see how to import it for the feature module.

## Bootstrap Nest.js app

**Step 1**: Amend the **/src/main.ts** file to load the TypeORM connection options and pass them to AppModule. The code bootstraps the applica creates an instance of AppModule.

Replace the line below inside the `main.ts` file:

```
const app = await NestFactory.create(AppModule);
```

With the code below:

```
const app = await NestFactory.create(AppModule.forRoot(await
    getDbConnectionOptions(process.env.NODE_ENV)),
);
```

The code makes use of the help function defined to load the TypeORM connection options based on the current executing environment and t feeds the results to the `AppModule.forRoot()` method.

**Step 2:** Let's automate the process to run database migrations while the application is bootstrapping.

By default, Nest.js expects that you generate or create your own database migrations using TypeORM CLI commands and then run them again database.

Open the **/src/main.ts** file and add the following line of code just beneath the code block that creates the AppModule.

```
/**
* Run DB migrations
*/
await runDbMigrations();
```

You defined the `runDbMigrations()` method back in Step 4 of the Configure TypeORM section.

Now when the application is bootstrapping and before it starts listening to new HTTP requests, it runs any pending migration against the data makes sure that the application entity model is in sync with the database table model.

This is exactly what happens when you set **synchronize: "true"** in the ormconfig.json file. However, being able to decide when to run migratic you in the driver's seat with greater control on migrations.

## Define Entity Relationships

**Step 1:** Let's convert the `TodoEntity` and `TaskEntity` objects to real TypeORM entities.

The rule is simple. To map a JavaScript entity object to a PostgreSQL database table, you decorate the entity object with **@Entity(name: strin** decorator offered by TypeORM API. You can read up on TypeORM entities here: https://github.com/TypeORM/TypeORM/blob/master/docs/e

To map a property on the entity object to a column on the database table, you decorate the property with **@Column()** decorator, also offerec TypeORM API. **Listing 7** shows both entities decorated and ready to be used by TypeORM to create their corresponding database tables. Ther many decorators offered by TypeORM.

Listing 7: TodoEntity and TaskEntity classes

```
import { TaskEntity } from '@todo/entity/task.entity';
import {
    Entity,
    PrimaryGeneratedColumn,
    Column,
    CreateDateColumn,
    OneToMany,
    } from 'TypeORM';

@Entity('todo')
export class TodoEntity {
    @PrimaryGeneratedColumn('uuid') id: string;
    @Column({ type: 'varchar', nullable: false }) name: string;
    @Column({ type: 'text', nullable: true }) description?: string;
    @CreateDateColumn() createdOn?: Date;
    @CreateDateColumn() updatedOn?: Date;

    @OneToMany(type => TaskEntity, task => task.todo) tasks?: TaskEntity[];
    }
```

For example, you have relation decorators to define a relation between one entity and another, listener decorators to react to events triggered TypeORM, and many others. Here's a list of all decorators supported by TypeORM: https://github.com/TypeORM/TypeORM/blob/master/docs/decorator-reference.md.

TodoEntity defines a property `tasks` of type array of TaskEntity. Here, the relation is that one ToDo entity has one or many Task entities. Hen code uses **@OneToMany()** decorator for this purpose.

On the other side of the relation, TaskEntity defines a property `todo` of type TodoEntity. One Task belongs to one and only one ToDo entity.

**Step 2:** Import the TypeORMModule into the TodoModule feature module. With this import, you need to list all TypeORM entities that you wi repository for while coding the module.

```
@Module({
    imports:
    [TypeORMModule.forFeature([TodoEntity, TaskEntity])],
    controllers: [TodoController, TaskController], providers: [TodoService, TaskService],
    })
export class TodoModule {}
```

You import the module by adding it to the list of imports on the feature module and specifying the list of entities to manage and have their re ready for use.

**Step 3:** Generate a migration!

Before running a migration, make sure the ormconfig.json file has all the necessary information it needs to instruct TypeORM about the locatic migrations on disk.

If you recall in Step 8, one of the settings was to specify the local directory where migration files are stored inside.

```
"migrations": ["src/migration/**/*.ts", "dist/migration/**/*.js"]
```

It's important to add this configuration setting before you generate or run your migrations.

As a prerequisite, you also need to install the **tsconfig-paths** NPM package. This package helps loading modules whose location is specified i `paths` section of the `tsconfig.json` file.

Previously, you've added some paths to the `tsconfig.json` file so that you can always use a unique path to refer to the TodoEntity and Task entities. Because of that, this NPM package is needed. To install the package as a dev dependency, issue the following command:

```
yarn add tsconfig-paths -D
```

Now, to generate a migration, let's first add an NPM script so that you don't have to write a long command every time you want to generate a migration.

```
"TypeORM": "ts-node -r tsconfig-paths/register./node_modules/TypeORM/cli.js",
"migration:generate":"yarn run TypeORM migration:generate -n",
```

The first script is to run the TypeORM CLI using the **ts-node** Typescript execution and REPL for Node.js.

To generate a new migration, you simply use the following command:

```
yarn run "migration:generate" InitialMigration
```

The command compares the entity objects in the application to the corresponding database tables (if already present) and then generates the steps so that both models are in sync.

**Step 4:** Run the migrations!

Previously, you've configured the application to run any pending migrations during the bootstrapping phase.

To run the new migrations that you've generated, simply start the application and migrations will run automatically.

To run the application, issue the following command:

```
yarn run start:dev
```

**Step 5:** Change the TodoController class. There isn't much change required except decorating the **update()** and **create()** methods with the **@UsePipes()** decorator.

In Nest.js, `Pipes` allow you to transform data from one format to another. You can even use Pipes to perform data validation on the input tha client passes with the HTTP Request.

Nest.js comes with two built-in Pipes: `ValdiationPipe` and `ParseIntPipe`. You use the former to add validation over the input parameters latter to validate and convert an input parameter into a valid integer value.

You can also create your own custom Pipe. Go to: https://docs.nestjs.com/pipes for more.

For this project, you're going to use ValidationPipe. But first, you need to install the following NPM packages because the ValidationPipe uses internally.

```
yarn add class-transformer class-valdiator
```

Switch to the TodoController class and amend both the update() and create methods as shown in **Listing 8**.

Listing 8: TodoController class

```
@Post()
    @UsePipes(new ValidationPipe())
    async create(@Body() todoCreateDto:TodoCreateDto): Promise<TodoDto> {
        return await this.todoService.createTodo(todoCreateDto);
    }

    @Put(':id')
    @UsePipes(new ValidationPipe())
    async update(
        @Param('id') id: string,
        @Body() todoDto: TodoDto,): Promise<TodoDto> {
            return await this.todoService.updateTodo(todoDto);
        }
```

The code uses the **@UsePipes()** decorator passing a new instance of the ValidationPipe class. Now, to add validation rules on the TodoCreate TodoDto objects, let's decorate the `DTO` objects with a few validations that are offered by **class-validator** NPM package. For the complete list validation decorators that the class-validator library offers, visit: https://github.com/typestack/class-validator.

**Listing 9** shows all the DTO objects annotated with the proper validation rules.

Listing 9: DTO classes

```
export class TodoDto {
    @IsNotEmpty()
    id: string;

    @IsNotEmpty()
    name: string;

    createdOn?: Date;
    description?: string;
```

```
    tasks?: TaskDto[];
}

export class TodoCreateDto {
    @IsNotEmpty()
    name: string;

    @MaxLength(500)
    description?: string;
}
```

You can check the validation annotations on TaskDto and TaskCreateDto objects on the GitHub repository of this article or by checking out th version of this article.

## Use Repositories within your Services

Amend the TodoService class. There are major changes to this class to adjust the way the code manages data. Instead of using an in-memory store the To Do items and tasks, you'll vary that to connect directly to the database.

You start by injecting the `Repository<TodoEntity>` instance as follows:

```
constructor(
    @InjectRepository(TodoEntity)
    private readonly todoRepo:
    Repository<TodoEntity>,
) {}
```

The @nestjs/TypeORM defines the **@InjectRepository()** decorator. Its role is to retrieve a Repository instance for a specific entity from the Ne Dependency Injection system and make it available for the service.

Earlier, when you imported **TypeORMModule.forFeature([TodoEntity, TaskEntity])** into the module, Nest.js registered a Repository service (token and factory method) for each and every Entity in the Nest.js Dependency Injection System.

Next, you'll look at a few methods. The rest you can find in the source code accompanying this article at the GitHub repository and in the onlir of this issue of CODE Magazine.

**Listing 10** below shows how to implement `getOneTodo()` method. The method uses the `findOne()` function, available on the Repository i query for a single TodoEntity based on the TodoEntity ID column. In addition, it returns all of the related TaskEntity lists on this object. If the er found in the database, the code throws an HttpException. Otherwise, it converts the TodoEntity object into a TodoDto objects and returns the the calling controller.

Listing 10: getOneTodo() function

```
async getOneTodo(id: string): Promise<TodoDto> {
    const todo = await this.todoRepo.findOne({
        where: { id },
        relations: ['tasks', 'owner'],
    });

    if (!todo) {
        throw new HttpException(
```

```
        `Todo list doesn't exist`,
        HttpStatus.BAD_REQUEST,
       );
    }

    return toTodoDto(todo);
}
```

Let's take a look at the toTodoDto() utility function that maps a TodoEntity to a TodoDto object. You could use some other advanced libraries automapper but for this article, I've decided to keep it simple and create my own mapper function. **Listing 11** shows the toTodoDto() function

Listing 11: toTodoDto() function

```
export const toTodoDto = (data: TodoEntity): TodoDto => {
    const { id, name, description, tasks } = data;

    let todoDto: TodoDto = {
        id,
        name,
        description,
    };

    if (tasks) {
        todoDto = {
        ...todoDto,
        tasks: tasks.map(
        (task: TaskEntity) => toTaskDto(task)),
        };
    }

    return todoDto;
};
```

**Listing 12** shows the code for the TodoService createTodo() function.

Listing 12: createTodo() function

```
async createTodo(todoDto: TodoCreateDto): Promise<TodoDto> {
    const { name, description } = todoDto;

    const todo: TodoEntity = await this.todoRepo.create({
        name, description,
    });

    await this.todoRepo.save(todo);
    return toPromise(toTodoDto(todo));
}
```

> A best practice is to always inject Repositories inside your Services rather than working directly with Repositories inside your Controllers.

The Repository object exposes the `create()` function to create a new instance of an entity. Once you create a new instance of TodoEntity, yc entity in the database by using another function exposed by the Repository object, which is the `save()` function.

# Conclusion

In this article, you've seen how easy it is to connect a Nest.js application to a database using the TypeORM library.

In the upcoming article, you'll be looking at adding and dealing with users and authentication modules.

Happy Nesting!

## TypeORM

TypeORM is an ORM that can run in NodeJS, Browser, Cordova, PhoneGap, Ionic, React Native, NativeScript, Expo, and Electron platforms a be used with TypeScript and JavaScript (ES5, ES6, ES7, and ES8).

## @nestjs/TypeORM package

The @nestjs/TypeORM package wraps TypeORM into a native Nest.js module that integrates seamlessly with the Nest.js Dependency Inject system


### This article was filed under:

Nest.js Java Script JavaScript Mobile Development Node Modules node.js