NestJS

# Introduction to NestJS & Pre-requisites

## What is NestJS

### NestJS

* NestJS is an open source framework for **building efficient, scalable Node.js server-side applications**.
* NestJS was built with full support for **TypeScript** although it still supports JavaScript.
* It combines elements of Object Oriented Programming, Functional Programming and Reactive Programming.

### Under the Hood

* Under the Hood, NestJS makes use of robust HTTP server frameworks. By default, it uses **Express.js**.
* It provides a level of abstraction above these frameworks which makes its incredibly joyful to code in TypeScript. However it still exposes the original APIs directly to the developers. This allows us to for example use Express specific libraries even when using NestJS.

### The Philosophy behind NestJS

* The rise of popular Web technologies such as Angular, React and Vue has massively improved the development experience on the front end.
* However while Node.js (for the service side) has plenty of great libraries, but none of them effectively solves the main problem of **architecture**.
* **NestJS provides an out of the box application architecture** which allows developers and teams to create highly testable, scalable, loosely coupled and easy to maintain applications.

### The NestJS CLI

* NestJS CLI is a command line interface tool that helps you to initialize and develop your applications.
* It has many benefits – from scaffolding a project, to building a well-structured application.
* It is possible to generate a project and generate schematics in a project using the CLI. This leaves almost no room for mistakes in terms of project structure.
* Using the CLI to generate projects and schematics can save you a lot of time writing boilerplate code.

### NestJS Documentation

* The NestJS documentation is well maintained and contains a lot of information about techniques, recipes and real examples.
* Official Website: <https://nestjs.com/>
* NestJS Documentation: <https://docs.nestjs.com/>

## Installing NestJS CLI

* NestJS CLI can help you by first of all making you really productive and fast.
* You could use the CLI to generate schematics such as module, controller and even the entire NestJS application or your entire project and this will save you a lot of time because well you don't have to write so much boilerplate code anymore and you get an entirely scaffold application or scaffold schematics that you can just start working.
* It also applies some best practices for you such as naming of files and positioning of files within the project folder.
* Install NestJS CLI globally,

npm install -g @nestjs/cli

* To verify if the NestJS CLI is properly installed, run below command

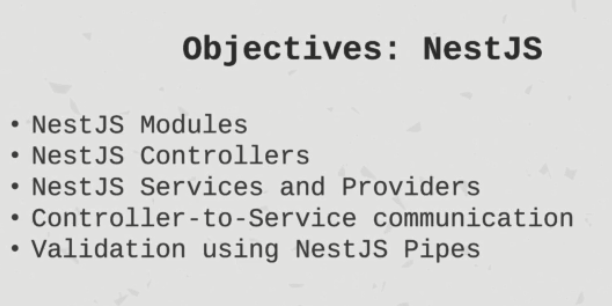
nest --version

* NestJS CLI Documentation: <https://docs.nestjs.com/cli/overview>

# Building REST APIs

## Project Overview

* Objectives



## NestJS Project Structure

* To create NestJS project structure, execute

nest new project-name

* This will default project structure and few files for your app.
* Note: Depending of the version of NestJS, the automatically generated files may be added, removed or renamed.
* tslint.json – This is the Jason configuration for typescript linting. Linting basically helps us follow some certain guidelines on how to write our script code. So this is the default configuration that is provided by NextJS.
* tsconfig.json – This is the file that tells the typescript compiler how to compile our code.
* tsconfig.build.json – This is extension of normal tsconfig.json. And that is specific for when we build our application for production. It has some exclusions for certain folders in module folder, test, dist folders and any file ending with .spec.ts.
* package.json – This is a classic file in any npm project. It has dependencies, dev dependencies, scripts and general project information. NestJS provides us some predefined scripts ready to be used.
* nodemon.json – specific configuration file for project.
* nodemon.debug.json – the predefined script start:debug uses this file.
* src folder – This is where we're gonna write most of our code.
* src/main.ts – entry point of our application. This is where server starts listening.
* src/app.module.ts – is the root module of our application. (A NestJS Module)

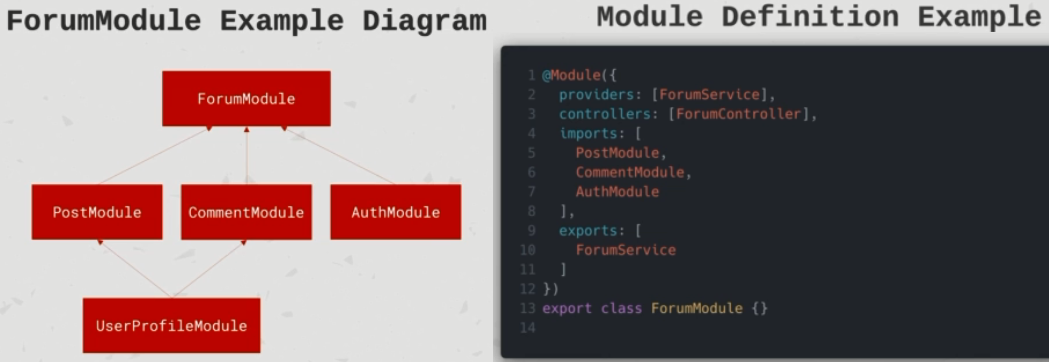
## Introduction to NestJS Modules

### What is a module

* Each application has at least one module – the root module. That is the starting point of the application.
* Modules are an effective way to organize components by a closely related set of capabilities. For example a module per feature.
* It is a good practice to have a folder for a module containing the module’s components.
* Modules are **singletons**. Therefore one module can be imported by multiple other modules.

### Defining a Module

* A module is defined by annotating a class with **@Module** decorator.
* The decorator provides metadata that NestJS uses to organize the application structure.
* @Module properties –
  + providers – is an array of providers/services to be available within the module via dependency injection.
  + controllers – properties an array of controllers to be instantiated within the module.
  + exports – property is an array of providers or modules to export to any other modules.
  + imports – property is a list of modules that are required by this module. Any exported provider by these modules will now be available in our module via Dependency Injection.
* When developing NestJS applications, you should try to treat your modules in an isolated way as much as you can.
* Example –



### Creating a Module

* CLI allows us to generate schematics, and one of those schematics is a module.
* Here is the command –

nest g module <module\_name>

* g means generate.

module is the schematic that we want to create.

<module\_name> is the name of the schematic which is also the path of the schematic relative to the src folder.

* Executing this command will create a new module with the specified name and will annotate it with @Module decorator and also updates the AppModule (which is the root module of our application) by adding the new module into the ‘imports’ array of AppModule.
* So NextJS schematics really save us quite some time, also in terms of writing boilerplate code which means more time for us being productive.

## Introduction to NestJS Controllers

### What is NestJS Controller

* Controllers are responsible for handling incoming **requests** and returning **responses** to the client.
* Controllers are bound to a specific **path**. For example “/tasks” for the task resource.
* Controllers contain **handlers** which handle **endpoints** and **request methods** such as GET, POST, DELETE, etc.
* Controllers can take advantage of **dependency injection** to consume providers within the same module.
* Controllers are really just a simple concept. There are some layer of **abstraction** between the client and some things happening in the background.

### Defining a Controller

* Controllers are defined by decorating a class with **@Controller** decorator.
* The decorator accepts a string, which is the **path** to be handled by this controller.
* For example, if we have a class named TasksController which is decorated with @Controller decorator with path as “/tasks” provided to it as a string. So any requests incoming to “/tasks” within the application is going to be handled and routed to this controller.

### Defining a Handler

* Handlers are simply **methods** within the controller class decorated with decorators such as **@Get, @Post, @Delete,** etc.
* So each handler function corresponds to a specific HTTP method.

### Creating a Controller

* Here is how we can create controller using NestJS CLI.

nest g controller <path> --no-spec

e.g. nest g controller tasks --no-spec

* g for generate

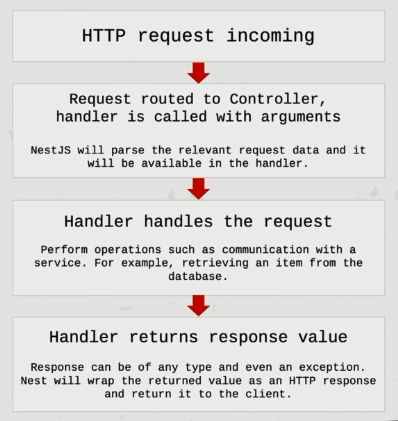
controller for the controller schematic.

<path> for the path in which the controller will be created relative to the src folder.

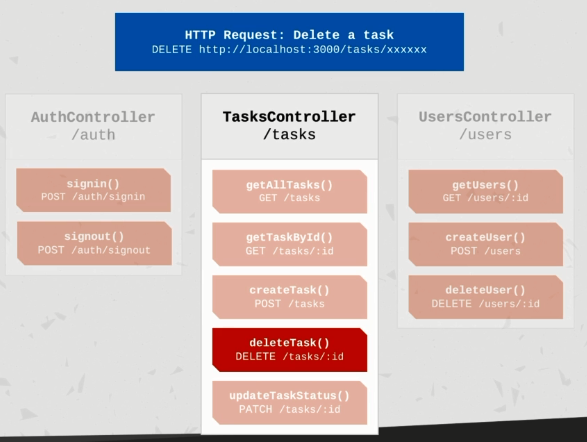
--no-spec flag to tell the CLI to not generate a spec file for us because we're not writing any unit tests right now.

* Executing this command will create new controller file in the specified path (folder under src) and will update module file in that path. E.g. nest g controller tasks --no-spec here, it will create tasks.controller.ts file inside src/tasks folder and will annotate it with the @Controller decorator. Also will update existing tasks.module.ts file by adding this controller into the ‘controllers’ array of TasksModule.

### Flow (Request -> Response)



Flow with example –



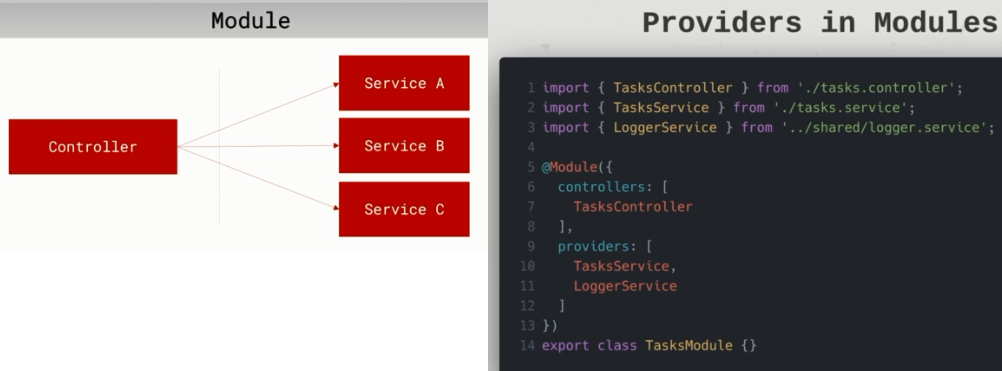
## Introduction to NestJS Providers and Services

### NestJS Providers

* Providers can be injected into constructors if decorated as **@Injectable** via dependency injection.
* Providers can be of plain value, classes, synchronous and a synchronous factories etc.
* Providers must be provided into a module for them to be usable. So they must be defined in the ‘providers’ array of a module.
* Providers can be exported from a module and then be available to other modules that import it.

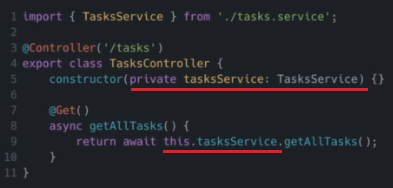
### Service

* **Services are defined as providers**. Now it's important to note that **NOT all providers are services**.
* Services are a common concept within software development and are not exclusive to NestJS, JavaScript or back-end development.
* When services are wrapped with the @Injectable decorator and are provided to a module, they act as singleton services. That means the same instance will be shared across the application acting as a single source of truth.
* Services are the main source of business logic. For example a service will be called from a controller to validate data, create an item in the database and return a response.
* Eg.



### Dependency Injection in NestJS

* Any component within the NestJS ecosystem can inject a provider that is decorated with @Injectable decorator, if they're in the **same module**.
* We define the dependencies in the **constructor** of the class. NestJS will take care of the injection for us and it will then be available as a class property.
* E.g.



### Creating a Service

* Here is how we can create service using NestJS CLI.

nest g service <path> --no-spec

e.g. nest g service tasks --no-spec

* g for generate

service for the service schematic.

<path> for the path in which the service will be created relative to the src folder.

--no-spec flag to tell the CLI to not generate a spec file for us because we're not writing any unit tests right now.

* Executing this command e.g. nest g service tasks --no-spec will create tasks.service.ts file under folder src/tasks and will annotate it with the @Injectable decorator. Also it will also update the tasks.module.ts file by adding this newly created service in the ’providers’ array of TasksModule.

## Creating a Model

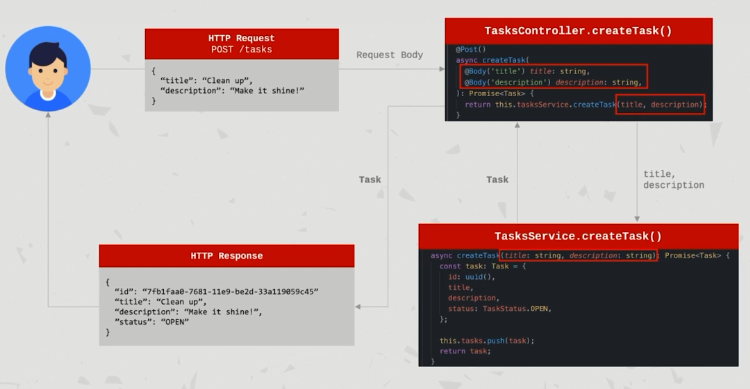
* Model defines the shape of the business object which we are dealing with. E.g. Task, User, etc.
* We can define a model either as a class or as an interface.
* Interfaces are a typescript concept that simply enforces the shape of an object upon compilation. Therefore after compilation, interfaces are not preserved as interfaces anymore.
* Classes however already exist in JavaScript since ES6. Therefore even post computation our classes will be preserved. Classes are useful when you want to create objects based on a blueprint and add some self-contained functionality to them using methods for example.
* And if you're not sure, I'd suggest you to start with an interface and then upgraded to a class if you need to because that's quite easy to do.

## Extracting information from HTTP Request Body

* Two ways to extract information from request body
* Using @Body decorator to get entire request body.
  + So when an HTTP request comes in, NestJS will make the request body available to use on the declared parameter.
  + E.g. createTask(@Body() body) {...}
* 2. Using @Body decorator with specific parameters
  + So when an HTTP request comes in, NestJS will bind the mentioned parameter from request body to the declared parameter.
  + E.g. createTask(@Body('title') title: string, @Body('description') description: string ) {..}

## Introduction to Data Transfer Objects (DTOs)

### Understanding the problem



* If you see above example, if in future the input to creating a task changes (adding new parameters, changing or removing existing parameters), we would have to TaskController, TaskService and any other places wherever it is used, in standalone.
* This can get really messy and hard to maintain and you also lose the sense of reliability in the shape of data.
* We do not have a unified way to define what the data looks like throughout the process.

### Solution: Use DTOs

* Wikipedia says a data transfer object is an object that carries data between processes.
* In stackoverflow, it says a data transfer object is an object that is used to encapsulate data and send it from one subsystem of an application to another.
* In the official NestJS application, it says a DTO is an object that defines how data will be sent over the network.
* DTO concept is not specific to NestJS.
* DTOs can result in more bullet proof code as it can be used as a typescript type.
* DTOs do not have any behavior except for storage, retrieval, serialization and deserialization of its own data.
* Sometimes DTOs result in increased performance. It depends on the application although it's quite negligible in small applications.
* DTOs can be useful for us if you want to validate data.
* Important Note: **DTO is NOT a model definition**. It can be confusing and sometimes it is confused as one. DTOs define the shape of data for a specific case. For example, the data that we expect when creating a task.
* DTOs can be defined using an interface or a class.

### Choosing a class or interface for a DTO

* The recommended approach as per NestJS documentation is to **use classes**.
* The reason is that interfaces are a part of TypeScript and therefore are not preserved after compilation.
* Classes allow us to do more. And since they are a part of JavaScript, they will still be preserved after a compilation. This will be useful for us in the future when using pipes.
* NestJS cannot refer to interfaces during runtime but can refer to classes.
* TLDR: Classes are the way to go for DTOs.

### Important Note about DTOs

* Using DTOs is not mandatory. You can still develop applications without using DTOs.
* However the value that DTOs add makes it worthwhile to use them when applicable.
* Applying the DTO pattern as soon as it is possible will make it easy for you to maintain and refactor/expand your application.

# Validation and Error Handling

## Introduction to NestJS Pipes

* Pipes operates on the arguments to be processed by the route handler just before the handler is called.
* Pipes can perform data transformation or data validation.
* Pipes can return data – either original data or modied data which will then be passed on to the route handler.
* Pipes can throw exceptions. Exceptions thrown in the pipe will be handled by NestJS and then parsed into an error response.
* Pipes can be asynchronous.

## In-built pipes in NestJS

* NestJS ships with useful pipes within the @nestjs/common module.

### ValidationPipe

* This one validates the compatibility of an entire object against the class (goes well with DTOs).
* If any property cannot be mapped properly for example mismatching type, validation will fail.
* This is a very common use case. Therefore having a built in validation pipe is extremely useful for us.
* The ValidationPipe uses the DTO which is also the type of the parameter. Then, ValidationPipe uses the class-validator package decorators as rules, and validates the data against those rules.

### ParseIntPipe

* By default, arguments are of type String. This pipe validates that an argument is a number and if so the argument will then be transformed into a number of type and passed onto the handler.
* This is especially useful when you expect a number and don't want to parse it manually every time.

## Custom Pipes in NestJS

* We can also implement our own custom pipes.
* Pipes are classes that are annotated with the **@Injectable** decorator.
* Pipes must implement the **PipeTransform** generic interface. Therefore every pipe must have a **transform()** method. This is the method that will be called by NestJS to process the arguments.
* The transform() method accepts two parameters -

value – which is the value of the processed arguments and

metadata – which is optional. That is an object containing metadata about the argument.

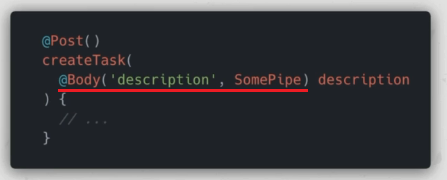
* Whatever is returned from the transform() method will be passed on to the route handler.
* Exceptions that are thrown will then be sent back to the client in the shape of an HTTP error response.
* Pipes can be consumed in different ways.

## Consuming/Using Pipes

* There are **handler level pipes** which are defined at the handler level via the **@UsePipes** decorator. Such pipe will process all parameters for the incoming requests.



* Then there are the **parameter level pipes** which are defined in the parameter level. Only that specific parameter for which the pipe has been specified will be processed.



* Finally there are **global pipes**. These are defined that the application level and will be **applied to any incoming request within the application**.



### Parameter-level VS Handler-level pipes

* **Parameter-level** pipes tend to be slimmer and cleaner. However they often result in extra code added to handlers and this can get messy and hard to maintain.
* **Handler-level** pipes require some more code but provide some great **benefits** –
  + Such pipes do not require extra code at the parameter level.
  + They are easier to maintain and expand. If the shape of the data changes, it is easy to make the necessary changes within the pipe only.
  + The responsibility of identifying the arguments to process is shifted to one central file – the file in which the pipe is defined
  + Finally handler-level pipes promote the usage of DTOs which is a very good practice.

## Example Pipe Usage



## Third party validation package to be used along with ValidationPipe

* Install and use these 2 packages –

npm install class-validator class-transformer --save

* Documentation: <https://docs.nestjs.com/pipes#class-validator>
* GitHub: <https://github.com/typestack/class-validator>

# Data Persistence: PostgreSQL and TypeORM

## Introduction to ORM and TypeORM

### ORM (Object Relational Mapping)

* Object Relational Mapping or ORM is a technique that lets you query and manipulate data from a database using an object oriented paradigm.
* There are many ORM libraries that allow developers to communicate to the database using their preferred programming language rather than sending plain queries directly.

#### Pros

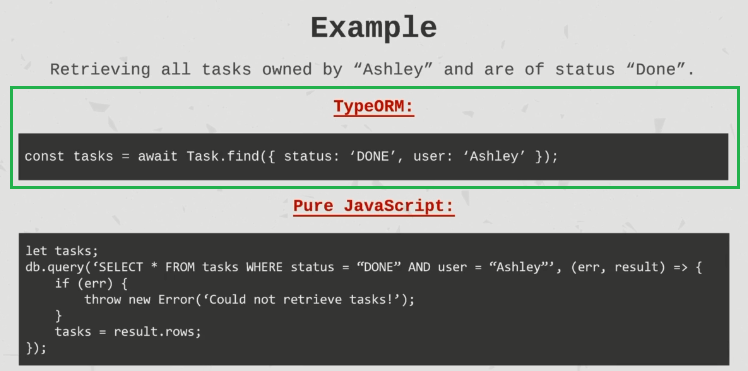
* Writing the data model in one place is easier to maintain. There is less repetition.
* Lots of things are done automatically. Such as database handling, better typesetting (data types) and relations.
* SQL syntax is very easy to learn about but very hard to master. So there is no need to write SQL syntax anymore. You can use your natural way of coding using your preferred language.
* Database abstraction – You can change the database type whenever you wish. And a good ORM will support multiple databases without you having to change the code.
* It leverages Object Oriented Programming. Therefore things like inheritance are easy to achieve.

#### Cons

* You have to learn it. And ORM libraries are not always simple.
* Performances all right but it's easy to neglect. To overcome this, we need to use the Query Builder in TypeORM which is a method that helps us improve the performance for certain operations.
* ORMs make it easy to forget or actually never learn what's happening behind the scenes which can lead to a variety of maintainability issues.

## TypeORM

* TypeORM is an ORM library that can run in Node.js and be used with TypeScript or JavaScript.
* It helps us define and manage entities, repositories, columns, relations, replications, indices, queries, logging and so much more.
* Example –



* Official website: <https://typeorm.io/>

## Connecting NestJS to Database using TypeORM

### Installing required packages

* Luckily NestJS provides us with a TypeORM module that makes it super easy to plug it into our application.
* Install the dependency as –

npm install @nestjs/typeorm typeorm –save

typeorm is the actual TypeORM npm module.

@nestjs/typeorm is a specific bridge that NestJS has created for working with TypeORM.

* Also we need to the install the database driver corresponding to the database we are using. TypeORM will use the driver to interact with our database.  
  E.g. to install DB driver for PostgreSQL –   
  npm install pg –save

### Initialize the connection in our application

* We need to provide different DB configuration parameters in order to initialize DB connection from our application.
* The DB configuration includes parameters like DB type, host, port, username, password, database, etc. and entities.
* There are multiple ways of **configuring** the database connection

1. Using a static JSON File as a configuration.
2. Providing the configuration data as an object.
3. Providing the configuration data asynchronously from a service.

* Imp Note: When we start the application in production, the TypeScript code is the transpiled into JavaScript. Therefore, our entity files (\*.entity.ts) will not be picked up by TypeORM. So we need to use both .js and .ts entity files as -   
   entities: [\_\_dirname + '/../\*\*/\*.entity.{js,ts}']
* No matter of which way we choose to define configuration, we then need to provide this configuration to the AppModule, so that on server startup the connection will be established.
* E.g.

@Module({

  imports: [TypeOrmModule.forRoot(typeOrmConfig), TasksModule],

})

export class AppModule {}

## Creating TypeORM Entity

* In TypeORM, we define entities that represent tables and those entities can carry logic under the hood for us that makes it's very easy to work with a database without having to manually write queries all the time.
* Entity file naming convention: <name>.entity.ts
* Each entity class must be annotated with **@Entity** decorator from typeorm package.
* Each entity class must extend **BaseEnity** class from typeorm package.
* Primary key should be annotated with **@PrimaryGeneratedColumn** or **@PrimaryColumn** decorators. **@PrimaryGeneratedColumn** just tells TypeORM that this is a primary key column and that the ID should be automatically generated and incremented whenever we create a new object of this entity class.
* Fields can be mapped to DB columns by annotating the field with **@Column** decorator.

## Creating Repository

* By using repositories for our entities we apply that repository pattern.
* Further reading – <https://github.com/typeorm/typeorm/blob/master/docs/custom-repository.md>
* In short the repositories can manage entities in an encapsulated manner.
* In our repository we can still perform same operations that we would normally perform on the Entity class directly but we can also add more custom logic. And this will become very useful for us in our application.
* We end up encapsulating heavy logic related to persistence layer to our database and we also end up removing code from our service which results in shorter methods into service and code that is easier to understand.
* The repository we're going to create is going to be called from the service whenever we need to perform database operations related to that entity.
* Repository file naming convention: <name>.repository.ts
* Each Repository class must be annotated with **@EntityRepository(<entiry-class-name>)** decorator from typeorm package.
* Each Repository class must extend **Repository<entiry-class-name>** class from typeorm package.
* Extensing the **Repository<entiry-class-name>** class from typeorm package gives us access to many usefule methods like create(), update(), delete(). findOne(), findByIds(), find(), createQueryBuilder() and many more.
* Once repository file is created, we need to add it to the ‘imports’ array of the module where we want to use it. E.g.

import { TypeOrmModule } from '@nestjs/typeorm';

import { TaskRepository } from './task.repository';

@Module({

  imports: [

  /\*

  Passing array of repositories that we want to include   
 in this Task Module.

  Since its not AppModule, so using forFeature() instead of forRoot().

  Below line makes the TypeOrmModule which comes from NestJS

  include this TaskRepository instance injectable independency injection   
 throughout this module. \*/

    TypeOrmModule.forFeature([TaskRepository]),

  ],

  controllers: [TasksController],

  providers: [TasksService],

})

export class TasksModule {}

* Finally we can now just inject the repository in say service class (by adding into constructor arg) and use it in that file.

E.g.

export class TasksService {

// inject TaskRepository instance

constructor(

 @InjectRepository(TaskRepository) private taskRepository: TaskRepository,

) {}

* Creating a repository allows for a place to contain our database-interaction related logic. This way, we can keep our services cleaner. Service will still be able to handle business logic, but database-related logic will be handled by the repository.

# Authentication – Setting up JWT/passport.js

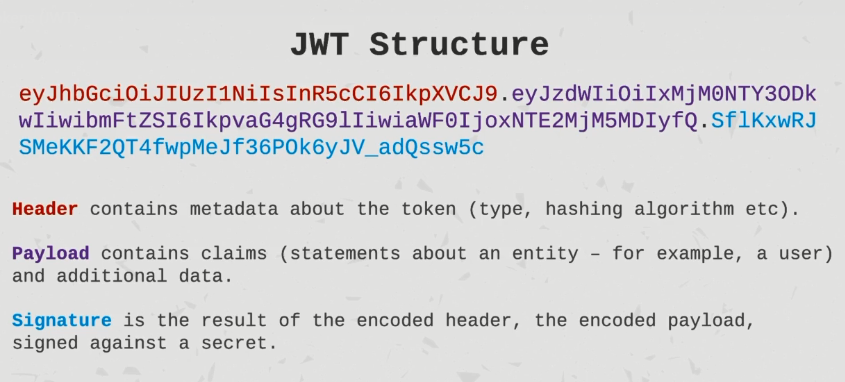
## Storing Passwords

* Of course we should never store passwords as plain text. We should always encrypt it.
* **Salt** is used as some additional inputs to a one way hashing operation to safeguard the password in storage.
* If we just encrypt only password string with say SHA-256 hashing and save the generated hash in database. Then lets say if our database is compromised, then through online tools like encryption-description, our encrypted hash can be decrypted. We can prevent this by using **salts** when we generate our hashes for our passwords.
* So our application is actually going to hash something like salt+actual password. So it's going to prefix the password with some random unique string and then after we encrypt this password our hash result is going to be completely different. And if the attackers get access to this hash they will try to decrypt it but they will not be able to find out our password because well even though our password is 1 2 3 4 5 6 is actually encrypted with the salt and this salt is going to be unique per user.
* This is very important unique per user. Sometimes people tend to have an application wide hash and this is not good or unique per user. That's the way to go.
* What do we prevent here by using hashes and unique salts per user is first of all if our database is ever hacked, the password will not be plaintext. That's already a nice thing, the hackers can only see the passwords and the salt but it is a one way hash so nothing is exposed. Use a per user salt so that hackers cannot use rainbow table (A rainbow table is a database that is used to gain authentication by cracking the password hash) and we never log or save the user's password when signing up, we simply hash it in storing the database.

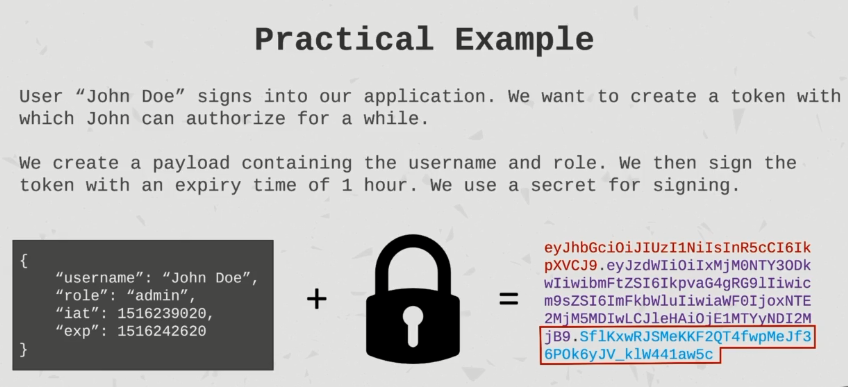
## Introduction to JSON Web Tokens (JWT)

* It is an open source industry standard.
* It is usable for authorization or secure exchange of information between parties.
* It’s used to verify that the sender is who it/she/he claims to be and it is signed by an issuer, the one issuing the token using a secret or a keypair using some cryptographic algorithms such as HMAC, RSA or CBSA.

### JWT Structure



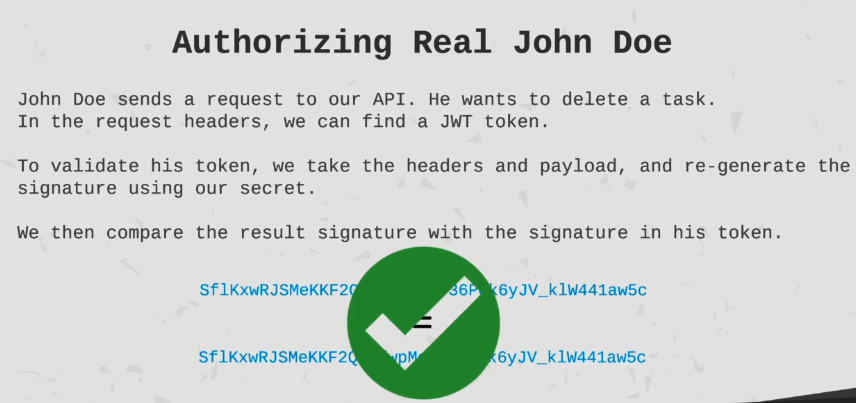
### Practical Example



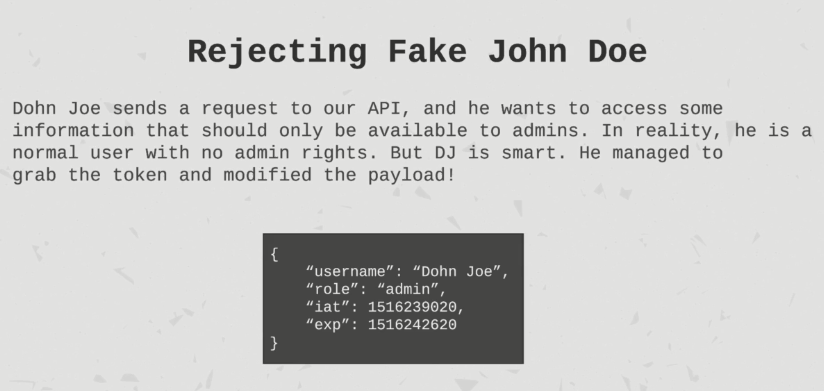
* The signature which is the last part is the result of the header and payload processed against a secret (which only we know) after a cryptographic hash.
* If we are the only ones who know the secret, nobody can fake the signature.

### Authorization using JWT

#### Authorizing Real User



#### Authorizing Fake User





### Important Notes about JWT

* JSON tokens can be decoded by anyone so they should not contain any sensitive information such as passwords.
* It is useful for front end applications to use these tokens for example in a front end application. We could use the token that says a user is an admin to show a certain button.
* JWT token should ideally be short lived. For example the user’s name is John Doe. It could be that John Doe changed his user name 10 minutes afterwards. That’s why you want the JT with tokens to be relatively short lived so that you generate a new token every once in a while that contains the most recent information about the user.

## JWT Authentication

* Passport.js is an authentication middleware for Node.js applications.
* Passport.js supports multiple strategies. And one of them is JWT token strategy.
* After configuring Passport.js, it's going to help us implement authentication in a very easy way. It’s doing a lot for us under the hood, authenticating based on the token, requests and retrieving the user, injecting it into our request so that we don't have to do that manually every time.

### Installing required packages

* npm install --save @nestjs/jwt @nestjs/passport passport passport-jwt

@nestjs/jwt is wrapper module from NestJS for working with JWT.

@nestjs/passport is wrapper from NestJS for implementing passport middleware

passport is the actual passport.js library

passport-jwt for configuring passport.js to use JWT.

### Configuring/Setting up JWT and Passport in our application

* Here we need to setup JWTModule and PassportModule in our Auth Module.
* E.g.

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

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

@Module({

  imports: [

    // configuring Passport to take JWT tokens and use that for authenticating the user

    // passport supports multiple strategies. Here we are going to use JWT

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

    // configuring JWTModule

    JwtModule.register({

      secret: 'TopSecretTaskManagement', // for signing the token

      signOptions: {

        expiresIn: 3600, // 1 hr

      },

    }),

// some more code

})

export class AuthModule {}

### Signing JWT Token upon Authentication

* The JwtModule exports service called JwtService which we can then inject and use it to create a token, signing a token, etc.
* After successful authentication, we can create the token and send it to client.
* E.g.

// 1. define payload

const payload: JwtPayload = { username };

// 2. sign the payload using secret key to generate the token

const accessToken = await this.jwtService.sign(payload);

// 3. return token

return { accessToken };

* Now at this point the front end application that is consuming the REST API is going to have this token and pass it on with every request (as an Authentication Bearer token) where it has to be authenticated.

## Setting up the JWT Strategy for Authorization (required by passport.js)

* When using passport.js, we have to define which strategy we are going to use. There are different strategies that can be used with passportjs e.g. jwt, OAuth, Google Auth, etc.
* After successful authentication, we need to set up JWT strategy for authorization, so that we could store the ‘user’ object for further use within the protected route.
* **validate()** method must exist when defining a strategy.
* Once we setup the JWT strategy for authorization, for every controller in the application we could basically guard by using **@UseGuards(AuthGuard())**
* We could do it at controller level so that it will be applied to all the endpoints in that controller or more specifically at endpoints level to apply authorization at specific endpoints.
* E.g. ***jwt.strategy.ts***

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

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

import { InjectRepository } from '@nestjs/typeorm';

import { Strategy, ExtractJwt } from 'passport-jwt';

import { JwtPayload } from './jwt-payload.interface';

import { User } from './user.entity';

import { UserRepository } from './user.repository';

/\*

  Setting up JWT passport strategy for authorization

  Here Lot of things are done under the hood by @nestjs/passport using passport-jwt

  Using @Injectable because we are going to expose it as a service,

\*/

@Injectable()

export class JwtStrategy extends PassportStrategy(Strategy) {

  constructor(

    @InjectRepository(UserRepository) private userRepository: UserRepository,

  ) {

    super({

      // define how to extract JWT. Here from Authorization Bearer Token

      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),

      // to verify the signature of the token that is extracted from the request. (Internally done by Passport)

      secretOrKey: 'TopSecretTaskManagement', // same secret as defined in auth.module.ts

    });

  }

  /\*

    This is an important method which must be present here

    This payload is ALREADY VERIFIED at this point.

    So first passport.js is going to verify the signature using the secret that we provided above (super()).

    If it's not valid, it's going to be throw an error.

    If it is valid,  it's going to call this validate() method with the payload.

  \*/

  async validate(payload: JwtPayload): Promise<User> {

    /\*

        Here we're going to do some validation and at the end whatever we return

        from here is going to be injected into the request of any operation that is guarded with authentication.

    \*/

    const { username } = payload;

    const user = await this.userRepository.findOne({ username });

    if (!user) {

      throw new UnauthorizedException(

        'User is not authorized for this operation.',

      );

    }

    return user;

  }

}

## Creating Custom Parameter Decorator

* Parameter decorator is the one which can be applied on to function parameters.
* It is very easy to write a Parameter decorator in NestJS.
* Below is an example decorator which extracts the user from the request object so that it can be used easily plugged to some endpoint’s parameter for further use.

***get-user.decorator.ts***

import { createParamDecorator, ExecutionContext } from '@nestjs/common';

import { User } from './user.entity';

/\*

Creating custom decorator to extract the user from the request object

and we're going to use that quite a bit.

Ignoring data because we are not going to call this decorator with any data.

whatever we return from this function is going to be set with the parameter that is decorated with this decorator.

This will be called as @GetUser() in the parameter

\*/

export const GetUser = createParamDecorator(

  (data, ctx: ExecutionContext): User => {

    const req = ctx.switchToHttp().getRequest();

    // user is present in the requst because after successful authentication,

    // we are storing the user in the request. See JwtStrategy.validate() function

    return req.user;

  },

);

In say ***auth.controller.ts***

@Post('/test')

  @UseGuards(AuthGuard())

  test(@GetUser() user: User) {

    console.log(user);

}

# Authorization – Task Ownership

* Less information you give to somebody who didn't care about this information the better your security will be.
* For example if you tried to access a GitHub repository on your browser or a private repository that you don't have access to, you're gonna get a 404 Not Found error instead of 401 unauthorized error because github is not gonna give you any hints on whether this repository exists or not.

# Logging

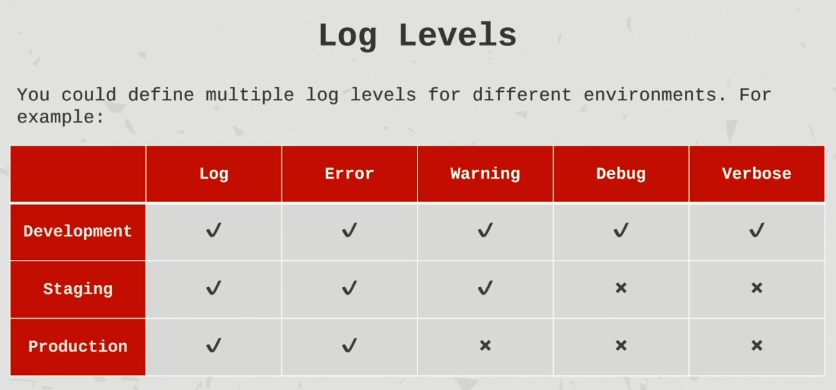
* We have so many different operations. Some are very important. Some are less important. Some are destructive for users and some are not. If something goes wrong you could use logging to help us be aware of that.
* You could also provide some useful information that will help us find the problem and the cause for what went wrong. So logs can be useful for us as developers but also for people who operate and use our software.
* There are different types of logs and these types can be useful for different situations and sometimes even different people.

## Types of Logs

* First type would be the **general log**. General purpose logging of important information. So for example when we start our application we could log “Application started on port 3000” or whatever the port is.
* Then there is these **warning logs**. So unhandled issues that are not fatal or destructive. For example when we manage to save a task in database but only after two attempts. So this is something we wanted to look into. But it is not destructive for the user or for the application.
* Nest is the **error type of log** and these are unhandled issues that are fatal or destructive. So for example when we failed to save a task in the database even though all the validation passed and the task is good to be saved.
* The next step is the **debug** log. Useful information that can help us debug the logic in case of an error/warning. It is intended for developers. For example we could log the state of the application during some operation.
* Nest is the **verbose** log. And this is information providing insights about the behavior of the application. And this is intended for operators. For example the support and usually this is too much information for the average user who looks at the logs.

## Log Levels

* You could define multiple log levels for different environments for the application.
* E.g. Below is just an example table. You should of course use what works for you, for your project and for your team.



## Applying logging in NestJS

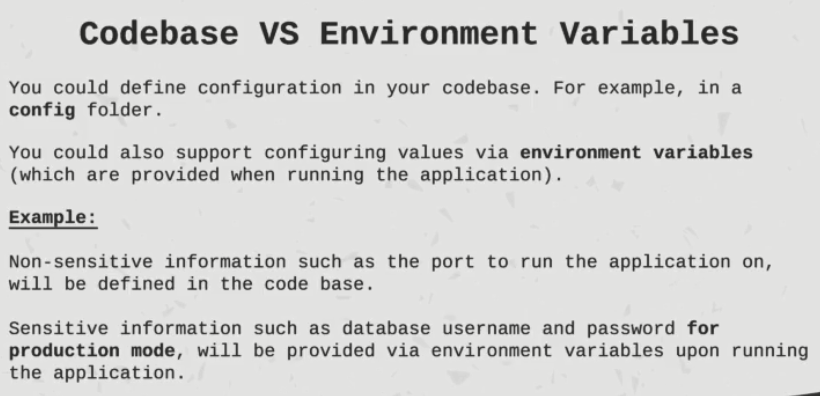
* NestJS ships with a logger which is a part of the @nestjs/common module.
* When you develop a real world application, you should probably apply logging per feature as you develop that feature.
* Here is official NestJS documentation: <https://docs.nestjs.com/techniques/logger>

# Pro-Production Configuration

## Introduction to Configuration

* Configuration is a central way of defining values that are loaded upon starting the application. These values should not be changed on the fly during runtime. They are static values.
* Configuration can be done per Environment – development, staging, production etc.
* Configuration can be defined in the code base so it's useful if you work with multiple developers using version control such as Git. Your configuration is then tied to your codebase and should always work supporting the code that consumes it.
* Configuration can be defined in many ways using JSON, YAML, XML, Environment variables, etc. And you could also use custom solutions or open source libraries.

### Codebase VS Environment Variables for configuration



## Configuration management setup

* There is an NPM package called ‘config’ which is of huge help in setting up env specific configurations and reading it.
* To install the package –

npm install config --save

* This ‘config’ package expects that you have a ‘config’ folder at root level where you have default as well as env specific configurations.
* You can define the configurations in YML, XML, JSON, properties file, etc.
* YML is a great way to define nested env variables which makes it easy to read as an object from ‘config’ package.
* We can have default.yml file which has common configuration for all environments like db type, db name, db port, server port. And then we can overwrite this and/or add more config variables in env specific yml files. default.yml is available in all environments.
* But irrespective of using third party packages like ‘config’, we must never store production specific sensitive information like JT secret, DB username and password into codebase. For sensitive information, we should use environment variables.
* How does ‘config’ (from config package) knows which environment we're running on? The answer is the ‘config’ package use the NODE\_ENV environment variable to know in which env it is running in.
* Actually this NODE\_ENV environment variable is used by many libraries, many technologies. So it is the common way to define an environment in NestJS but by default the NODE\_ENV environment variable is undefined. And then ‘config’ takes the development configuration by default.
* Priority goes for environment variables always and then to file based configuration.

E.g. Environment variables can be passed like >PORT=3001 npm run start:dev

Then in code, we can access it like this- process.env.PORT

# Testing NestJS Application

## Jest

* NestJS ships with a testing infrastructure built in. That means we don't have to configure a lot in terms of testing.
* It uses Jest as a testing framework by default. <https://jestjs.io/>
* Jest is one of the most common testing framework these days.
* Another cool thing about Jest is that by default requires zero configuration.
* Now of course you can configure Jest for your custom needs but the default configuration shipping with Jest is pretty much enough for most projects.
* Whenever we run the npm test script our tests are going to run.
* One thing to note – if we do not have a NODE\_ENV variable defined, Jest is going to set the value to “test”. And that might not work for application. We might need NODE\_ENV set to either development or production for our configuration reasons.
* So your test script should be like this –

"test": "NODE\_ENV=development jest",

* By default, Jest is looking for files that end with .spec.ts and will execute them.
* Mock functions in Jest can do a lot for us. It can track how many times a function has been called. It can track with which arguments it's being called.
  + We can use it to mock functionality to simulate a certain behavior that we want for our specific test case.
  + We can change the return value or even mock a rejected or resolved promise.
  + And this is super useful.

# Tips and Tricks

* **Decorators** are a typescript feature that allows annotation of classes or class members such as methods or properties in order to add extra functionality.