**Aula 20 - Autorização e Autenticação**

# Autenticação e autorização

## Autenticação: Identifique-se!

Já havíamos trabalhado com esse conceito, em que o cliente deve primeiro se identificar para tentar acessar um recurso.

A autenticação é a primeira etapa no tempo de vida de uma sessão de cliente e servidor.

Para que um cliente possa se autenticar, deve haver um registro anterior armazenado em algum lugar. O cliente envia um identificador (como um e-mail) e o servidor irá procurá-lo em seu banco de dados para ver se já existe. Se sim, você poderá responder com suas credenciais completas (não confidenciais).

Caso um cliente tente autenticar antes de ter gerado um registro, o servidor não o encontrará no banco de dados e não haverá credenciais para retornar.

## Métodos de autenticação

* Nome de usuário e senha: É o método tradicional mais utilizado, onde o usuário digita o nome de usuário ou e-mail e senha para autenticar.
* Sem senha (passwordless): Consiste em que, cada vez que quisermos fazer login em um recurso, será enviado um link para o e-mail que nos permitirá acessá-lo sem a necessidade de senha.
* Por redes sociais: Vários aplicativos nos dão a opção de fazer login diretamente com uma rede social. A principal vantagem é que os dados dessa conta social são usados diretamente para fazer login.
* Dados biométricos: Autentica os usuários por meio de impressões digitais.
* JWT (JSON Web Token): Este método de código aberto permite a transmissão segura de dados entre as diferentes partes. É comumente usado para autorização de um par de chaves contendo uma chave pública e uma privada.
* OAuth 2.0: Permite que o usuário se autentique e acesse os recursos do sistema de que necessita por meio de uma API.

## Autorização: definindo o escopo de cada usuário

A autorização é o processo pelo qual o servidor decide se, apesar das credenciais que você possui, você tem permissão para acessar um recurso ou não. Em outras palavras, autorizar não significa que o servidor não saiba quem você é.

Devemos ter conjuntos de serviços aninhados para:

* usuários comuns.
* Usuários premium (se trabalhamos com um sistema hierárquico)
* Administrador

ou por exemplo

* Um empregado.
* Um chefe
* Um administrador.

**Importante!**

Como são processos diferentes, não vamos esquecer que eles também devem ter um código de status diferente:

* Para processos de autenticação com falha: 401
* Para usuários rejeitados por querer acessar um recurso não autorizado: 403

Os status **não são intercambiáveis** . Nunca os use levianamente

Três cenários possíveis

1. O cliente deseja acessar um recurso sem credenciais: O servidor irá rejeitá-lo com status 401 = Não autorizado. Forçando-o a autenticar primeiro.
2. O cliente deseja acessar um recurso com credenciais de uma hierarquia não autorizada: O servidor irá rejeitá-lo com status 403 = Proibido. Indicando que, se você deseja acessar o recurso, precisará de credenciais com uma função superior.
3. O cliente deseja acessar um recurso com credenciais de uma hierarquia autorizada: O servidor verifica se as credenciais são válidas e entrega o recurso solicitado.

## Protegendo senhas: bcrypt

Você percebeu?

Se revisarmos o banco de dados que usamos na aula anterior:

* Você notará que os usuários foram salvos exatamente como as informações foram enviadas: incluindo a senha
* Para proteção de dados, devemos salvar uma senha para que não possa ser visualizada, nem mesmo por nós mesmos.
* Para fazer isso, antes de salvar a senha, ela deve ser processada com uma operação conhecida como hash.

**Usando bcrypt para poder fazer um hash**

Devemos reconhecer que não somos especialistas em segurança de computadores, portanto, trabalhar com questões de segurança tão internalizadas pode complicar as coisas para nós.

Para isso, bcrypt se encarregará de realizar a operação de proteger nossas senhas de forma fácil e segura.

Para usá-lo, precisaremos apenas instalá-lo a partir do npm



Considerar:

Se uma senha com hash não puder ser revertida nem por nós mesmos, como saberemos se o cliente fez login corretamente?

* Não podemos fazer uma comparação tão grosseira quanto body.password == user.password. Um recurso diferente deve ser usado.
* bcrypt tem um processo de comparação de senha de sua função de comparação.
* Assim, poderemos saber se o cliente digitou sua senha corretamente, sem precisar saber qual é.

# Estratégias de Autenticação: Passaporte

## O que é Passaporte?

O Passport é um gerador de estratégias de autenticação e autorização, para manter um código limpo, estruturado e altamente configurável.

Podemos usar e configurar várias estratégias de autenticação e autorização com o passaporte. Desta vez vamos criar uma estratégia local.

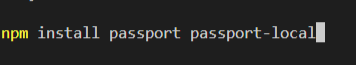
### Reestruturando nosso sistema de cadastro e login com o Passport-local

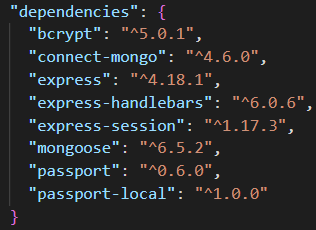
Configuração inicial

O Passport deve ser instalado em dois módulos:

* O primeiro módulo é o núcleo do Passport.
* O segundo módulo é a estratégia a ser usada.

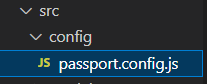
Nossa instalação será feita então:



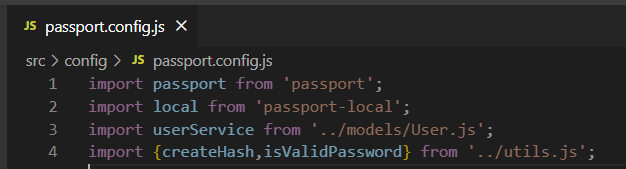


### Definimos o arquivo de configuração

Vamos criar um arquivo passaporte.config.js em uma pasta de configuração:



Este arquivo terá os elementos a serem utilizados importados. Observe que o usuário e a lógica de hash irão para este lado, então importamos as funções userService e bcrypt.



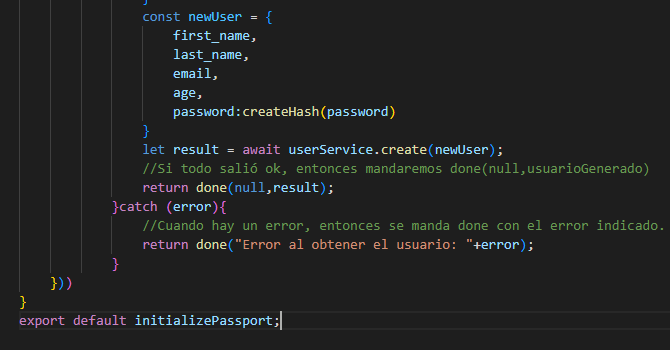
## noções importantes

* O passaporte local sempre exigirá duas coisas: nome de usuário e senha. Se o passaporte não encontrar nenhum desses dois elementos, retornará um erro e não permitirá que você prossiga com a estratégia.
* Podemos alterar o campo "nome de usuário" para que ele ocupe o campo que queremos como identificador, neste caso não estamos interessados em nosso nome de usuário, estamos realmente interessados no e-mail, portanto, podemos alterá-lo com {usernameField : 'valor'}
* O Passport usa um retorno de chamada “concluído”, que é resolvido da seguinte forma:
* O primeiro parâmetro de done é o erro, se passarmos done(null) indicamos que não há erro.
* O segundo parâmetro deve ser o usuário gerado, então para retornar um usuário, fazemos done(null, user).
* Se passarmos done(null, false) indicamos que não há erro, mas o usuário não estará disponível.
* Cada estratégia que queremos configurar no passaporte é um middleware por conta própria, então usaremos o elemento passaporte.use() para configurar diferentes middlewares/estratégias.

## Geração da Estratégia de Cadastro: Parte I

## 

## Geração da Estratégia de Registro: Parte II

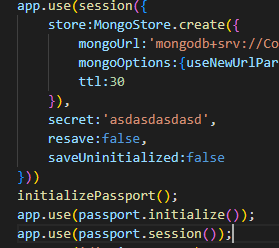


## Vamos reestruturar app.jsinicialização que criamos em nossa configuração, para uso em app.js

## 

Primeiro, vamos importar o núcleo do passaporte e a função de 

Agora, na seção onde declaramos todos os nossos middlewares, vamos adicionar a inicialização conforme indicado nas últimas linhas

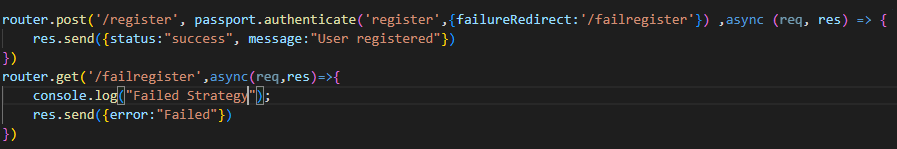


## O roteador também exigirá alterações

Agora, para uma implementação de passaporte simples, uma estratégia geralmente tem dois caminhos: um caminho de processo principal e outro caminho de "escape" caso o processo falhe em algum ponto.

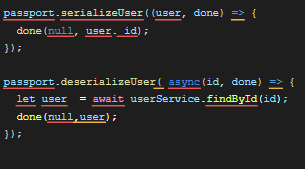
A rota principal chamará o middleware do passaporte especificamente na estratégia que solicitamos.

**Reestruturar caminhos de registro**

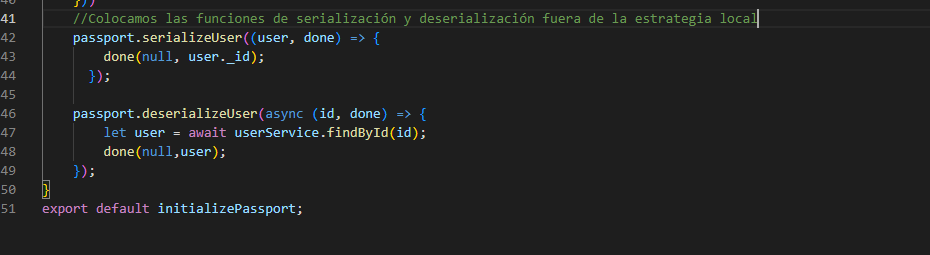


## serializar e desserializar

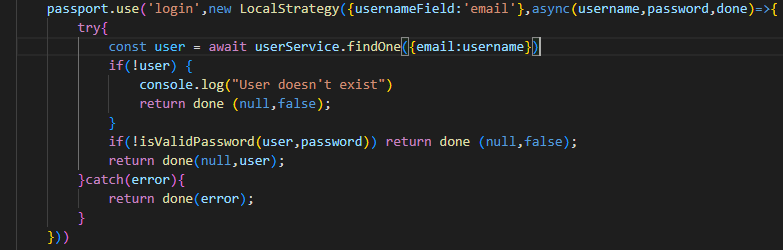
* Para restaurar o estado de autenticação por meio de solicitações HTTP, o Passport precisa serializar os usuários e desserializá-los fora da sessão.
* Isso é feito para que cada solicitação subsequente não contenha as credenciais do usuário anterior.
* Geralmente é implementado fornecendo o ID do usuário ao serializar e consultar o registro do usuário por ID do banco de dados ao desserializar.
* Os métodos que o Passport fornece para isso são serializeUser e desserializeUser.
* O código de exemplo para ambos os métodos é mostrado abaixo.
* Você pode ver que o método serializeUser usa o ID do usuário e o deserializeUser



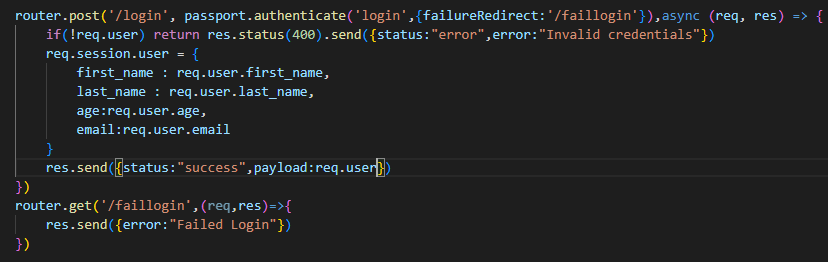
## Serializador e desserializador geral (aplica-se a todas as estratégias, definidas fora)



## Criação de estratégia de login



## Mudanças de caminhos de login



## E... Alguma coisa mudou?

Se fizermos as coisas corretamente, não perceberemos nenhuma mudança. O cadastro terá sido feito exatamente igual a primeira implementação feita. Não fizemos essa reestruturação para obter algum resultado novo

O motivo de implementar essas estratégias é deixar a lógica de autenticação e autorização para o passaporte para que tudo seja controlado em uma camada interna.

Em futuras aplicações suas ou de sua empresa você terá que autenticar seus usuários de MUITAS formas, então é melhor controlar todas essas estratégias em um módulo específico para isso.