BCDV 1041

NestJs - A Node.js Framework

2023 Fall

week 01 - class 03

- What is NestJS
- NestJs vs Angular
- NestJs Architecture

What is NestJS?

"A progressive Node.js framework for building efficient, reliable and scalable server-side applications"

NestJS

- is a framework for building Node.js server-side applications
- is built with and fully supports TypeScript
- allows developers to create testable, scalable, loosely coupled and easily maintainable applications
- the architecture is heavily inspired by Angular

Purpose of NestJS?

- Nest.js was created to address several key challenges faced by developers when building Node.js applications, such as:
    - Lack of structure: Many Node.js applications lack a clear and organized structure, leading to code that's difficult to maintain.
    - Scalability concerns: As applications grow, maintaining scalability and code maintainability becomes a challenge.
    - Limited TypeScript support: While TypeScript is a powerful language, Node.js didn't provide native TypeScript support out of the box.
- Nest.js aims to provide solutions to these challenges, making it easier to build scalable and maintainable Node.js applications.

Typescript Support

- TypeScript is a statically typed superset of JavaScript, and Nest.js embraces it fully.
- Writing server-side code in TypeScript offers benefits such as enhanced code quality, better tooling support, and improved developer productivity.
- With Nest.js, you can write your entire backend in TypeScript, ensuring type safety and better code readability.

https://www.typescriptlang.org/

```
class UserAccount {
  name: string;
  id: number;

  constructor(name: string, id: number) {
    this.name = name;
    this.id = id;
  }
}

const user: User = new UserAccount("Murphy", 1);
```

Modular Architecture

Nest.js Encourages a Modular Approach

- Nest.js promotes a modular architecture for building applications.
- The core idea is to divide your application into smaller, manageable modules.
- Each module typically consists of its own set of controllers, services, and components, which work together cohesively.

```
// books.module.ts
import { Module } from '@nestjs/common';
import { BooksController } from './books.controller';
import { BooksService } from './books.service';

@Module({
  controllers: [BooksController],
  providers: [BooksService],
})
export class BooksModule {}
```

Advantages of a Modular Approach

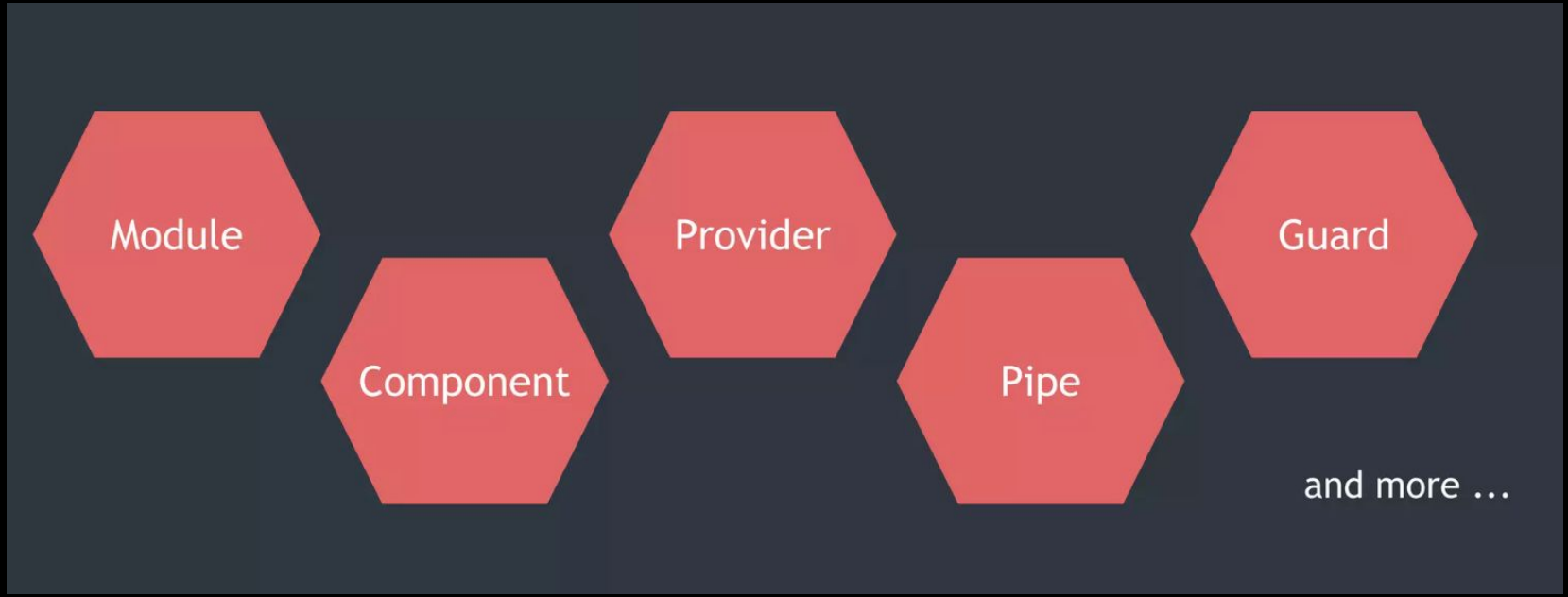
```
// app.module.ts (Main Application Module)
import { Module } from '@nestjs/common';
import { BooksModule } from './books/books.module';
import { UsersModule } from './users/users.module';

@Module({
  imports: [BooksModule, UsersModule],
})
export class AppModule {}
```

- Improved Organization: Breaking your application into modules enhances its organization. Each module can be responsible for a specific feature or functionality.
- Reusability: Modules are designed to be reusable. You can easily use the same module in multiple parts of your application or even in different projects.
- Maintainability: Smaller modules are easier to maintain and test, reducing the complexity of your codebase.
- Scalability: As your application grows, it becomes easier to scale because you can add new modules or extend existing ones without affecting other parts of the application.

Angular

Angular *enables developers to create dynamic* ***web applications****.*

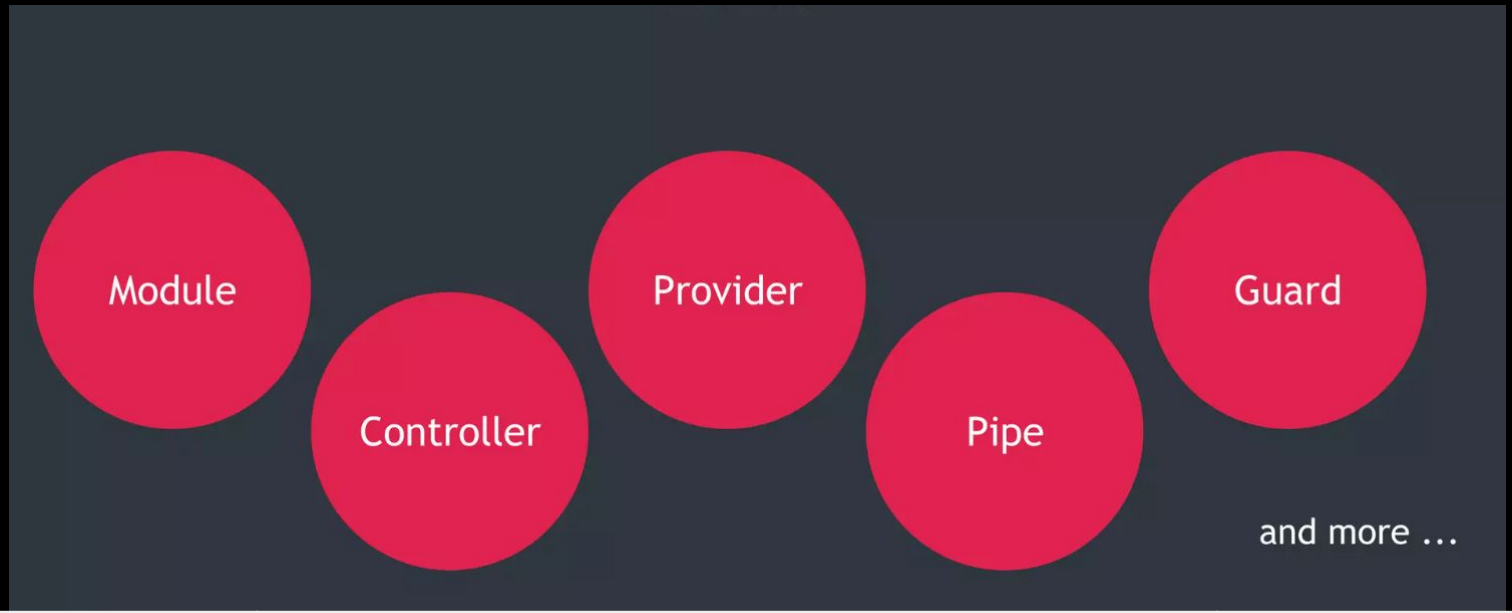
https://angular.io/



NestJS

NestJS**, on the other hand, **is a progressive Node.js framework** that helps programmers to **build scalable server-side applications by using JavaScript as a server-side programming language.

https://nestjs.com



Create a New Project

```
$ npm i -g @nestjs/cli
$ nest new my-nest-app
```

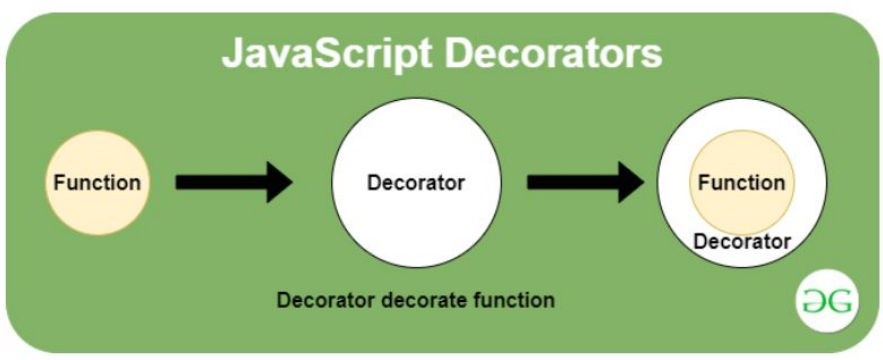
```
my-nest-app
└ src
  └ app.controller.spec.ts
  └ app.controller.ts
  └ app.module.ts
  └ app.service.ts
  └ main.ts
└ test
  └ app.e2e-spec.ts
  └ jest-e2e.json
└ nest-cli.json
└ package.json
└ tsconfig.build.json
└ tsconfig.json
```

Decorators

“An ES6 Decorators are the way of wrapping one piece of code with another or apply a wrapper around a function adding new functionality to it, without altering its original code”

Core NestJS Decorators

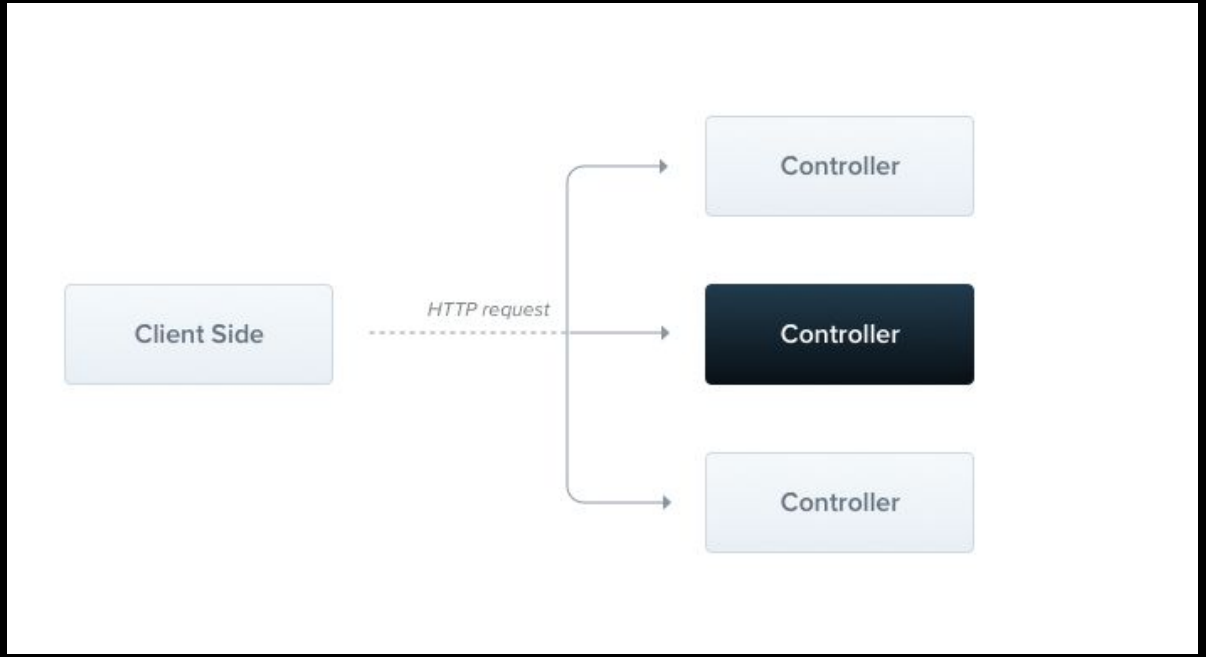
- @Controller
- @Injectable
- @Module
- @Pipe
- @Request
- @Response
- @Param
- @Session



Controllers

Controller

A service is a provider for the controller, and is responsible for data storage and retrieval



Controller

A module is a class annotated with a `@Controller()` decorator. The routing mechanism controls which controller receives which requests. Frequently, each controller has more than one route, and different routes can perform different actions.

/src/app.controller.ts

```ts
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';
@Controller()
export class AppController {
  constructor(private readonly appService: AppService) { }

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }
```

Services

Services

A service is a class annotated with a `@Injectable()` decorator. *A service is a provider for the controller, and is responsible for data storage and retrieval*

/src/app.service.ts

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

@Injectable()
export class AppService {
  getHello(): string {
    return 'Hello World!';
  }
}
```

Dependency Injection in Nest.js

In Nest.js, dependency injection is a foundational concept that optimizes data and service management within your application. It delivers several advantages, including code reusability, enhanced testability, and streamlined maintenance, by enabling the injection of dependencies like services and components into various application sections.

Key Benefits:

1. Code Reusability: Dependency injection facilitates the creation and reuse of services and components throughout different segments of your application. This diminishes redundant code and fosters a more structured and organized codebase.

2. Testability: By employing dependency injection, writing unit tests for your application becomes more straightforward. During testing, you can effortlessly provide mock or stubbed versions of services, ensuring that your tests remain isolated and concentrate on specific components.

3. Loose Coupling: Dependency injection promotes loose coupling in your code. This means that distinct application segments aren't tightly bound to specific implementations. Instead, they rely on interfaces or abstract classes, simplifying the process of swapping implementations or extending functionality without impacting other code sections.

Dependency Injection Example

```ts
// logger.service.ts
import { Injectable } from '@nestjs/common';

@Injectable()
export class LoggerService {
  log(message: string) {
    console.log(`[Logger] ${message}`);
  }
}
```

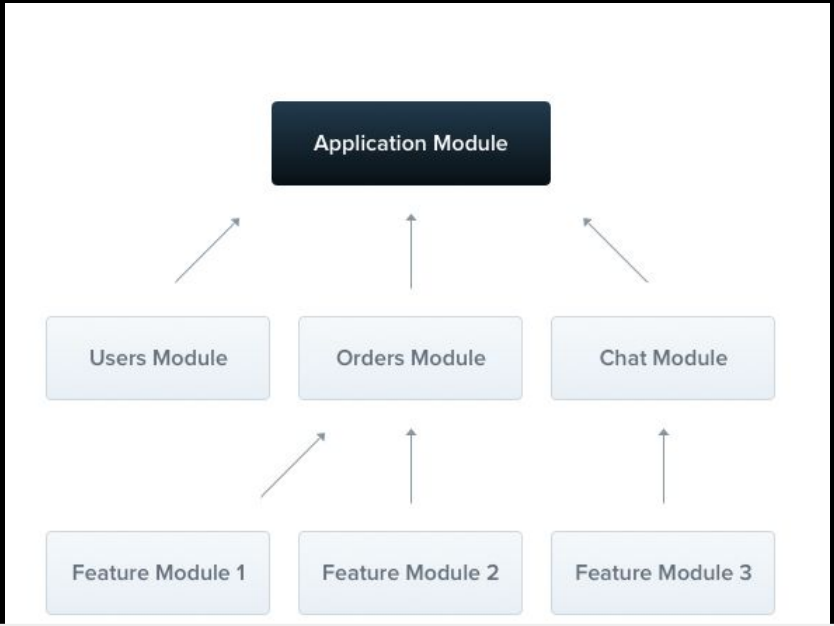
```ts
// app.controller.ts
ontroller, Get } from '@nestjs/common';
oggerService } from './logger.service';

r()
ss AppController {
tor(private readonly loggerService: LoggerService) {}

(): string {
oggerService.log('Hello World Request');
  return 'Hello World!';
 }
}
```

- We use the @Injectable() decorator to mark the LoggerService as an injectable service. (Provider)
- In the AppController, we inject the LoggerService into the constructor using TypeScript's parameter property shorthand.
- Inside the getHello method, we use the injected LoggerService to log a message.

Modules

Root Module

Each application has at least one module, a **root module**. The root module is the starting point Nest uses to build the application graph - the internal data structure Nest uses to resolve module and provider relationships and dependencies.



Module

A module is a class annotated with a `@Module()` decorator. The `@Module()` decorator provides metadata that Nest makes use of to organize the application structure.

/src/app.module.ts

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule { }
```

main.ts

This file will be the entry point for our NestJS application, and it will serve everything defined from here onwards. app. service.

/src/main.ts

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

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);
}
bootstrap();
```

Nest CLI

Nest CLI

The Nest CLI is a command-line interface tool that helps you to initialize, develop, and maintain your Nest applications. It assists in multiple ways, including scaffolding the project, serving it in development mode, and building and bundling the application for production distribution. It embodies best-practice architectural patterns to encourage well-structured apps.

```
# Compiles and runs an application in watch mode
$ nest start --watch

# Compiles an application or workspace into an output folder.
$ nest build

# Generates files based on a schematic
$ nest generate module cats
$ nest generate controller cats
$ nest generate service cats
...
```