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

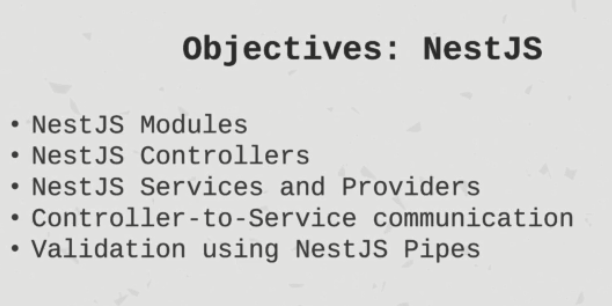
nest --version

* 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)

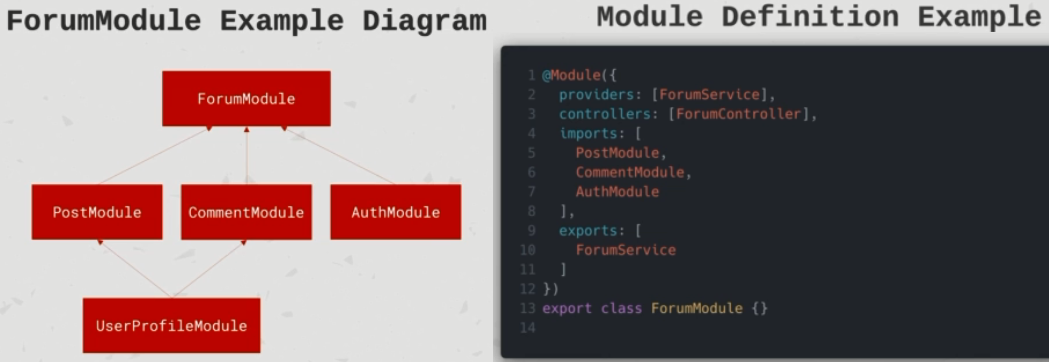
## Introduction to NestJS Modules

### 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

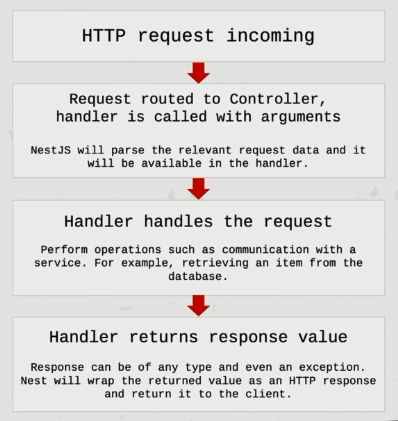
controller for the controller schematic.

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

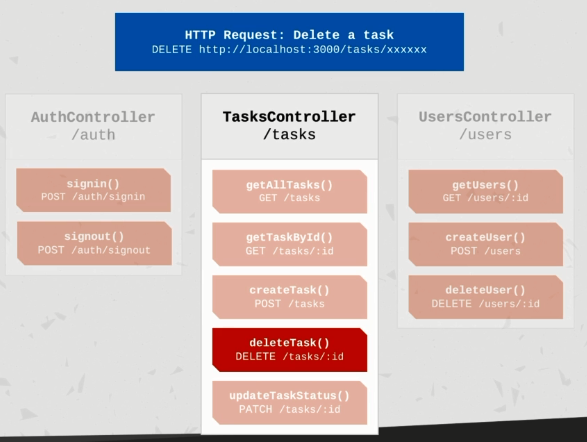
--no-spec flag to tell the CLI to not generate a spec file for us because we're not writing any unit tests right now.

* 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 –



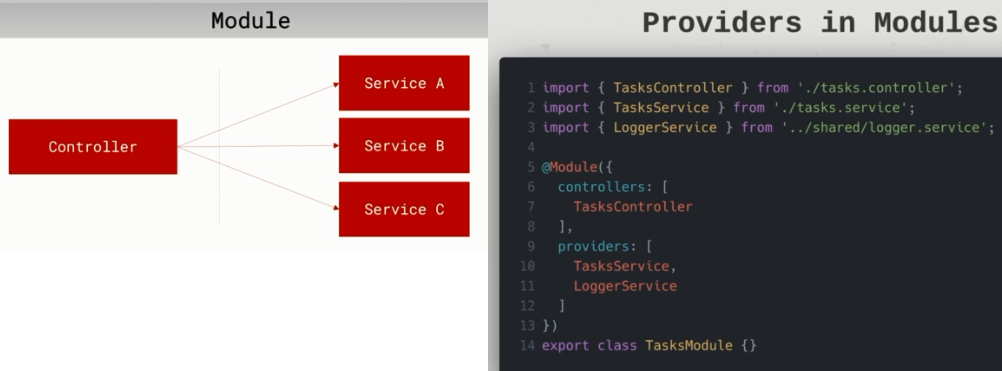
## Introduction to NestJS Providers and Services

### 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.

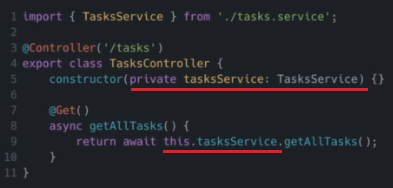
### Service

* **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

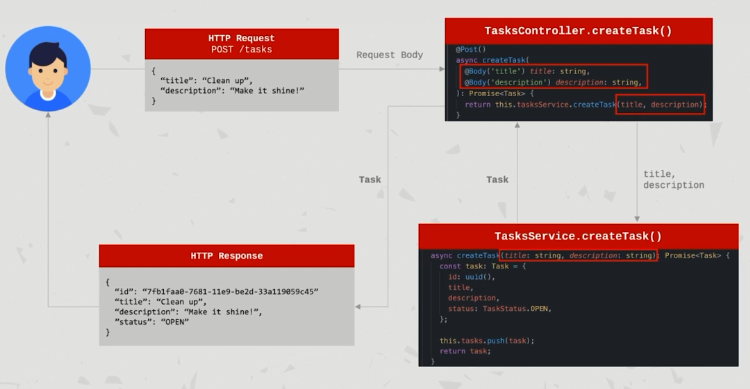
* Model defines the shape of the business object which we are dealing with. E.g. Task, User, etc.
* We can define a model either as a class or as an interface.
* Interfaces are a typescript concept that simply enforces the shape of an object upon compilation. Therefore after compilation, interfaces are 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

# 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.