NestJS

NestJS, um framework que funciona por base de decoradores, divididos em 3 partes: **module, service e controller.** Eles vão funcionar como routers e o controlle de como conhecemos. O nest pode usar o express ou fastify.
Para prosseguir, precisamos entender o que são **decoradores**: eles basicamente funcionam como **funções** que adicionam funcionalidades extras a uma outra função ou classe, sem modificar seu código original.

📦 O que acontece *por baixo dos panos* com os decoradores?

Quando você usa um decorador como `@Controller('home')`, o NestJS **anota informações** na sua classe usando algo chamado **"metadata"**.

Essas informações ficam **guardadas na classe**, como se fosse um "pós-it invisível" que o NestJS cola ali, dizendo:
"Ei, essa classe é um Controller da rota `/home`."

🧠 Em palavras simples:

- O decorador **não altera** o funcionamento da classe diretamente.
- Ele **anota informações extras** (metadata) sobre a classe ou sobre o método.
- Depois, o NestJS **lê essa metadata** para saber o que fazer.

Por isso o NestJS consegue, no momento que o app sobe, olhar para todas as classes e ver:

- "Ah, essa aqui é um módulo."
- "Essa aqui é um controller na rota `/home`."
- "Esse método responde a um GET."

🤯 Moral da história:

- O decorador **não muda o comportamento da função nem da classe**.
- Ele **grava** informações escondidas (**metadata**).
- Depois o "framework" (NestJS) **varre todas as classes** e **lê a metadata** para **montar o servidor**.

💬 **Ou seja: Decoradores são como "anotações inteligentes" que o NestJS lê depois.**

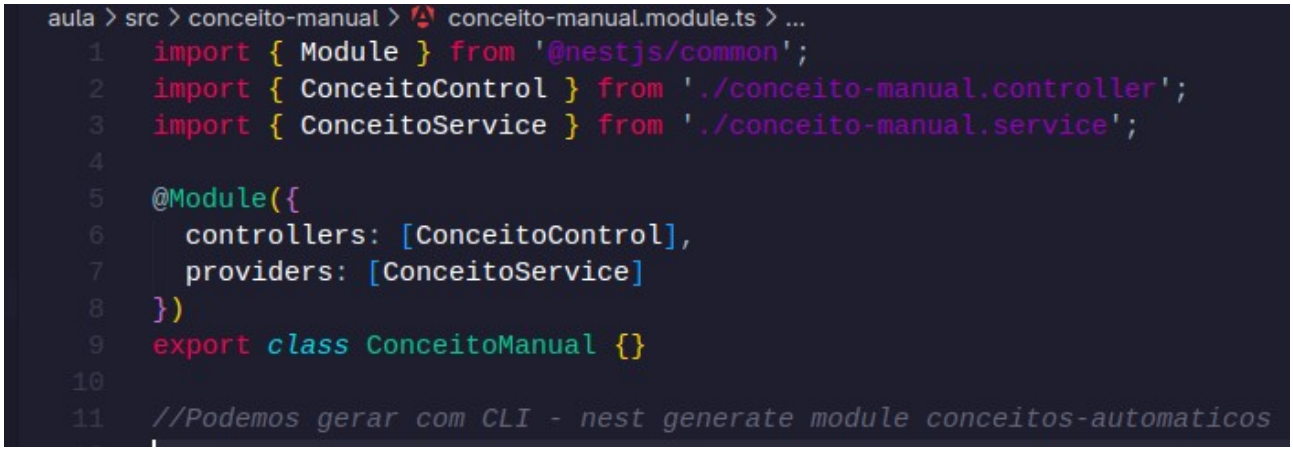
Obs: Para nestJS, temos que usar o typscritp. Agora vamos ver nosso app.ts (que fiz em classe) que irá fazer nosso servidor ficar de pé

O **NestFactory** é a nossa biblioteca e o AppModulo é o nosso modulo, responsavel por fazer o gerenciamento do controller e service

O **módulo** é como se fosse o **organizador** ou **gestor** do sistema.

Ele:

- **Agrupa** Controllers e Services.
- **Informa ao Nest** quem depende de quem.
- **Gerencia a criação** dessas classes (Controller e Service).
- **Entrega** automaticamente (injeção de dependência) as classes já instanciadas.



Agora vamos para o controlador(Controll) responsavel por ‘rotear’ as rotas

```
aula > src > conceito-manual > conceito-manual.controller.ts > ...
import { Controller, Get } from '@nestjs/common';
import { ConceitoService } from './conceito-manual.service';

@Controller('home')
export class ConceitoControl {
  constructor(private readonly conceitoService: ConceitoService) {} // Não esqueça que propriedades no constructor só server para receber argumentos na classe
  @Get()
  home(): any {
    return this.conceitoService.GetHome();
  }
}
//você tb pode gerar por CLI nest generation controller <nome>
//use nest --help
```

Exatamente! No código que você compartilhou, estamos importando os decoradores `@Controller` e `@Get` de `@nestjs/common`, que são parte do NestJS. Eles são usados para definir a estrutura da aplicação de uma forma declarativa. Vamos revisar e entender um pouco mais detalhadamente:

`@Controller`: Esse decorador é usado para **definir a classe como um controlador** e associá-la a uma rota específica. No seu caso, a rota base é `'home'`.

`@Get`: Esse decorador é usado para **definir um método** dentro de um controller que será responsável por responder a requisições HTTP do tipo `GET`. O `@Get()` pode ser usado sem parâmetros (que irá gerar uma rota igual à do controller) ou com parâmetros para definir uma rota mais específica.

Para você entender, usamos o typscript, que diz:
“Quero receber apenas essa classe já instanciada”

Injeção de Dependência: O NestJS, ao ver o **`constructor(private readonly conceitoService: ConceitoService)`**, **instancia a classe `ConceitoService`** automaticamente e a **passa como parâmetro** para o `ConceitoControl` sem que você precise instanciar a classe manualmente.

Por fim, o service:

Lá no modulo, ele fica em providers

O decorador `@Injectable()` no NestJS é fundamental para a **injeção de dependência**. Ele marca uma classe como **injetável**Ao adicionar `@Injectable()` a uma classe, você está dizendo ao Nest que essa classe será gerenciada pelo sistema

de **injeção de dependência** e que ele será responsável por criar instâncias dessa classe, bem como injetá-la em outros componentes quando necessário.

Quando o Nest vê o `@Injectable()`, ele registra essa classe internamente, tornando-a disponível para a injeção em outros lugares onde seja necessário.

O que é Injeção de Dependência?

Imagine que você tem uma **máquina** que precisa de **energia elétrica** para funcionar, mas, em vez de você ter que conectar essa máquina a uma tomada diretamente (o que seria instanciar manualmente a energia), há uma **central de energia** que se preocupa em fornecer a energia para todas as suas máquinas automaticamente, quando elas precisarem.

Então, a central de energia **fornece a energia** para a máquina no momento certo, sem que você precise se preocupar em fazer isso manualmente, e **garante que sempre haverá energia disponível** quando você precisar.

Como isso se aplica ao código?

No contexto do NestJS (e de muitos outros frameworks), a injeção de dependência é exatamente isso, mas no lugar de **energia**, você tem **objetos ou serviços** (como classes que oferecem funcionalidades específicas, por exemplo, um serviço de banco de dados ou um serviço de autenticação).

Exemplo:

- Imagine que você tem uma **classe Controller**, que precisa de uma **classe Service** para funcionar (digamos, um `ConceitoService` que traz dados para o controller). Se você não tivesse DI, o Controller teria que instanciar manualmente o Service dentro dele.

Mas com a injeção de dependência, o NestJS está dizendo:

- **"Eu, como o NestJS, vou garantir que o `ConceitoService` seja instanciado e injetado no `ConceitoController` automaticamente. Você não precisa se preocupar com isso."**