Protegendo rotas da sua aplicação back-end com JWT

Durante do desenvolvimento de uma aplicação web um ponto de extrema importância é a segurança, principalmente se sua aplicação tiver usuários ou trabalhar com cadastro de informações públicas, por exemplo, imagine que você está desenvolvendo uma aplicação que seja basicamente um CRUD, esse CRUD possui um painel administrativo onde você pode cadastrar informações que são renderizadas na sua página Home, obviamente essa rota onde você pode fazer cadastro de dados não pode ficar exposta para qualquer um, afinal de contas, qualquer pessoa poderia fazer cadastros na sua aplicação e bagunçar as informações que são renderizadas na Home, por esse motivo e outros nós fazemos a proteção de determinadas rotas da nossa aplicação.

Tanto rotas na nossa API quanto rotas no nosso front-end devem ser protegidas se forem sensíveis, mas como nós podemos fazer a proteção dessas rotas e deixar elas privadas? Geralmente esse tipo de proteção é feita usando um token, esse token serve para validar se o usuário que está tentando acessar tal rota tem autorização para isso ou não.

Se você em algum ponto, enquanto estava desenvolvendo um projeto, se preocupou em como poderia fazer para proteger certas funcionalidades do seu código para que apenas pessoas autorizadas pudessem acessá-las, bom, é isso que vamos aprender hoje.

Antes, você pode estar se perguntando: "O que raios é JWT", porque ta bem ali no título, eu vou te explicar, de forma bem resumida o JWT é um padrão da indústria que serve para transmitir ou armazenar de forma segura objetos JSON, ele é usado tanto para trabalhar com autorizações quanto para trabalhar com troca de informações. Se você quiser se aprofundar mais e entender melhor sobre JWT eu te indico esse artigo da Dev media: https://www.devmedia.com.br/como-o-jwt-funciona/40265

Nesse artigo o meu foco é te mostrar na prática como fazer para proteger as suas rotas usando esse carinha(JWT), então vamos começar.

No back-end

Como vamos fazer tudo de forma prática eu vou te falar as techs que vou usar aqui nesse artigo, pro back-end vai ser Node, Express, TypeScript, Mongoose, cors, bcrypt e o JWT. Com relação ao banco de dados você pode ter ele na sua máquina ou usar o Atlas, que é o serviço do MongoDB em cloud que inclusive é de graça, se você quer saber como proteger suas rotas com JWT acredito que já tem noções de como conectar sua aplicação com banco de dados, então vou pular para o que interessa.

Primeiro vamos criar o projeto e fazer a instalação de todas as dependências, é só me seguir, eu vou usar o yarn, mas sinta-se a vontade para usar o gerenciador de pacotes de sua preferência.

Instalando dependências

1. `yarn init -y`
2. `yarn add express mongoose cors jasonwebtoken bcrypt`
3. `yarn add typescript @types/node @types/express @types/bcrypt @types/cors @types/jsonwebtoken tsx -D`
4. `yarn tsc --init`

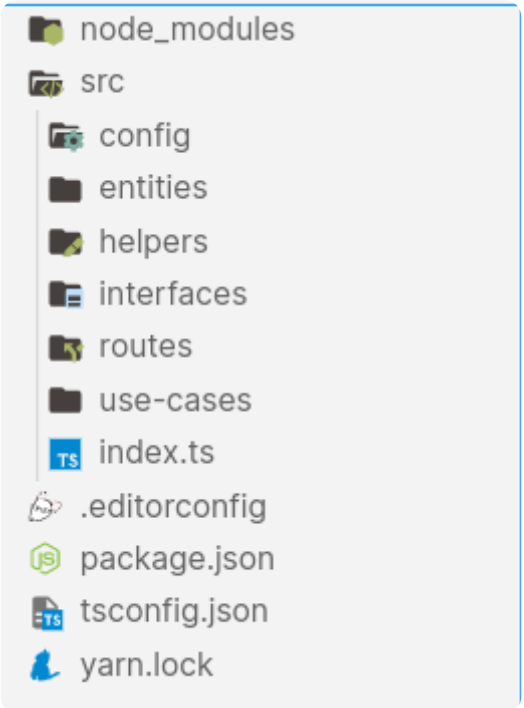
Aqui está o meu tsconfig.json caso queira deixar igual

```json
{
    "compilerOptions": {
        "target": "ES2020",
        "lib": ["ES2020"],
        "module": "ES2020",
        "rootDir": "./",
        "moduleResolution": "Node",
        "outDir": "./build",
        "esModuleInterop": true,
        "forceConsistentCasingInFileNames": true,
        "strict": true,
        "noImplicitAny": false,
        "skipLibCheck": true
    }
}
```

Depois de instalar todas as dependências vamos iniciar a configuração base do nosso Express, para testar as rotas da API irei usar o Insomnia, mas novamente, fique a vontade para usar o software que preferir.

Configuração base

Primeiro vamos criar uma pasta 'src' para colocar todo nosso código dentro, nesta pasta criaremos uma estrutura de pastas com config, onde faremos a configuração do banco de dados, entities, que serão nossas entidades dentro do banco, helpers, que serão funções para usos específicos, interfaces, para nossas interfaces do TS, routes, onde vamos configurar as rotas, use-cases, que serão nossos casos de uso para cada rota e um arquivo index.ts, no final de tudo nossa estrutura de pastas fica assim:



Depois de montarmos nossa estrutura vamos dar inicio no index.ts.

Faremos um setup inicial do express, dessa forma:

```typescript
import express from 'express'
import cors from 'cors'

const port = 3000
const app = express()
```

```
app.use(cors())
app.use(express.json())

app.listen(port, (): void => {
    console.log(`Server is running on port ${port}`)
})
```

No nosso package.json vamos adicionar o script dev:

```json
"scripts": {
    "dev": "tsx watch src/index.ts"
}
```

Depois disso podemos testar nossa aplicação com `yarn run dev` se tudo der certo o seu console irá mostrar a mensagem 'server is running on port 3000'.

Conectando ao banco

Na nossa pasta config vamos criar um arquivo db.ts e fazer a seguinte configuração:

Eu optei por usar o banco de dados na minha própria máquina, então a configuração fica dessa forma, depois da porta 27017 adicionamos o nome do banco de dados.

```typescript
import mongoose from 'mongoose'

// Essa parte serve para retirar um warning do mongoose e é opcional
mongoose.set("strictQuery", false);

async function main(): Promise<void> {
    await mongoose.connect('mongodb://localhost:27017/authjwt')
    console.log('Banco de dados conectado com sucesso!')
}

main().catch((err) => console.log(err))
```

Após isso não esquecer de importar o db.ts no index.ts dessa forma:

`import './config/db.js'`

Se tudo der certo deverá aparecer no seu console também a mensagem 'Banco de dados conectado com sucesso'.

Criando Entities

No nosso exemplo vamos criar apenas uma Entidade User, vai ser na criação de um novo usuário que um Token será gerado.

Primeiramente vamos criar uma interface, na pasta de interfaces crie IUser.ts, dentro desse arquivo criaremos a seguinte interface:

```typescript
export interface IUser {
    _id?: string,
    name: string,
    login: string,
    password: string,
    confirmpass: string
}
```

Agora importamos ela dentro da nossa pasta entities no nosso arquivo user.ts e criamos nossa entidade usuários.

```typescript
import { Schema, model } from 'mongoose'
import { IUser } from '../interfaces/IUser'

const user = new Schema<IUser>({
    name: { type: String, required: true },
    login: { type: String, required: true },
    password: { type: String, required: true }
})

export const User = model<IUser>('user', user)
```

Agora temos acesso ao User e podemos registrar usuários com ele.

Criando rotas

Agora dentro da pasta routes vamos criar o arquivo userRoutes.ts, nesse arquivo vamos fazer uma pequena configuração inicial:

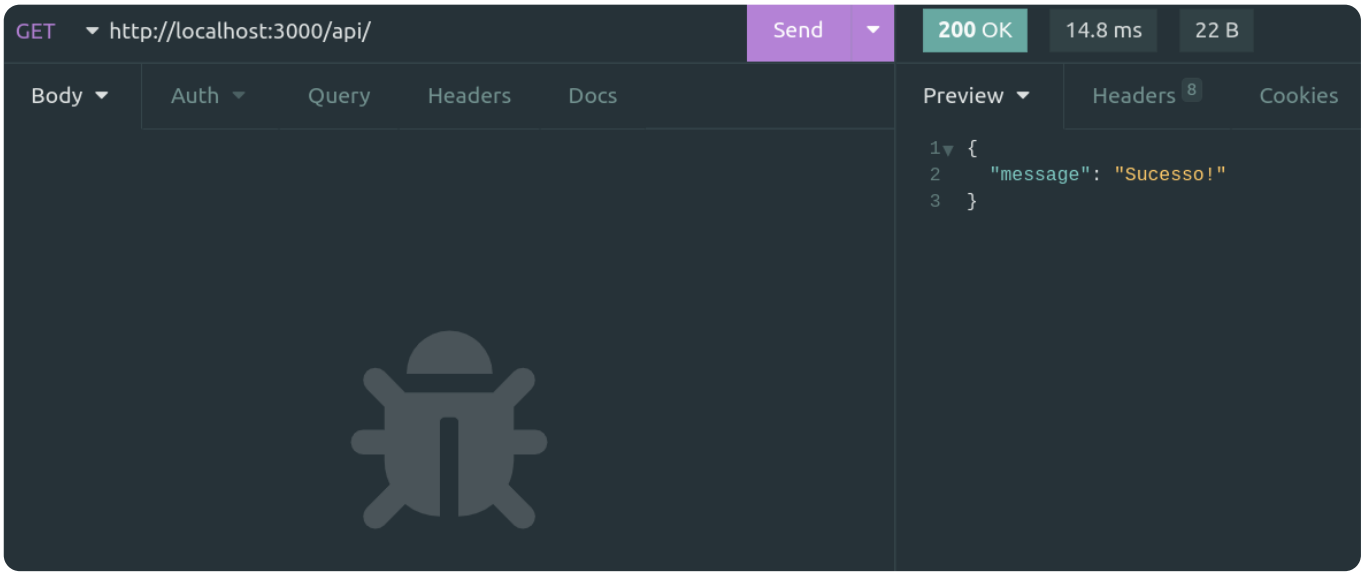
```typescript
import { Router } from 'express'

export const userRouter = Router()

userRouter.get('/', (req,res) => {
    res.status(200).json({ message: 'Sucesso!' })
})
```

Criamos uma rota get apenas para teste, agora vamos importar no nosso index.ts

```typescript
import express from 'express'
import cors from 'cors'

import './config/db.js'

import { userRouter } from './routes/userRoutes.js' // NOVA LINHA

const port = 3000
const app = express()

app.use(cors())
app.use(express.json())

app.use('/api', userRouter) // NOVA LINHA

app.listen(port, (): void => {
    console.log(`Server is running on port ${port}`)
})
```

Dessa forma todas as nossas rotas serão '/api/rotas criadas'

Abrindo o insomnia e dando um get na rota:
'http://localhost:3000/api/' teremos nossa mensagem 'sucesso!'.



Agora vamos criar uma rota para registrar nossos usuários, para isso iremos na pasta use-cases e criaremos o userController.ts.

No useController.ts criaremos uma função assíncrona para registrar nosso usuário no banco, dessa forma:

```typescript
import { User } from '../entities/user'
import { Request, Response } from 'express'
import bcrypt from 'bcrypt'

export const createUser = async (req: Request, res: Response) => {
const {name, login, password, confirmPass } = req.body

// Validations
if(!name || !login || !password || !confirmPass) {
 return res.status(422).json({ message: "Por favor preencha todos os campos" })
}

if(password !== confirmPass) {
 return res.status(422).json({ message: "As senhas não coincidem" })
}

// Check if user exists
const userExists = await User.findOne({ login: login })

if(userExists) {
 return res.status(422).json({ message: "Usuário já cadastrado!" })
}

// Create password
const salt = await bcrypt.genSalt(12)
```

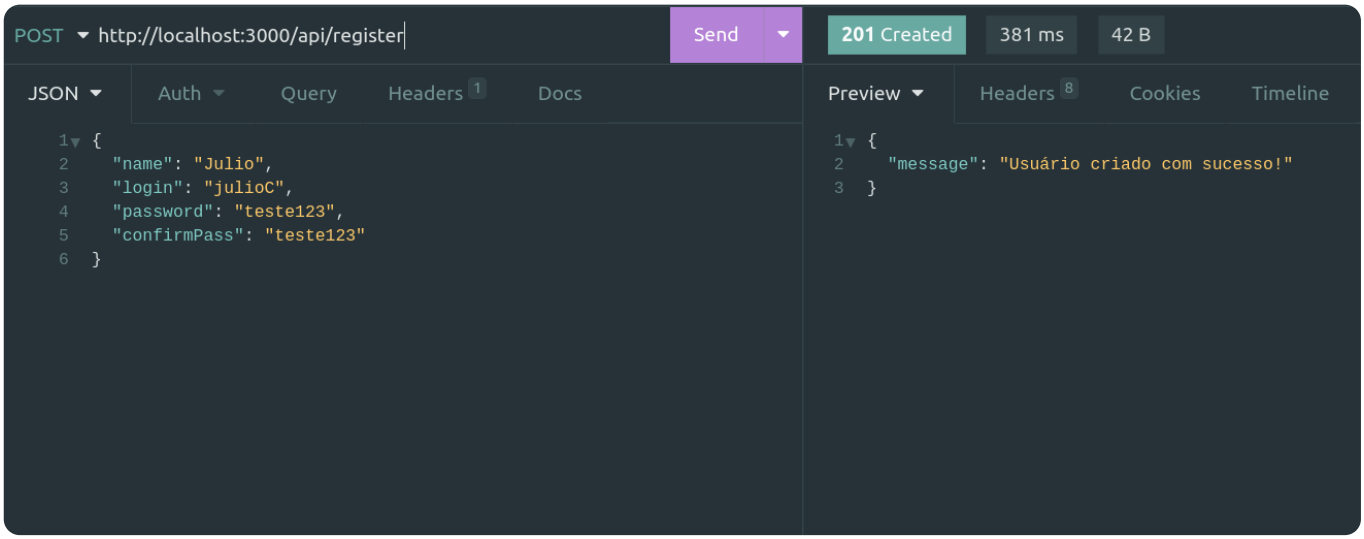
```
const passwordHash = await bcrypt.hash(password, salt)

// Create user
const user = new User({
 login,
 name,
 password: passwordHash
})

try {
 await user.save()
 res.status(201).json({ message: "Usuário criado com sucesso!" })
} catch(err) {
 console.log(err)
 res.status(500).json({ message: "Algo deu errado, tente mais tarde!" })
}
}
```

Com essa função nós criamos o usuário no banco de dados com todas as validações feitas e de quebra ainda te ensinei a como salvar senhas criptografadas no banco com o bcrypt ;)

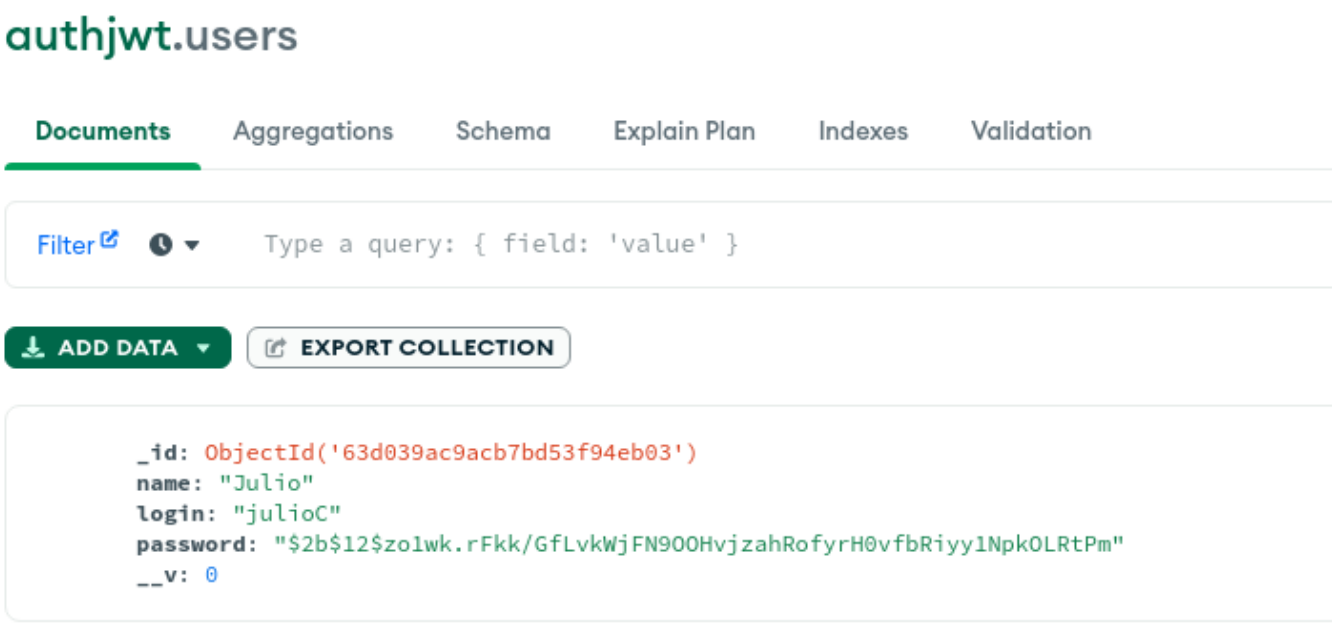
Após essa função estar pronta vamos voltar no userRoutes.ts e criar uma rota.

```
TYPESCRIPT
import { Router } from 'express'
import { createUser } from '../use-cases/userController' // NOVA LINHA

export const userRouter = Router()

userRouter.get('/', (req,res) => {
 res.status(200).json({ message: 'Sucesso!' })
})

userRouter.post('/register', createUser) // NOVA LINHA
```

Agora podemos voltar no insomnia e testar nossa rota!



Podemos inclusive usar o compass para acessar nossas informações salvas no banco de dados, e tanto o usuário foi salvo como sua senha salva foi a senha criptografada.



Ok, agora precisamos usar o jwt para criar um token que sirva para autenticar esse usuário, nós vamos criar uma função que será a create-user-token.ts que irá gerar um token para o usuário quando ele se registrar e também quando ele fizer o login, tendo esse token nós poderemos privar nossas rotas através de um middleware, mas vamos por partes.

Gerando token com JWT

Vamos lá, para criar um token vamos na pasta helpers e criaremos o arquivo create-user-token.ts, em seguida vamos criar a seguinte função:

```typescript
import { Request, Response } from 'express'
import { IUser } from '../interfaces/IUser.js'
import jwt from 'jsonwebtoken'

export const createUserToken = async(user: IUser, req: Request, res: Response) => {
 const secret: string = 'JUYDAiudha2437y4214O(*^(*&^(*^(aiuwdau'

 const token = jwt.sign({
  name: user.name,
  id: user._id
 }, secret)

 res.status(200)
 .json({ message: "Usuário autenticado com sucesso!", token: token })

}
```

Agora um ponto de ATENÇÃO, perceba a variável secret, eu mesmo criei aquele hash maluco, o token será criado com base naquela sequência louca de caracteres que eu coloquei de forma aleatória, faça o mesmo, crie uma sequência de caracteres aleatórios, não precisa seguir a minha, o ideal é que você use um .env para armazenar o seu segredo e não deixe ele exposto dessa forma, porém como esse é um exemplo didático podemos deixar assim.

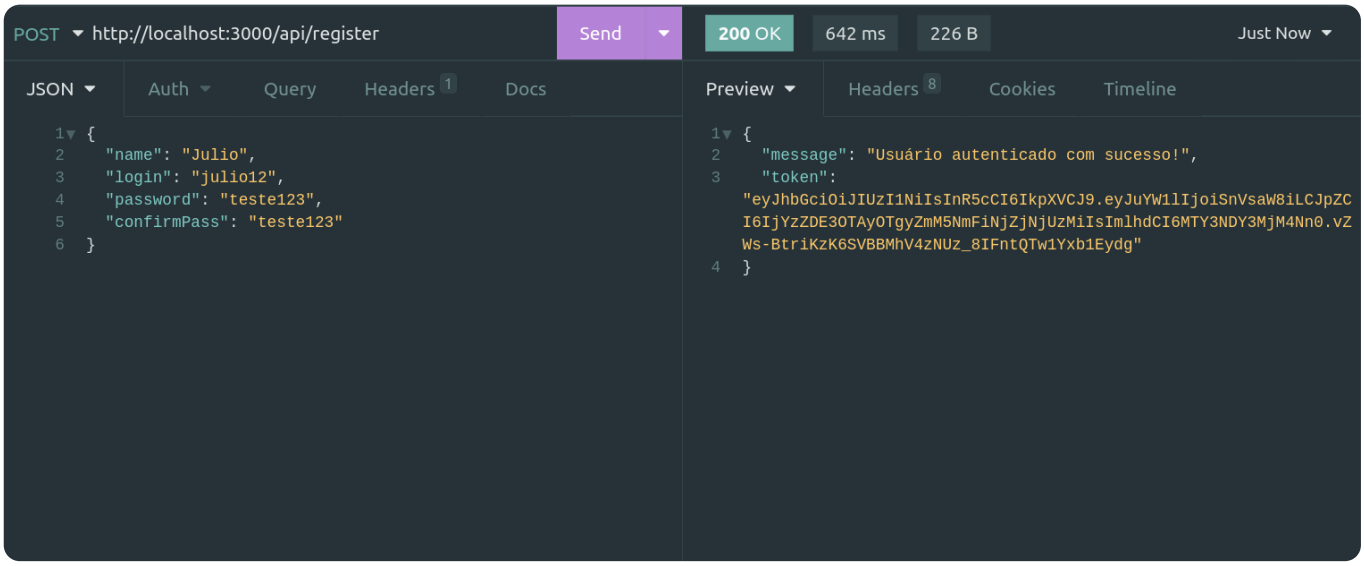
Certo, agora temos uma função que cria um token aleatório para o usuário e podemos adicioná-la a nossa função de registro de usuário, vamos lá:

No nosso use-cases, no arquivo userController.ts na função de createUser vamos mudar o seguinte:

```typescript
import { User } from '../entities/user'
import { Request, Response } from 'express'
import { createUserToken } from '../helpers/create-user-token' // NOVA LINHA
import bcrypt from 'bcrypt'

export const createUser = async (req: Request, res: Response) => {
const {name, login, password, confirmPass } = req.body
```

```
// Validations
if(!name || !login || !password || !confirmPass) {
 return res.status(422).json({ message: "Por favor preencha todos os campos" })
}

if(password !== confirmPass) {
 return res.status(422).json({ message: "As senhas não coincidem" })
}

// Check if user exists
const userExists = await User.findOne({ login: login })

if(userExists) {
 return res.status(422).json({ message: "Usuário já cadastrado!" })
}

// Create password
const salt = await bcrypt.genSalt(12)
const passwordHash = await bcrypt.hash(password, salt)

// Create user
const user = new User({
 login,
 name,
 password: passwordHash
})

try {
 const newUser = await user.save() // NOVA LINHA
 await createUserToken(newUser, req, res) // NOVA LINHA
} catch(err) {
 console.log(err)
 res.status(500).json({ message: "Algo deu errado, tente mais tarde!" })
}
}
```

Dessa forma quando formos no insomnia e registrarmos um usuário será devolvido para nós o token desse usuário, vamos testar:



Agora quando fazemos o registro do usuário nós recebemos de volta um token, com esse token podemos validar o usuário e privar nossas rotas, porém antes disso vamos criar a nossa função de login.

Criando função de login

No nosso userController.ts vamos criar uma nova função que será a loginUser, vamos lá!

Então abaixo da nossa função de createUser teremos o seguinte:

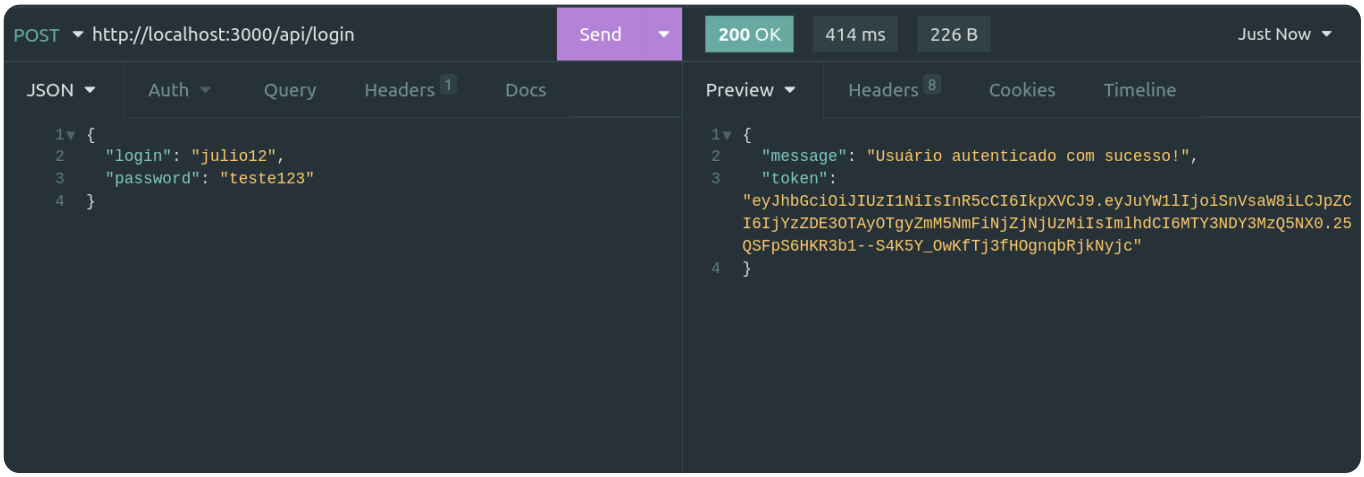
```typescript
export const login = async (req: Request, res: Response) => {
const { login, password } = req.body

// Validations
if(!login || !password) {
 return res.status(422).json({ message: "Por favor preencha todos os campos!" })
}

// Check if user exists
const user = await User.findOne({ login: login })

if(!user) {
 return res.status(404).json({ message: "Este usuário não existe" })
}

// Check if password match
const checkPassword = await bcrypt.compare(password, user.password)
```

```
if(!checkPassword) {
 return res.status(422).json({ message: "As senhas não coincidem" })
}

await createUserToken(user, req, res)

}
```

Vamos agora criar uma rota de login e atribuir essa função para nossa rota.

```typescript
import { Router } from 'express'
import { createUser, login } from '../use-cases/userController' // LOGIN IMPORTADO

export const userRouter = Router()

userRouter.get('/', (req,res) => {
 res.status(200).json({ message: 'Sucesso!' })
})

userRouter.post('/register', createUser)
userRouter.post('/login', login) // NOVA LINHA
```

Agora vamos até o insomnia testar nossa função de login.



Pronto! Dessa forma nós também devolvemos um token para autenticar o usuário quando ele fizer login.

Muito bem! Agora que já temos o nosso token criado nós já podemos privar nossas rotas para que apenas usuários com tokens válidos possam acessá-las, para isso vamos criar mais duas funções dentro dos nossos helpers.

Criando função getToken

Primeiro de tudo vamos criar dentro da nossa pasta helpers uma função para pegar o token dos headers, é uma função muito simples que podemos reaproveitar sempre que quisermos coletar o token do usuário.

Faremos dessa forma:

```typescript
export const getToken = (req) => {
 const authHeader = req.headers["authorization"]
 const token = authHeader.split(" ")[1]

 return token

}
```

Pronto! Dessa forma nós conseguimos coletar o token que vem do nosso header.

Agora vamos criar um middleware que vai validar o nosso token!

Criando verifyToken

Esse middleware vai ser o responsável por privar nossas rotas, ele irá verificar se quem está tentando acessar tal rota possui um token e se esse token é realmente válido, vamos lá!

Faremos dessa forma:

```typescript
import jwt from 'jsonwebtoken'
import { getToken } from './get-token'

export const verifyToken = (req, res, next) => {
 const secret: string = 'JUYDAiudha2437y4214O(*^(*&^(*^(aiuwdau'

 if(!req.headers.authorization) {
  return res.status(401).json({ message: "Acesso negado!" })
 }

 const token = getToken(req)

 if(!token) {
```

```
  return res.status(401).json({ message: "Acesso negado!" })
 }

 try {
  const verified = jwt.verify(token, secret)
  req.user = verified
  next()
 } catch(err) {
  console.log(err)
  res.status(400).json({ message: "Token inválido!" })
 }

}
```

Lembre-se de definir o mesmo secret que você colocou na função create-user-token.ts.

Ok, com esse middleware criado nós podemos agora fazer os testes definitivos para privar nossas rotas, vamos primeiro criar uma rota exemplo, que por algum motivo não queremos que usuários não autenticados possam acessar.

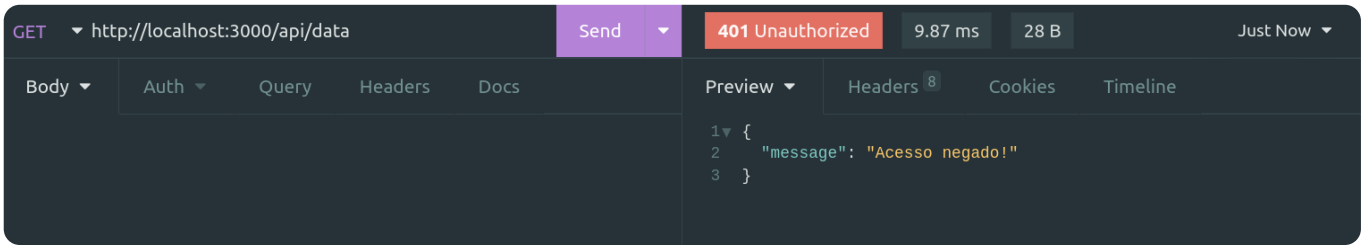
Muito bem, no nosso arquivo userRoutes.ts eu fiz o seguinte:

TYPESCRIPT

```typescript
import { Router } from 'express'
import { createUser, login } from '../use-cases/userController'

import { verifyToken } from '../helpers/verify-token.js' // NOVA LINHA

export const userRouter = Router()

userRouter.get('/', (req,res) => {
 res.status(200).json({ message: 'Sucesso!' })
})

userRouter.post('/register', createUser)
userRouter.post('/login', login)

// NOVA ROTA
userRouter.get('/data', verifyToken, (req, res) => {
 res.status(200).json({ message: "informação sensível" })
})
```

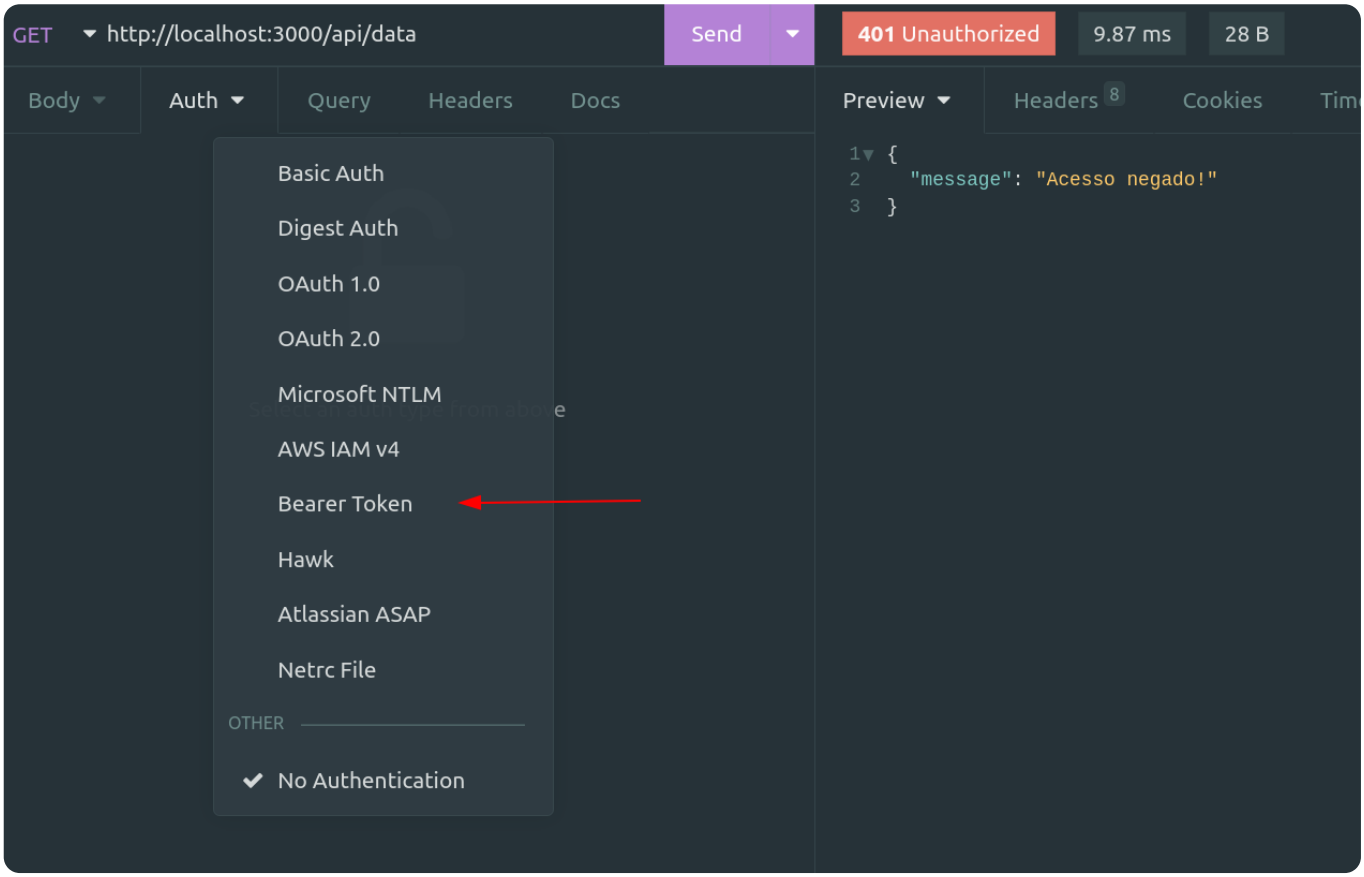
Adicionei uma rota qualquer apenas para testarmos, perceba que adicionei entre o caminho e a função que retorna um json o nosso

verifyToken, para que ele atue como middleware, se tentamos acessar essa rota /data sem um token veja o que acontece:

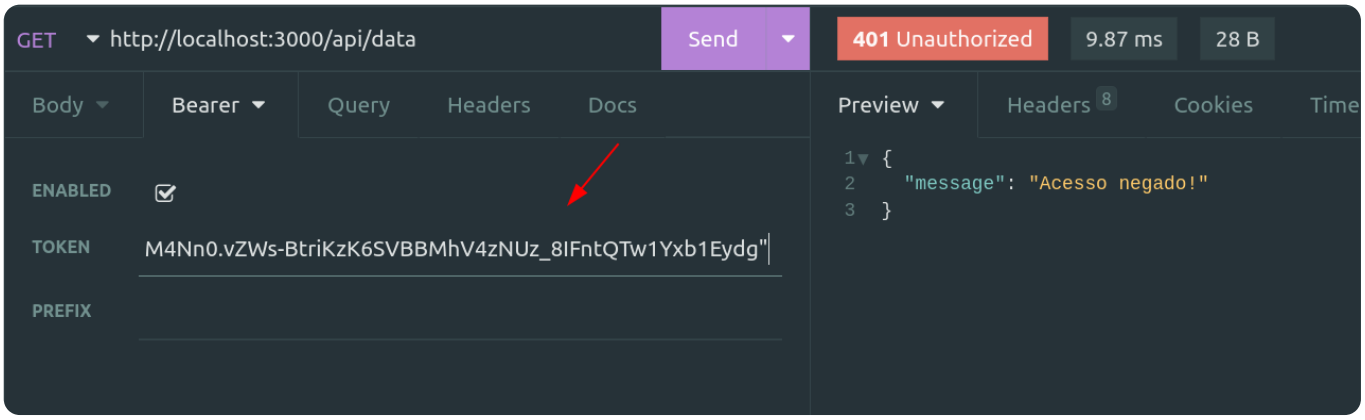


Agora vamos adicionar um token válido ao insomnia, vamos pegar um dos tokens que é gerado para nós quando registramos um usuário ou fazemos login e nós iremos inserir ele no insomnia da seguinte forma:

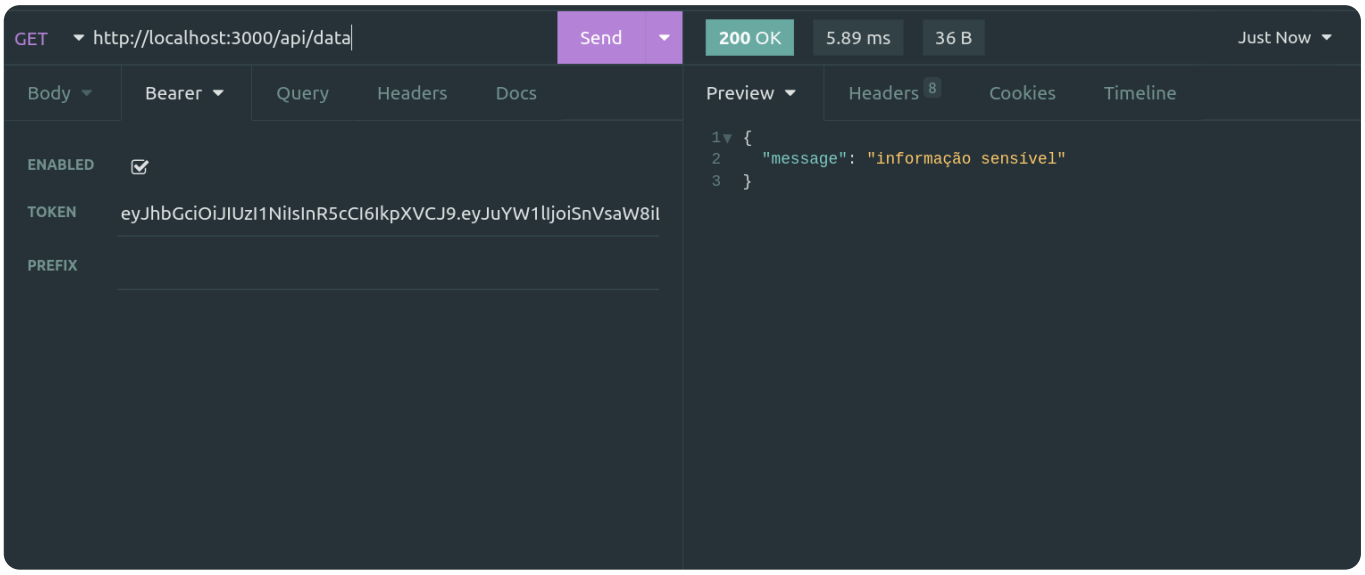
Vamos até a opção Auth e escolheremos Bearer token



Entrando lá vamos adicionar um dos tokens válidos
OBS: Lembre de remover as aspas, adicione apenas o token!



Agora quando damos novamente um "send" veja o que acontece:



Conseguimos acessar com sucesso a nossa rota apenas quando utilizamos um token válido.

Conclusão

Bom, chegando aqui nós concluímos nosso objetivo, protegemos uma rota do nosso back-end usando autenticação com JWT através de um Token, com o middleware de verifyToken você poderá privar qualquer rota que quiser.

Espero ter ajudado no seu seu projeto e/ou aprendizado.