**NestJS**

NestJS é um **framework do NodeJS** altamente modular**,** usado como estrutura para criar o backend de aplicações. Podemos usá-lo para criar APIs, aplicações em tempo real via Sockets e microsserviços, por exemplo.

Embora seja possível fazer esses mesmos tipos de aplicações com Express, Fastify e outros frameworks, o padrão e a estrutura do Nest facilita a criação e manutenção do código. O Nest foi desenvolvido de maneira a tornar um projeto Node mais eficiente, escalável e organizado, contando com o uso de TypeScript.

**Arquitetura Cliente-Servidor**

O modelo **cliente-servidor**, em computação, é uma estrutura de aplicação que distribui as tarefas e cargas de trabalho entre os fornecedores de um recurso ou serviço designados como **servidores,** e os requerentes dos serviços, designados como **clientes.**

Um servidor, também chamado de **host ou server**, é um computador que hospeda um serviço ou aplicação que normalmente pode ser acessado via internet.

Um cliente é um computador, tablet, celular ou dispositivo que possui um programa ou aplicação que solicita dados de um servidor.

Solicitação do cliente para o servidor: **request**

Resposta do servidor para o cliente: **response**

**Criando um projeto novo com Nest CLI**

A maneira recomendada para criar um novo projeto é usando o Command Line Interface (CLI) do Nest. Para isso, instalamos de forma global usando o NPM:

**npm i -g @nestjs/cli**

Após instalado, você pode criar um novo projeto Nest com o comando:

**nest new project-name**

Ou então, você pode criar um novo projeto sem nem mesmo instalar o CLI em sua máquina, fazendo uso do seguinte comando:

**npx @nestjs/cli new project-name**

É também possível usar um boilerplate oficial de um repositório remoto para uma aplicação Nest. Os comandos são os seguintes:

**git clone** [**https://github.com/nestjs/typescript-starter.git**](https://github.com/nestjs/typescript-starter.git) **project-name**

**cd project**

**npm install**

**Métodos HTTP (tipos de request)**

Métodos HTTP são formas de acesso. Por meio deles, nós dizemos ao servidor que tipo de solicitação nós queremos fazer para ele. A seguir, os métodos HTTP mais conhecidos são:

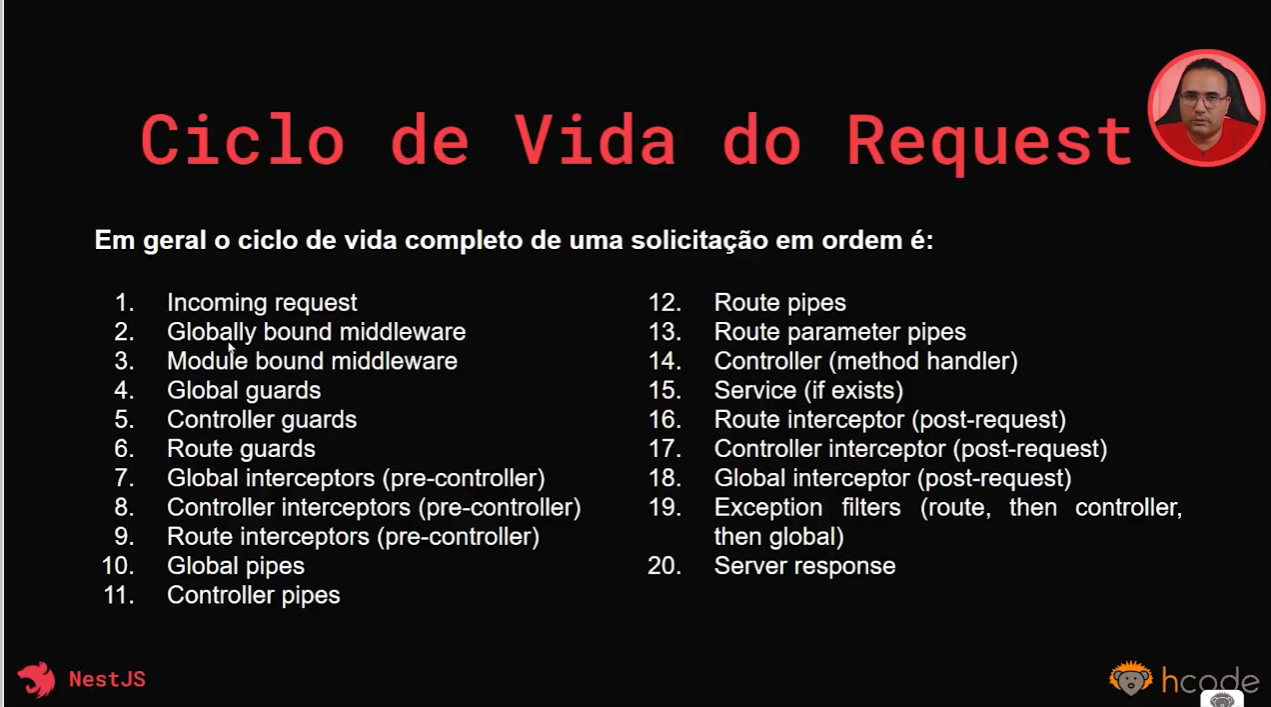
**C ---------------** create ----------- **POST:** enviar/inserir dados

**R** ---------read ------------ **GET:** obter dados

**U** --------------- update ---------- **PUT:** editar dados

**D** -------------- delete ------------ **DELETE:** excluir dados

**Ciclo de Vida do Request (middlewares)**

****

**Principais arquivos do NestJS**

**Bootstrap – main.ts:** é o arquivo com as configurações iniciais da nossa aplicação. Nele, instanciamos o app e revelamos a porta em que ele será executado. Além disso, configurações de CORS, WebSockets e outros também podem ser incluídos no arquivo Bootstrap.

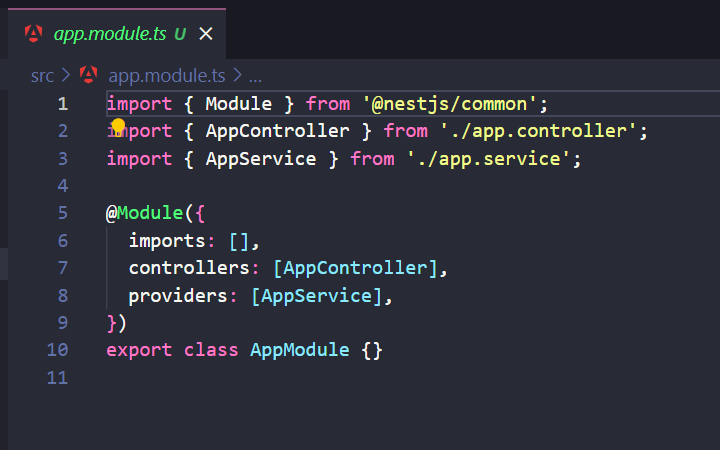
**Module – app.module.ts:** arquivo que agrupa todos os recursos (módulos) de nossa aplicação.

**Controller – app.controller.ts:** o controller é a porta de entrada para toda request que será realizada em nossa aplicação. Nele, obtemos as solicitações HTTP e enviamos as solicitações para o service da nossa aplicação. Ele é como um garçom em um restaurante: será o primeiro a interceptar requests.

**Service – app.service.ts:** no service, encontramos funções que agrupam as lógicas e regras de negócio de cada rota na nossa aplicação. Voltando no exemplo do restaurante, o service seria o cozinheiro: responsável por preparar o prato.

**Decorators**

Decorators são recursos do Typescript que nos permitem modificar um componente (classe, objeto, função) em tempo de execução, afim de testar seus comportamentos para diferentes situações.



No exemplo acima do arquivo inicial app.module.ts, nós vemos essa classe vazia AppModule. Embora vazia, a classe recebe funcionalidades e comportamentos por meio do decorator @Module, que vem antes dela.

**API**

A sigla API deriva da expressão inglesa Application Programming Interface que, traduzida para o português, pode ser compreendida como uma interface de programação de aplicação. Ou seja, API é um **conjunto de normas que possibilita a comunicação entre plataformas por meio de uma série de padrões e protocolos.**

Por meio de APIs, desenvolvedores podem criar novos softwares e aplicativos capazes de se comunicar com outras plataformas. Por exemplo: caso um desenvolvedor queira criar um aplicativo de fotos para Android, ele poderá ter acesso à câmera do celular através da API do sistema operacional, sem ter a necessidade de criar uma nova interface de câmera do zero.

O mesmo acontece com aplicativos que utilizam os serviços de mapas por meio da API do Google Maps ou, ainda, nas integrações entre apps, como o Spotify e o Instagram, que possibilita compartilhar faixas nos Stories.

Exemplo: Quando você vai a um restaurante, você não entra na cozinha para fazer o seu próprio prato, você olha o menu, escolhe o que deseja e o garçom leva o seu pedido para a cozinha. A cozinha prepara a comida e o garçom entrega a você.

Da mesma forma, uma API é como o garçom: um menu que permite que um programa de computador (ou aplicativo) peça informações ou ações de outro programa, serviço ou sistema, sem precisar entender como tudo funciona nos bastidores.

**API REST**

REST é a sigla para Representational State Transfer e representa um conjunto de restrições para uma API.

Para uma API ser considerada do tipo RESTful, ela precisa estar em conformidade com os seguintes critérios:

* Ter uma arquitetura cliente/servidor formada por clientes, servidores e recursos, com solicitações gerenciadas por HTTP.
* Estabelecer uma comunicação stateless entre cliente e servidor. Isso significa que nenhuma informação do cliente é armazenada entre solicitações GET e todas as solicitações são separadas e desconectadas.
* Armazenar dados em cache para otimizar as interações entre cliente e servidor.
* Ter uma interface uniforme entre os componentes para as informações serem transferidas em um formato padronizado. Para tanto, é necessário que:

1. Os recursos solicitados sejam identificáveis e estejam separados das representações enviadas ao cliente;

2. Os recursos possam ser manipulados pelo cliente por meio da representação recebida com informações suficientes para tais ações;

3. As mensagens autodescritivas retornadas ao cliente contenham informações suficientes para descrever como processá-las;

4, Hipertexto e hipermídia estão disponíveis. Isso significa que após acessar um recurso, o cliente pode usar hiperlinks para encontrar as demais ações disponíveis para ele no momento.

* Ter um sistema em camadas que organiza os tipos de servidores (responsáveis pela segurança, pelo carregamento de carga e assim por diante) envolvidos na recuperação das informações solicitadas em hierarquias que o cliente não pode ver.

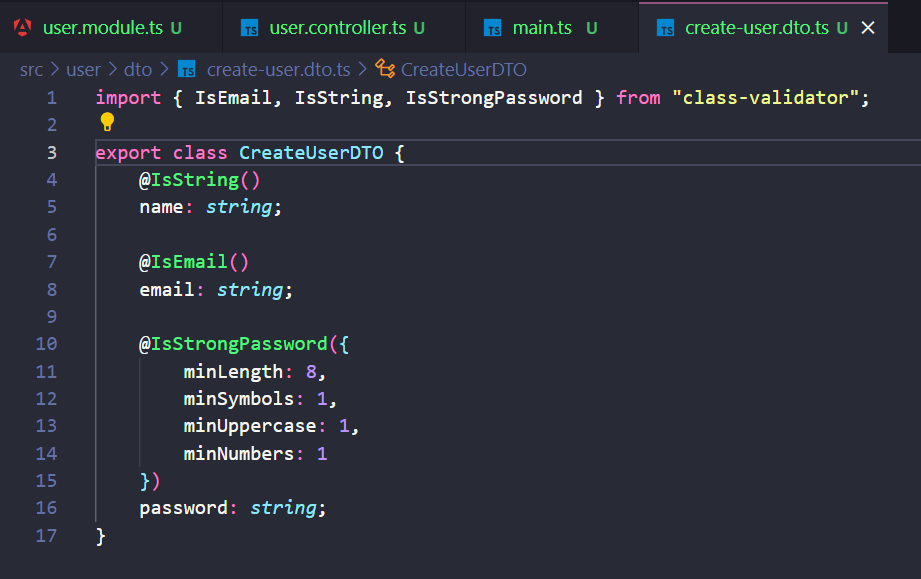
**Validação com DTO**

**DTO (Data Transfer Object)** significa Objeto de Transferência de Dados e é um padrão de projeto de software usado para transferir dados em uma aplicação de um ambiente para outro. Normalmente, obtemos um objeto de dados ao acessar um banco de dados ou no NestJS podemos obter um objeto de dados recebidos de uma request.

Documentação de DTO: <https://docs.nestjs.com/techniques/validation>

Ao usar o padrão de DTO, facilitamos a validação de dados, uma vez que podemos verificar e até alterar os dados que estão em um objeto. Ou até mesmo lançar uma exceção caso o objeto não possua a estrutura ou os tipos de dados que nós estamos esperando.

A validação do DTO ocorre por meio de uma biblioteca de decorators, podendo ser class-validator e class-transformer.



**ORM**

ORM (Object Relational Mapping) é uma técnica de desenvolvimento utilizada para representar as tabelas ou coleções de um banco de dados com classes e objetos.

Cada tabela ou coleção terá umaa classe que descreve sua estrutura e as instâncias dessa classe/objeto são os registros.

Os ORMs que são mais facilmente integrados com o NestJS e são citados atualmente na documentação do Nest são:

* MikroORM
* Squelize
* TypeORM
* Knex
* Prisma

**USANDO PRISMA**

**npm i --save-dev prisma:** instala a dependência do Prisma.

**npx prisma init:** inicializa o Prisma Client.

A partir daí, basta seguir os passos citados após digitar o comando acima (isso inclui estabelecer uma conexão com o banco de dados e informar sua URL num arquivo .env, além de setar qual é o provider e fazer um pull do banco de dados após estabelecer a conexão, caso ele já exista).

**PIPES**

No Nest, os Pipes são são usados para transformar os dados que passam por eles ou para validá-los.

Eles são classes com o decorator **@Injectable** e devem implementar a interface **PipeTransform.** Essa interface, por sua vez, irá abrigar a declaração do método transform() que receberá o valor que passará pelo “tubo”.

**INTERCEPTORS**

Interceptors são utilizados como middlewares. Eles possibilitam adicionar lógicas extras antes ou depois da execução de métodos, transformar o resultado retornado de uma função ou exceção lançada pela função, estender o comportamento de uma função ou até mesmo substituir completamente uma função dependendo de condições específicas.

Eles são classes com o decorator **@Injectable** e devem implementar a interface **PipeInterceptor.**

**MIDDLEWARES**

Middlewares são funções que são chamadas **antes** do manipulador de rota. Ao contrário dos interceptors, os middlewares não podem ser chamados depois.

Essas funções podem ser declaradas em uma classe com o decorator **@Injectable** e implementando a interface **NestMiddleware**, que irá obrigar a declaração da função middleware com o nome **use.**

Essas funções tem acesso aos objetos **Request** e **Response** e a função **next()** que irá chamar a próxima função middleware ou por fim o manipulador de rota, seguindo assim o design pattern “**Chain of Responsibility**”.

Com o uso de middlewares, podemos executar qualquer código, como por exemplo fazer alterações nos objetos de solicitação ou resposta, encerrar o ciclo de solicitação ou chamar a próxima função do middleware.

Visto que os middlewares são chamados sempre **antes** de uma rota, é possível aplica-los em algumas ou todas as rotas da nossa aplicação. E ainda também filtrar rotas que não devem usar o middleware.

**GUARDS**

No Nest, os Guards são classes com o decorator **@Injectable** e devem implementar a interface **CanActivate.**

Os Guards tem uma única responsabilidade: determinar se uma solicitação Request pode ser tratada pelo manipulador de rotas ou não.

Este recurso é muito útil para criarmos rotas que só podem ser obtidas respostas caso o cliente esteja autenticado e/ou tenha permissões suficientes.

**EXCEPTIONS**

No Nest, as Exceptions são capturadas automaticamente por uma camada da aplicação que é responsável por transformá-las em mensagens amigáveis.

Essas mensagens sempre irão ser respondidas como um objeto JSON contendo o Status Code HTTP e uma mensagem descrevendo de forma resumida o código de status.

Embora possamos usar as exceções que já existem no Nest, ainda podemos criar exceções personalizadas criando uma classe que deve se estender da classe HttpException.

**PARAM DECORATORS**

No NestJS, os param decorators (decoradores de parâmetro) são usados para extrair e validar informações de diferentes partes da requisição HTTP, como parâmetros de rota, corpo da requisição, cabeçalhos, query params, entre outros. Eles facilitam a manipulação de dados na camada de controladores.

**Principais Decoradores de Parâmetro no NestJS**

@Param(): Extrai parâmetros de rota.

@Query(): Extrai parâmetros de query.

@Body(): Extrai o corpo da requisição.

@Headers(): Extrai os cabeçalhos da requisição.

@Req(): Injeta o objeto de requisição completa.

@Res(): Injeta o objeto de resposta completa.

@Next(): Injeta a função next em middlewares.

@Session(): Injeta a sessão (útil para aplicações que utilizam sessões).

@Ip(): Extrai o endereço IP do cliente.

@HostParam(): Extrai parâmetros do host.

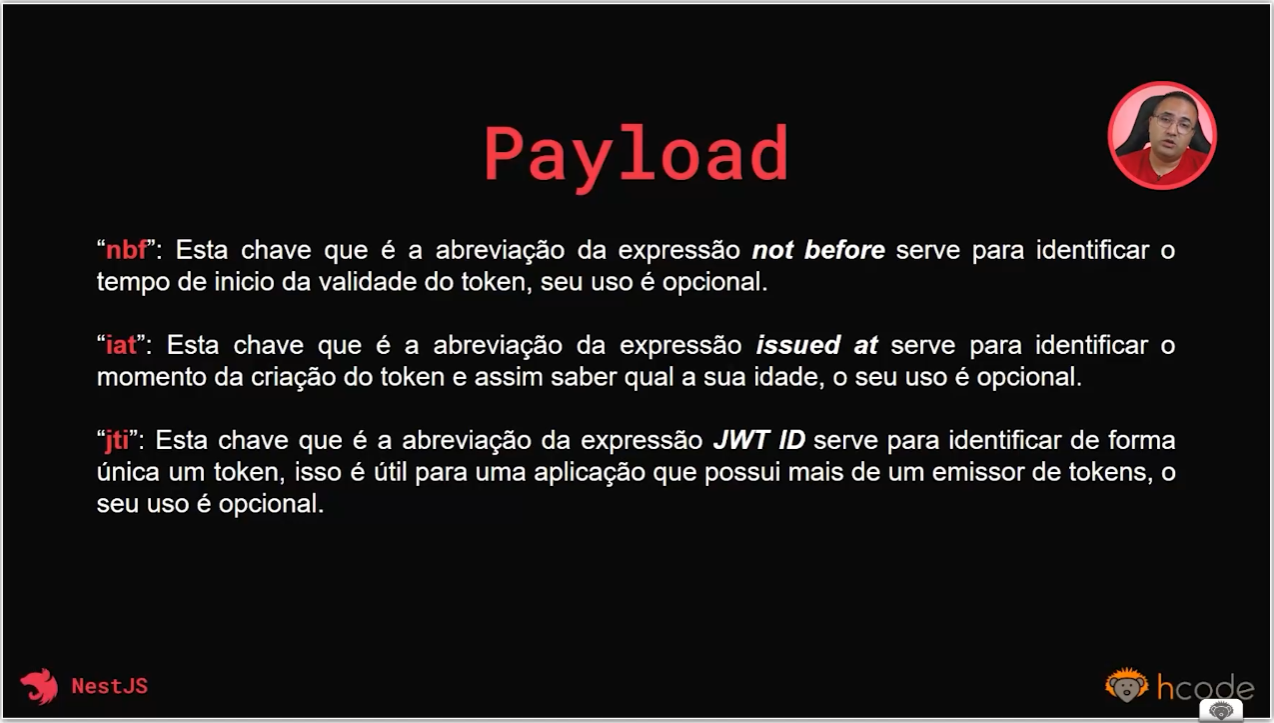
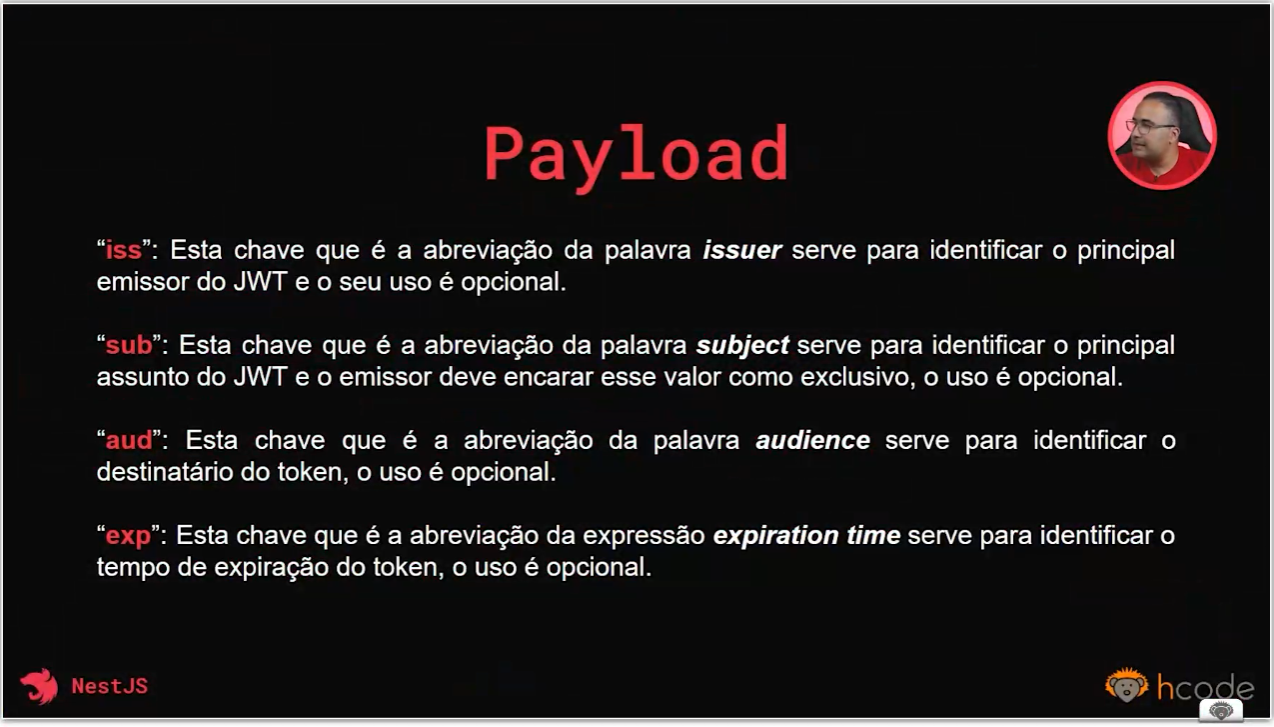
**JSON WEB TOKEN (JWT)**

JSON Web Token é um padrão aberto RFC 7519 que fornece um método seguro para comunicar informações entre duas partes. Um token JWT é uma string composta de partes separadas pelo caractere ponto (.). Cada uma das partes respectivamente são:

**1. Header:** é um JSON codificado em base64 que contém a informação do algoritmo usado na criptografia da assinatura e o tipo de token (padrão mais comum é o HS256).

**2. Payload:** é um JSON codificado em Base64 que contém os dados de fato que você deseja transportar. Embora qualquer dado possa ser colocado nesta parte do token, é importante frisar que a codificação do Base64 é facilmente revertida, por isso qualquer informação colocada no payload não deve ser sensível ou crítica caso seja exposta.

Alguns nomes de chaves no payload são reservados e ajudam no funcionamento da verificação do token, como por exemplo as chaves **iss, sub, aud, exp, nbf, iat** e **jti.**

****

**3. Signature:** uma string em Base64 do resultado da criptografia com o algoritmo escolhido e identificado no header. A string que será criptografada é a string do header codificada em Base64 concatenada com um ponto e com a string codificada em Base64 do payload.

Isso fará com que qualquer alteração no header ou payload mude o valor da assinatura, que por sua vez, só pode ser obtida de forma válida com uma chave secreta que apenas o emissor deve possuir. Isso evita falsificação de tokens.

**AUTORIZAÇÃO RBAC**

RBAC (Role-based Access Control) serve para liberarmos o acesso de um recurso de nossa API apenas se o usuário tem a função de obter essas informações.

Por exemplo, um usuário autenticado pode alterar o seu próprio nome, mas não pode alterar o nome de outro usuário. Já um usuário administrador pode além de trocar o seu próprio nome, trocar também o nome de outros usuários, pois ele tem essa função.

**ARMAZENAMENTO DE SENHAS COM HASH**

Um hash é um número ou código que é gerado a partir de dados de qualquer tamanho. Esse número tem sempre o mesmo tamanho, independentemente do tamanho dos dados originais.

**Características Principais:**

**Tamanho Fixo:** Não importa o quão longa seja a palavra ou frase que você coloca na máquina, o resultado (o hash) sempre terá o mesmo comprimento.

**Difícil de Reverter:** Se você apenas tiver o hash, é quase impossível descobrir qual foi a palavra ou frase original que foi colocada na máquina.

**Para Que Serve um Hash?**

**Guardar Senhas de Forma Segura:** Em vez de guardar sua senha real, um site guarda o hash da sua senha. Assim, mesmo que alguém roube o banco de dados do site, eles não terão acesso à sua senha real.

**Verificar Integridade de Arquivos:** Quando você baixa um arquivo, você pode usar um hash para garantir que o arquivo não foi alterado. O autor do arquivo pode fornecer o hash original e você pode comparar com o hash do arquivo baixado. Se os dois hashes forem iguais, o arquivo não foi corrompido.

**Assinaturas Digitais e Certificados:** Hashes são usados para garantir que uma mensagem ou documento não foi alterado. Eles ajudam a verificar se algo realmente veio de quem disse que enviou.

**Armazenamento e Busca Rápida:** Hashes são usados em computadores para encontrar dados rapidamente. Eles ajudam a organizar e buscar informações de forma eficiente.

**CORS**

CORS, ou Cross-Origin Resource Sharing, é uma política de segurança implementada nos navegadores para permitir ou restringir requisições feitas de uma origem (domínio) para um recurso localizado em outra origem. Essa política impede que uma página da web faça requisições para um domínio diferente do domínio que a serviu. Isso protege contra certos tipos de ataques, como o Cross-Site Request Forgery (CSRF).

Usar este recurso de segurança é muito importante pois um site FAKE poderia se passar pelo site real e fazer uso em determinados momentos da API real. Outro cenário é evitar o uso da sua API sem permissão ou abusos no uso de recursos de outras origens,

**ATAQUE DDOS**

É um tipo de ataque cibernético que tem como objetivo tornar um serviço online indisponível, sobrecarregando-o com uma quantidade massiva de tráfego malicioso. Diferente de um ataque DoS (Denial of Service), que é realizado a partir de uma única fonte, um ataque DDoS é realizado a partir de múltiplas fontes distribuídas geograficamente.

**Como Funciona um Ataque DDoS?**

**Botnets:** Os hackers frequentemente utilizam redes de dispositivos infectados, conhecidos como botnets, para realizar ataques DDoS. Esses dispositivos podem incluir computadores pessoais, servidores, dispositivos IoT, entre outros, que foram comprometidos por malware.

**Gerar Tráfego Massivo:** O hacker coordena os dispositivos da botnet para enviar um volume massivo de tráfego para o alvo do ataque, que pode ser um servidor web, serviço online, rede ou infraestrutura.

**Sobrecarregar o Alvo:** O tráfego sobrecarrega a capacidade de processamento do alvo, causando lentidão, interrupções ou completa indisponibilidade do serviço.

**Como se proteger desse ataque em uma API Nest?**

Por usar a biblioteca **@nestjs/throttler**, que possui recursos para bloquear acessos á rotas de sua API em determinada quantidade de tentativa de acessos.

**VARIÁVEIS DE AMBIENTE**

Variáveis de ambiente são valores dinâmicos que podem afetar o comportamento dos processos em execução em um sistema operacional. Elas são usadas para configurar informações importantes como caminhos de diretórios, configurações de sistema, credenciais de acesso e muito mais. As variáveis de ambiente permitem que os aplicativos sejam configurados de maneira flexível sem a necessidade de alterar o código.

Para utilizarmos variáveis de ambiente no NestJS, basta instalarmos o módulo **@nestjs/config** e importarmos ele no app.moduile.ts. A partir disso, podemos criar um arquivo .env que conterá todas as variáveis de ambiente necessárias da nossa aplicação.

**LIBS IMPORTANTES QUE PODEM SER USADAS EM NEST.JS**

**MULTER (upload de arquivos)**

Multer é um middleware para Node.js que facilita o manuseio de uploads de arquivos em aplicações web usando a biblioteca Express. Ele é especificamente projetado para lidar com formulários multipart/form-data, que são comumente usados para upload de arquivos.

**NODEMAILER (envio de e-mails)**

Nodemailer é uma biblioteca para Node.js que facilita o envio de e-mails. É uma ferramenta poderosa e flexível que permite enviar e-mails de maneira simples, integrando-se com diversos serviços de e-mail, incluindo SMTP, serviços de e-mail baseados na web como Gmail e até mesmo serviços de e-mail transacional como SendGrid e Mailgun.

**MIGRATIONS**

As migrations de banco de dados são uma prática comum no desenvolvimento de aplicativos para manter o controle de versões do esquema do banco de dados. Elas fornecem uma maneira de criar, modificar ou excluir tabelas e colunas de forma controlada, permitindo que as alterações no esquema sejam aplicadas de maneira consistente em diferentes ambientes.

Em vez de fazer alterações diretas no banco de dados manualmente, as migrations oferecem uma abordagem programática e rastreável para realizar essas alterações. Elas são escritas como scripts ou arquivos de código que descrevem as modificações a serem feitas no esquema do banco de dados.

**TYPEORM**

TypeORM é um ORM (Object-Relational Mapper) para Node.js e TypeScript que facilita o trabalho com bancos de dados relacionais de uma maneira orientada a objetos. Ele suporta vários bancos de dados, incluindo PostgreSQL, MySQL, MariaDB, SQLite, Microsoft SQL Server e Oracle.

**Funcionalidades do TypeORM**

**Mapeamento Objeto-Relacional:** Mapeia classes e propriedades TypeScript para tabelas e colunas no banco de dados.

**Migrações:** Facilita a gestão e execução de migrações de banco de dados.

**Consultas Avançadas:** Permite escrever consultas complexas usando uma API baseada em objetos.

**Relacionamentos:** Suporta todos os tipos de relacionamentos (um-para-um, um-para-muitos, muitos-para-um, muitos-para-muitos).

**Repositórios e Gerentes de Entidade:** Proporciona uma maneira organizada de acessar e gerenciar entidades.

**Facilidade de Uso com TypeScript:** Integra-se bem com TypeScript, aproveitando os recursos de tipagem estática.

Para representar uma nova tabela no banco de dados, sempre criamos **Entidades**. Elas são classes onde cada atributo representa uma coluna da tabela, gerada a partir dos decorators do TypeORM específicos para isso.

Tela de celular com texto branco sobre fundo preto

Descrição gerada automaticamente

**MIGRATIONS NO TYPEORM**

Sempre que quisermos fazer ALTERAÇÕES no BANCO DE DADOS, nós devemos usar as Migrations do TypeORM.

Elas funcionam como um versionamento das alterações realizadas no banco.

Por meio das migrations, podemos criar colunas, dados e tabelas usando código TypeScript, nossas alterações ficam registradas e podemos acompanhar as alterações de outros.

COMANDOS (Salvar em package.json):

"typeorm:ts": "typeorm-ts-node-esm",

"migration:create": "npm run typeorm:ts migration:create -- ./typeorm/migrations/Migrate"

"migration:up": "npm run typeorm:ts migration:run -- -d ./typeorm/data-source.ts",

"migration:down": "npm run typeorm:ts migration:revert -- -d ./typeorm/data-source.ts"

COMANDOS (A serem executados no CLI):

npm i typeorm mysql2 @nestjs/typeorm (instalação TypeORM)

npm run migrate

wizard 1: migrate create (cria um arquivo com a timestamp atual e o nome da migration, nele você encontra a estrutura inicial de um migration)

wizard 2: migrate up (aplica alterações pendentes das migrations no banco de dados)

wizard 3: migrate down (desfaz alterações de uma migration - uma de cada vez)

Ao criar a estrutura inicial de um migration, temos uma classe com duas funções: up e down.

Up representa a alteração que iremos fazer no banco (pode ser um update, insert, delete, etc)

Down representa a ação que vai desfazer essa ação previamente programada.

Usamos o objeto queryRunner e seus métodos para fazer queries (query()), criar tabelas (addTable()) e deletar tabelas (dropTable()).

Após preenchermos devidamente as funções up e down, podemos executar o migrate up para aplicar essas alterações no banco. Mas lembre-se: caso você esteja criando uma tabela e ocorra um erro no meio dessas alterações, talvez seja necessário apagar a tabela manualmente no banco de dados antes de tentar outra vez.

Após todos os testes e a confirmação de que a alteração foi realizada com sucesso, podemos commitar essa migration (ficará salvo no repositório essa alteração que fizemos) e executar o comando npm run prod para enviar isso para produção.

**JEST**

Jest é um framework de testes JavaScript desenvolvido pelo Facebook, amplamente utilizado para testar aplicações React, mas também adequado para qualquer projeto JavaScript ou TypeScript. Ele é conhecido por sua facilidade de uso, velocidade e funcionalidades poderosas.

**Funcionalidades do Jest**

**Zero Configuração:** Para projetos React e outros projetos JavaScript comuns, você pode começar a usar o Jest sem nenhuma configuração adicional.

**Mocks e Spies:** Facilita a criação de mocks e spies para funções e módulos, permitindo testar componentes de forma isolada.

**Snapshots:** Permite capturar e comparar a saída do componente ou função com um snapshot salvo anteriormente, útil para testar componentes de interface.

**Cobertura de Código:** Gera relatórios de cobertura de código detalhados, ajudando a garantir que todas as partes do código sejam testadas.

**Testes Assíncronos:** Suporta testes assíncronos com Promises, async/await, e callbacks.