Protegendo os endpoints de nossa API

Nosso objetivo agora é exigir de nossos clientes a **Autorização** para acessar os endpoints/resources de Nossa API. Deste modo, somente será concedido acesso aos dados, mediante entrega de um **Token válido,** emitido pelo Cognito após o processo de **Autenticação.**
Para realizar a validação deste token, emitido pelo Cognito, pediremos uma ajuda para a biblioteca **Passport**. O uso desta biblioteca em uma aplicação Nest é muito simples, uma vez que podemos utilizar o módulo **@nestjs/passport**.

Esta biblioteca possui um rico ecossistema de estratégias, que implementam diferentes mecanismos de autenticação.

Em nosso caso, iremos implementar uma estratégia baseada no uso de JWT. Para fazer isso no Nest, precisaremos estender a classe **PassportStrategy** e informar qual estratégia iremos utilizar. Ao utilizar esta classe, informaremos as opções da estratégia que estamos utilizando, invocando o método super() em nossa subclasse.
Em seguida, através do método validate(), podemos implementar o nosso callback de verificação. A biblioteca Passport espera que este callback retorne um usuário para indicar que a validação foi realizada com sucesso, ou null se falhar.

NestJS Guards
E para ativar esta validação baseada no uso do Token em nossos endpoints, contaremos com a ajuda dos Guards.
Um Guard é uma classe anotada com o decorator @Injectable(), que deve implementar a interface CanActivate.

Guards possuem uma responsabilidade muito simples. Eles determinam se uma requisição será tratada ou não pelo route handler, dependendo de determinadas condições (permissões, roles, etc) presentes em tempo de execução.

Este contexto é conhecido como **Autorização**, que normalmente caminha junto com seu primo **Autenticação**. Autorização e Autenticação normalmente são tratados por um middleware em aplicações Express tradicionais.
Um middleware pode ser uma boa escolha para suportar o processo de Autenticação, uma vez que atividades como validação de token ou anexar propriedades ao objeto request não estão diretamente conectadas a um contexto particular da rota.

Porém o middleware, por si só, não consegue identificar qual handler deverá ser executado após a chamada da função next().
Por outro lado, Guards possuem acesso a instancia de ExecutionContext, e deste modo, sabem exatamente o que será executado a seguir. Eles são projetados como Exception Filters, Pipes e Interceptors, ou seja, nos permitem introduzir lógica de processamento exatamente em um ponto específico do ciclo request/response.

Agora vamos partir para a implementação que protege nossas rotas.

Relembrando:

- Na aula anterior, já possibilitamos aos clientes/usuários que se autentiquem com suas credenciais, recuperando o JWT emitido pelo Cognito. Este token deverá ser utilizado nas chamadas subsequentes para nossa API protegida.
- Agora iremos proteger nossas rotas da API, com a exigência de um JWT válido como um Bearer Token.

Nota: Guards são executados depois de cada middleware, mas antes de qualquer interceptor ou pipe.