# **Typescript**

## Khái niệm:

Được viết dựa trên javascript, Khi viết bằng ts thì phải compile code TS về JS để trình duyệt (hoặc runtime environment như NodeJS) có thể hiểu và xử lý được.

Compile bằng **TypeScript Compiler (tsc)**

## TypeScript Data Type

Danh sách các default types trong TS:

1. let someString: string;
2. let someNumber: number;
3. let someBoolean: boolean;
4. let something: any; // có thể gán sang cho bất kỳ kiểu dữ liệu nào khác
5. let someStringArray: string[]; // tương tự cho number[], boolean[], any[]
6. let someObject: object;
7. let someNull: null;
8. let someUndefined: undefined;
9. let someUnknown: unknown;
10. let someNever: never; // ví dụ như một hàm throw exception
11. let someTuple: [string, number];
12. let someVoidFunction: () => void; // một hàm không trả về giá trị gì sau khi thực thi
13. let someFunction: () => string; // một hàm trả về giá trị có type "string" sau khi thực thi

## TypeScript interface & type

**Giống nhau:** TS cung cấp Interface và Type để có thể định nghĩa được type cho một đối tượng (object)

### Interface:

Interface Task{

taskId: number,

taskName: string,

}

**TH1:**

Nếu khai báo như dưới TS sẽ bắt lỗi vì không có các thuộc tính taskId và taskName như ở interface Task,

Const task : Task = {

}

**TH2:**

Nếu khai báo như sau thì TS sẽ bắt lỗi vì chưa khai báo dữ liệu cho taskName.

Const task : Task = {

taskId: 12

}

Hoặc

Const task : Task = {

taskId: “abcd”

}

Thì sẽ bị lỗi vì taskId khai báo number và gán giá trị là string.

**TH3:** Nếu muốn khi khởi tạo 1 biến khi khởi tạo task thì làm như sau:

Dùng options ?: . Ví dụ dưới áp dụng cho taskName

Interface Task{

taskId: number,

taskName?: string,

}

Do đó khi khởi tạo task chỉ khai báo duy nhất thuộc tính taskId thôi thì vẫn không có lỗi

Const task : Task = {

taskId: 123

}

TH4: trong 1 file có thể khai báo nhiều interface trùng tên nhau

### Type

* Type trong cùng 1 file không thể khai báo trùng tên.

## Union Type

Union Type là những types mang tính chất nôm na là Hoặc cái này Hoặc cái kia. Để viết Union Type, dùng Pipe Symbol (|).

Let a: number | string

## Decorators

Hiểu đơn giản là các hàm js, có thể wrap một hàm bằng 1 hàm khác, hoạt động giống như HOC trong reactjs.

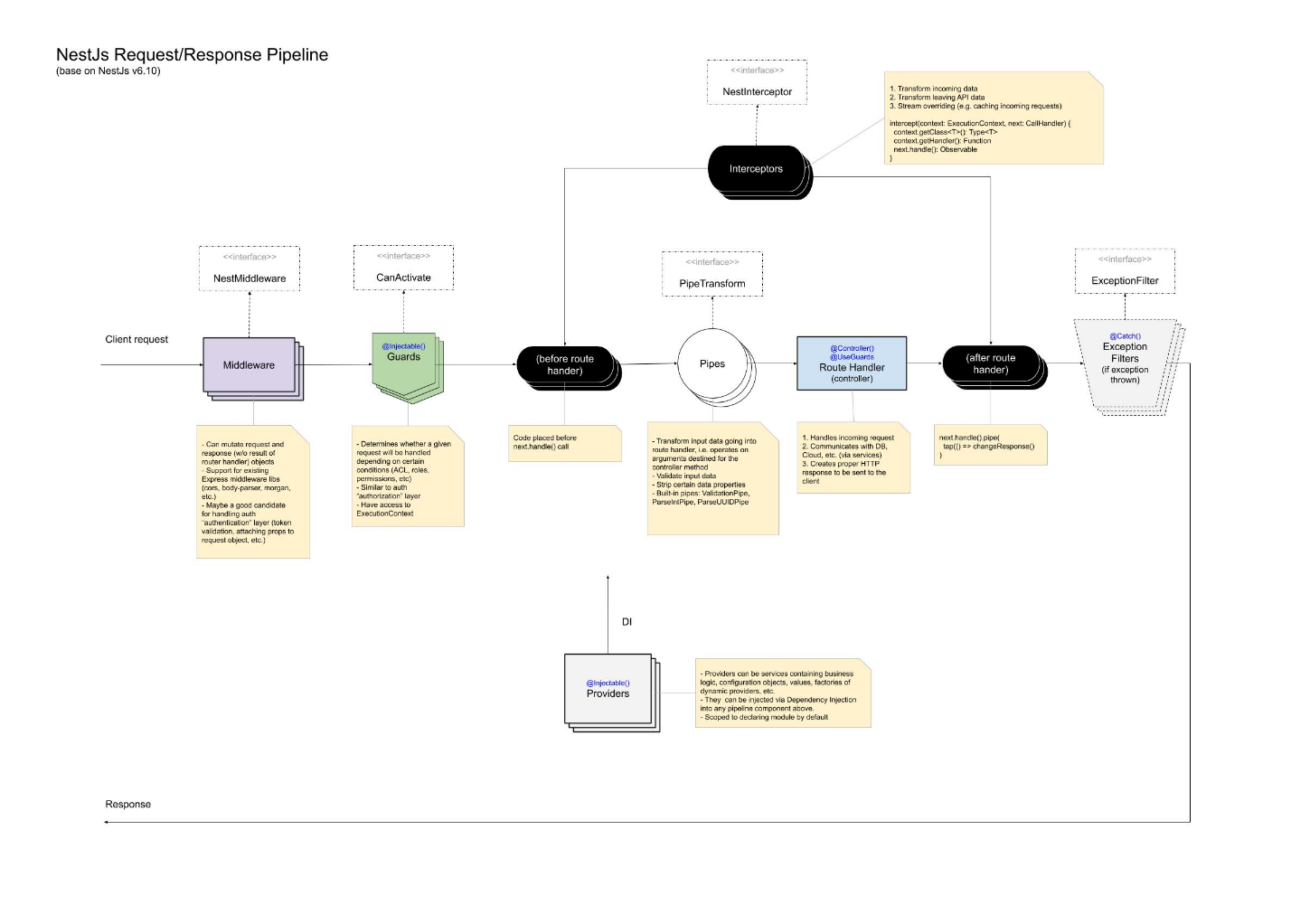
Decorator cách khai báo hơi đặc biệt, có thể được đính kèm 1 số metadata khi khai báo. Sử dụng từ khóa **@expression**  trong đó expression là 1 function sẽ được gọi

Ví dụ trong nestjs: 1 số decorator như @controller, @ modules

# **Nestjs**

life cycle diagram

## Life cycle diagram



## Section 1: Introduction to NestJS & Pre-requisites

### Cài đặt.

Cần đảm bảo Nodejs >= 10.13.0

Chạy lệnh sau:

npm i -g @nestjs/cli hoặc yarn global add @nestjs/cli

nest new project-name (với prokect-name là lên project muốn khởi tạo)

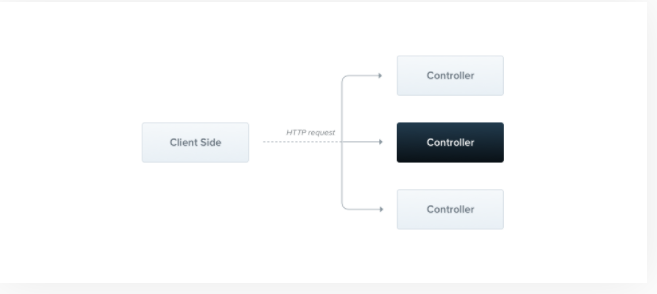
Khởi chạy ứng dụng: npm start hoặc yarn start

Mở trình duyệt truy cập : http://localhost:3000

### Các thành phần cơ bản

Cấu trúc chính project: main.ts => app.module.ts => app.controller.ts và app.service.ts

* **Main.ts**: **entry-point** của ứng dụng, là nơi khởi tạo ứng dụng
* **App.module.ts:** Module gốc của ứng dụng, được khởi tạo ở main.ts
* **App.controller.ts:** Controllers chịu trách nhiệm xử lý các request đến và response lại client, định nghĩa router (tức là url api) trong controller luôn.



Các phương thức Get, Post, Put, Patch và các phương thức hỗ trợ lấy thông tin ở trong body, params, query đều nằm trong gói **@nestjs/common** (link tham khảo chi tiết: <https://docs.nestjs.com/controllers> )

* **App.service.ts:** Là các class service có thể để xử lý các tác vụ lớn, phức tạp tương ứng ở controller.

Ngoài ra trong nestjs có khái niệm **provider:** đại diện cho các class services như trên hoặc các class khác được xem là **provider:** factory, repository,…

Ở ví dụ này.. các provider services sẽ được sử dụng ở controller thông qua **Dependency Injection (DI).**

Do đó, trong app.services.ts định nghĩa class đấy là 1 dependency thông qua decorator @Injectable() trong gói **@nestjs/common.**

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

@Injectable()

export class AuthService {

    //

}

Ngoài ra, **AppService** được inject vào **AppController** thông qua **constructor**

Sử dụng **Acess Modifỉer(public, private, protected)** trong constructor sẽ vừa khai

báo và khởi tạo thuộc tính trong class cùng lúc.

@Injectable()

export class AuthService {

    constructor (private jwtService: JwtService) { }

}

Để tạo 1 module, controller, service có thể dùng lệnh sau đây để tạo nhanh:

* **Module:** nest g module module-name (Với module-name là tên module)
* **Controller:** nest g controller controller-name (với controller-name là tên controller);
* **Service:** nest g service service-name (với service-name là tên service).

**Tóm tắt chức năng:**

* **\*\*\*.service.ts:** decorator @Injectable() sẽ khai báo cho biết \*\*\*.service.ts là 1 dependency.

@Injectable()

export class AuthService {}

* **\*\*\*.controller.ts:**  \*\*\*.controller khai báo nó cần dependency \*\*\*.service.ts thông qua constructor:

@Controller('auth')

export class AuthController {

    constructor(private authService: AuthService){}

* **\*\*\*.module.ts:** trong file này \*\*\*.service được import vào mà truyền vào trong mảng provides của decorator @module()

@Module({

  imports: [

    ConfigModule,

  ],

  providers: [AuthService, JwtStrategy], // để service ở đây

  controllers: [AuthController],

  exports:[JwtStrategy, PassportModule]

})

## Section 2: Task Management Application (REST API)

## Section 3: Validation and Error Handling

### Pipes:

được định nghĩa là một class, thường được sử dụng trong hai trường hợp:

1. Chuyển đổi dữ liệu đầu vào thành dạng dữ liệu mong muốn,
2. Kiểm tra dữ liệu đầu vào và báo lỗi nếu như dữ liệu đó không thoả mãn điều kiện.

Cách dùng:

1. Dùng usePipes(pipe cần chạy). đặt dưới phương thức trên hàm controller

@Post()

@UsePipes(new JoiValidationPipe(createCatSchema))

async create(@Body() createCatDto: CreateCatDto) {

this.catsService.create(createCatDto);

1. Truyền pipes vào làm tham số của hàm controller

@Get(':id')

async findOne(@Param('id', ParseIntPipe) id: number) {

return this.catsService.findOne(id);}

### Middleware

Ngoài ra có thể viết middleware để xác thực request trước khi vào controller để xử lý., …

Cần lưu ý: viết middleware phải gọi NextFunction sau lúc xử lý middleware để có thể tiếp tục chạy tới controller nếu pass qua middleware, không thì sẽ bị treo.

### Throw Exception:

1. Ở nestjs throw luôn ở service\controller, không cần trycatch như expressjs
2. Ở Expressjs có thể throw tùy ý, ví dụ: throw [‘Message-error’] cho client xử lý,

Còn ở Nestjs throw exception dùng những class, bộ lọc sẵn có, nếu không dùng, dùng sai cách thì mặc định sẽ trả về định dạng:

{

"statusCode": 500,

"message": "Internal server error"}

Có thể check req trong middle, ví dụ check xem request gửi lên có kèm roleId trong header của request hay không.

export class AuthMiddleware implements NestMiddleware{

use(req: Request, res: Response, next: NextFunction) {

const roleId = req.header("roleId")

if (!roleId) {

console.log('roleId', roleId);

// res.send("Từ khi gặp em")

res.status(400).json({

success: false,

messages: ['ROLEID\_IS\_REQUIRED'], // client có thể đọc message để translate bắn thông báo cho người dùng

content: roleId // có thể return giá trị gì đó về client

})

res.end(); // end

// throw new NotFoundException('roleId is Required')

} else {

next();

}

}

}

Để dùng middleware này thì chuyển qua file app.modules.ts.

export class AppModule implements NestModule{

configure(consumer: MiddlewareConsumer) {

consumer.apply(AuthMiddleware).forRoutes(AppController)

// .forRoutes({ path: '\*', method: RequestMethod.ALL }); // apply on all routes

}

}

## Section 4: Data Persistence - PostgreSQL and TypeORM

### Một số lưu ý trong chương này

**Lưu ý 1:** Khi tạo model table database, ví dụ tạo bảng Task:

@Entity()

export class Task{

@PrimaryGeneratedColumn('uuid')

id: string;

@Column()

name: string;

@Column()

description: string;

@Column({

type: "enum",

enum: Status,

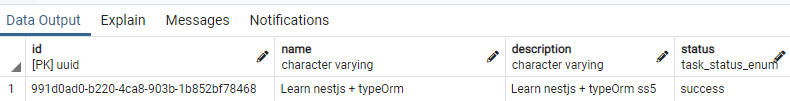
default: Status.INPROCESS

})

status: Status;

}

Có 4 trường được tạo, nếu trong cơ sở dữ liệu đã có dữ liệu của bản ghi có 4 cột này rồi thì sẽ hiển thị bình thường.

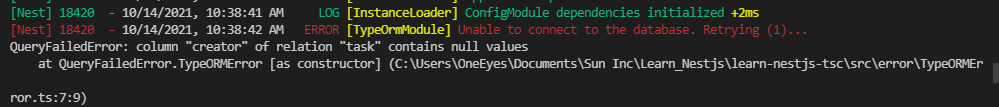


Nếu sau này sửa database, cụ thể ở đây là bảng Task, muốn thêm 1 trường là creator.

@Column()

creator: string;

Nếu khai báo như này thì mặc định khi save code, trường creator sẽ bằng null, sẽ dẫn tới lỗi.



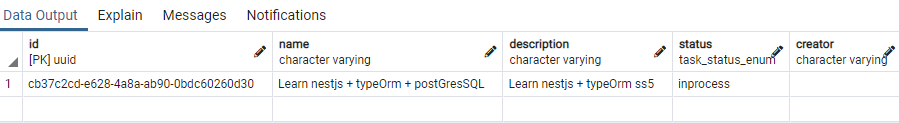
Để xử lý lỗi này thì khi thêm trường nào đó thì set giá trị default cho cột đấy luôn, đối với trường hợp này là tránh bị null. Sửa code như sau

@Column({

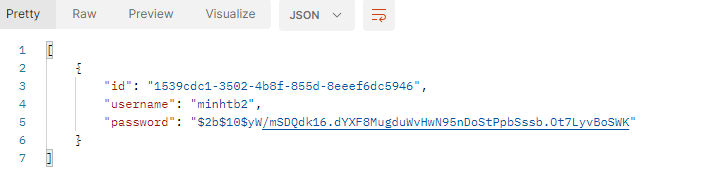
default: ""

})

creator: string;

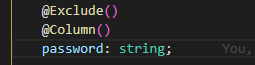


**Lưu ý 2:** Khi thiết kế bảng cơ sở dữ liệu khi truy vấn, ví dụ find danh sách user.. thì sẽ liệt kê hết các trường trong bảng user, cả password,….



Điều tất nhiên, quan trọng là các trường này không được lấy ra.

Để xử lý trường hợp này thao tác như sau.

Ở file tạo column cho bảng, ở trường password thêm thuộc tính @Exclude() 

Chuyển qua file controler: import UseInterceptors, ClassSerializerInterceptor của gói @nestjs/common

Tiếp theo ở hàm getIUsers thêm như sau:

 @UseInterceptors(ClassSerializerInterceptor)

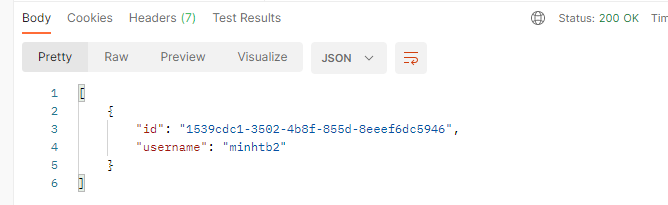
    @Get("/users")

    getUsers():Promise<User> {

        return this.authService.getUsers();

    }

Lưu lại , thì sau khi truy vấn trường password sẽ không còn hiển thị nữa

0

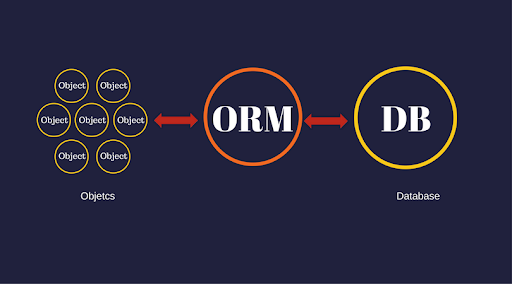
### CRUD cơ bản với DB postgres

* + 1. Prima

<https://docs.nestjs.com/recipes/prisma>

Để convert code viết bằng typeORM sang prisma : <https://www.prisma.io/docs/guides/migrate-to-prisma/migrate-from-typeorm>

Là một orm tức là ở giữa thực hiện ánh xạ CSDL sang các đối tượng trong các ngôn ngữ lập trình hướng đối tượng nestjs(TS),



## Section 5: Auth Part 1 – Authentication

### Khái niệm Authentication và Authorization



* **Authentication:** Xác định bạn có phải là user của hệ thống không. Có thể sử dụng một hoặc nhiều phương thức khác nhau để xác thực người dùng, tuỳ vào mức độ bảo mật của hệ thống: username+password, OTP,…..
* **Authorization:** Nếu đã là user của hệ thống, vậy bạn có những quyền gì trên hệ thống. Một trong những cách phổ biến nhất hiện nay là sử dụng JWT (JSON Web Token).

### Authentication

Cơ chế xác thực hầu như hệ thống nào cũng có, điển hình nhất là login vào hệ thống.

Để an toàn hơn khi xác thực thì đối với nodejs có thư viện passportjs là phổ biến, ngoài ra còn **jwt** được dùng trong project này.

Passport được tích hợp vào nest qua gói **@nestjs/passport**

**Lưu ý:**

Khi đăng ký tài khoản truy cập vào hệ thống, cần phải mã hóa password rồi mới lưu. Ví dụ sử dụng SHA256. Cài đăt như sau:

* Cài đặt gói bcrypt: yarn add bcrypt
  + ở hàm service signup

xử lý password client gửi như sau:

const salt = await bcrypt.genSalt();

      const hashPassword = await bcrypt.hash(password, salt);

Khi đó chuỗi **hashPassword** sẽ được lưu vào db.

#### 2.5.2.1Áp dụng cơ chế login bằng username/email + password (cách này là đơn giản nhất).

Vì password khi signup đã được mã hóa, nên để xác thực password ng dùng đã gửi lên lúc login có trùng với trong db hay không.

* Từ username client gửi lên truy vấn db lấy ra thông tin người dùng (có bao gồm cả password)
* Sử dụng bcrypt.compare(passwordclientgui, passwordcuauser\_db) để so sánh 2 password, tham số đầu tiên là password của user lấy ở bước 1, tham só thứ 2 là password client gửi lên

#### 2.5.2.2 Áp dụng cơ chế xác thực bằng jwt passport khi login

Mục đích để xác thực request gửi lên server có đúng/hợp lệ hay không

**Cài đặt**

Yarn add @nestjs/jwt @nestjs/passport passport passport-jwt

**Triển khai**

Mở file auth.modules.ts đã tạo trước đấy. import thêm 1 số thư viện

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

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

Xuống phần @Module chỗ imports, tiến hành thêm các module vừa import ở trên vào

@Module({

  imports: [

    ConfigModule,

    PassportModule.register({ defaultStrategy: 'jwt' }),

    JwtModule.register({

      secret: "WARE\_HOUSE",

      signOptions: {

        expiresIn: 3600

      }

    })

……..

Chuyển qua file **auth.service.ts ,**

Còn tiếp………..

## Section 6: Auth Part 2 - Task Ownership and Restrictions

Các loại quan hệ :

* [One-to-One](https://typeorm.io/#/one-to-one-relations)
* [Many-to-One and One-to-Many](https://typeorm.io/#/many-to-one-one-to-many-relations)
* [Many-to-Many](https://typeorm.io/#/many-to-many-relations)

## 2.9 Configuration

Hiện tại khi viết code, sẽ có một số trường dữ liệu càn được bảo mật, không được viết trong code. Do đó ta sử dụng Environment Variables (biến môi trường).

Các bước cài đặt:

* Cài đặt @nestjs/config: yarn add @nestjs/config
* Chỉnh sửa file app.module.ts, thêm đoạn code sau vào phần imports: []

ConfigModule.forRoot({ isGlobal: true, })

Định nghĩa **isGlobal** bằng **true**, thì cấu hình sẽ được áp dụng cho toàn module, không cần phải import cấu hình lại.

Mặc định viết như ở trên thì sẽ tự động đọc file .env ở thư mục gốc của project.

Nếu muốn thay đổi tên file hoặc vị trí đường dẫn tới file thì cấu hình thêm thuộc tính envFilePath.

* Tạo file .env trong thư mục gốc dự án:

#Sử dụng cho jwt

TOKEN\_SECRET=ware\_house

#Sử dụng cho kết đối db

DB\_HOST=localhost

DB\_PORT=5432

DB\_USERNAME=postgres

DB\_PASSWORD=postgres

DB\_DATABASENAME=task-management

* Sử dụng biến môi trường trong lúc cấu hình jwt:

Sửa file auth.module.ts

JwtModule.registerAsync({

      imports: [ConfigModule],

      inject: [ConfigService],

      useFactory: async (configService: ConfigService) => ({

        secret: configService.get("TOKEN\_SECRET"),

        signOptions: {

          expiresIn: 3600,

        }

      })

    }),

Vì cần tải config từ gói @nestjs/config không đồng bộ nên cần dùng **registerAsync** cần chạy. cần import Config Module và Inject ConfigService để dùng các phương thức ví dụ get,….

Thuộc tính inject là mảng các provider, nestjs sẽ phân giải và truyền làm đối số cho factory function trong quá trình khởi tạo.

Để đọc giá trị đọc file env.. gọi tới phương thức get và truyền key vào.

* Tương tự, cấu hình sử dụng biến môi trường lúc kết nối CSDL

Sửa file app.modules.ts

TypeOrmModule.forRootAsync({

      imports: [ConfigModule],

      inject: [ConfigService],

      useFactory: (configServer: ConfigService) => ({

        type: "postgres",

        autoLoadEntities: true,

        synchronize: true,

        host: configServer.get("DB\_HOST"),

        port: configServer.get("DB\_PORT"),

        username: configServer.get("DB\_USERNAME"),

        password: configServer.get("DB\_PASSWORD"),

        database: configServer.get("DB\_DATABASENAME"),

      })

    })

Sử dụng **forRootAsync** thay vì **forRoot,** chức năng của inject và useFactor tương tự như ở trên.

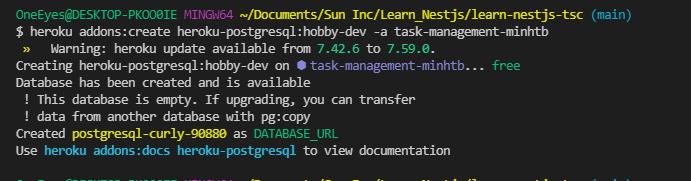
## 2.11 Deployment

### 2.11.1 Deploy trên heroku

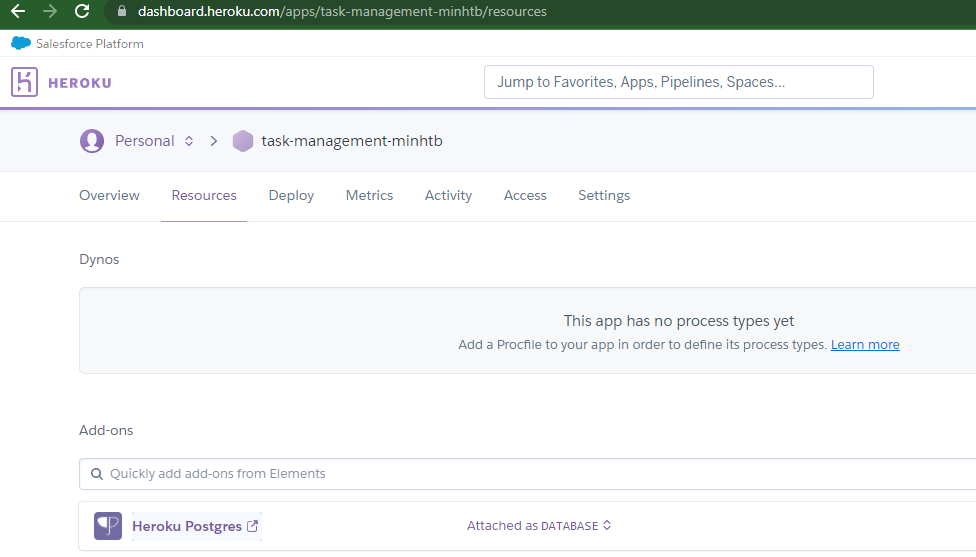
Các bước thực hiện:

1. Signup tài khoản trên heroku.com
2. Login web heroku và truy cập vào trang <https://dashboard.heroku.com/>
3. Click button Create new App: nhập tên app và chọn vị trí, và lưu
4. Trên máy tính cài heroku cli như sau: yarn global add heroku
5. Sau khi cài xong tiếp tục chạy : heroku addons:create heroku-postgresql:hobby-dev -a task-management-minhtb

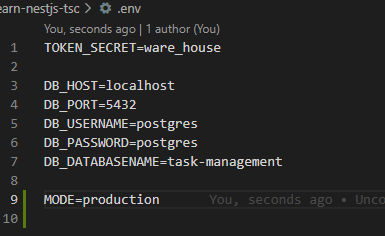
Với task-management-minhtb là tên app đã thiết lập ở bước 3



1. Sau khi create addOn xong f5 lại trang sẽ thấy Heroku Postgres trong tab resources



1. Click vào add-ons Heroku Postgres để có thể check 1 số thông tin khác
2. Sửa file .env thêm thuộc tính MODE=production, biến này để kiểm tra xem code đang chạy môi trường dev hay prod



1. Sửa code ở app.module.ts phần kết nối cơ sở dũ liệu như sau, thêm thuốc tính ssl+extra nếu ở bản product.

TypeOrmModule.forRootAsync({

      imports: [ConfigModule],

      inject: [ConfigService],

      useFactory: (configServer: ConfigService) => {

        const isProduction = configServer.get("MODE") === "production";

        return {

          ssl: isProduction,

          extra: {

            ssl: isProduction ? { rejectUnathorized: false } : null,

          },

          type: "postgres",

          autoLoadEntities: true,

          synchronize: true,

          host: configServer.get("DB\_HOST"),

          port: configServer.get("DB\_PORT"),

          username: configServer.get("DB\_USERNAME"),

          password: configServer.get("DB\_PASSWORD"),

          database: configServer.get("DB\_DATABASENAME"),

        }

      }

    })

1. Các bước khác xem thêm video udemy.

**Lưu ý:** rejectUnathorized set bằng false để không cần cấu hình ssl

extra: {

            ssl: isProduction ? { rejectUnathorized: false } : null,

          },

## 2.12 Unit test

# **Docker + pgadmin**

## Cài đặt

Cài đặt docker theo link: <https://www.docker.com/products/docker-desktop>

Cài đặt pgAdmin để quản lý database theo link: <https://www.pgadmin.org/download/>

## Khởi chạy postgress

### Khởi chạy bằng docker

Mở terminal chạy lệnh sau:

docker run --name postgres-nest -e POSTGRES\_PASSWORD=postgres -d postgres

Tham khảo ở link: <https://hub.docker.com/_/postgres> (mục **start a postgres instance**)

### Khởi chạy trên window

**Note:**

<https://stackoverflow.com/questions/33907835/docker-error-cannot-delete-docker-container-conflict-unable-to-remove-reposito>

# **TypeORM**

## Khái niệm

TypeORM là một ORM có thể chạy trong các nền tảng **NodeJS**, Browser, Cordova, PhoneGap, Ionic, React Native, NativeScript, Expo và Electron và có thể được sử dụng với TypeScript và JavaScript (ES5, ES6, ES7, ES8).

Phân biêt Data mapper và Active Record

<https://typeorm.io/#/active-record-data-mapper/what-is-the-data-mapper-pattern>

<https://github.com/typeorm/typeorm/blob/master/docs/select-query-builder.md>