

PROJETO API REST

Vamos criar um projeto que podemos visualizar quais são os alunos cadastrados.

Diferente de como já vimos como montamos o EXPRESS, iremos criar ele em classes

```
import dotenv from 'dotenv';
dotenv.config();
import express from 'express';
import routerHome from './routers/homeRouters.js';
import tokenRouter from './routers/tokenRouters.js'

class App{
  constructor(){
    this.app = express();
    this.middleware();
    this.routes();
  }
  middleware(){
    this.app.use(express.urlencoded({extended: true}));
    this.app.use(express.json());
  }
  routes(){
    this.app.use(routerHome);
    this.app.use(tokenRouter);
  }
}
export default new App().app;
```

Basicamente, iremos criar um arquivo que contem as funcionalidades do server. Chamaremos de app.js

Com isso, iremos criar um arquivo que server.js que irá só escutar e fazer com que o servidor fique no ar

```
src > server.js > ...
1 import app from './app.js';
2
3 app.listen(3000, ()=>{
4   console.log('Server on');
5   console.log('Access server in: http://localhost:3000');
6 })
7
```

O que acontece nesses arquivos: Nota-se que o app.js já exporta instanciado para o server.js e já iniciando suas funções de middleware e router

```
src > routers > .js homeRouters.js > ...
1  import { Router } from 'express';
2  import {aluno, user, update, deleteUser, userHome} from '../controllers/dbControll.js';
3  import tokenControll from '../controllers/tokenControll.js';
4  import LoginRequired from '../middleware/LoginRequired.js';
5
6  const router = new Router();
7
8  router.get('/', LoginRequired, aluno);
9  router.post('/', tokenControll.store);
10 router.post('/create', user)
11 router.get('/home', LoginRequired, userHome);
12 router.put('/update/:id', LoginRequired, update)
13 router.delete('/delete/:id', LoginRequired, deleteUser);
14
15 export default router;
```

Agora, iremos para o modelos do banco de dados, já falamos sobre em outro arquivo como funciona. Só que adicionamos uma mecânica a mais, que é quando a pessoa faz login, ele recebe um token, esse token é requerido em todas as outras rotas, se a pessoa não passar esse token, o acesso será negado.

Como estamos usando o INSOMNIA para fazer as requisições, a parte de fazer login, pode ser um pouco diferente do que da ultima vez.

Primeiro, iremos criar um model separado – para autenticar os dados e adquirir o token.

Vamos usar o modelo como nós aprendemos, validar e inserir os dados em um objeto após a validação

```

1  import db from './model.js';
2  import validator from 'validator';
3  import bcrypt from 'bcrypt';
4
5  class Login{
6    constructor(req){
7      this.body = req;
8      this.error = [];
9      this.user = null
10   }
11   async autenticar(){
12     this.inserir();
13     const [userDb] = await db.execute(`
14       SELECT * FROM user WHERE email = ?
15     `, [this.user.email]);
16     const userFormat = userDb[0];
17     if(!userFormat){
18       this.error.push('Email não encontrado');
19       return
20     }
21     const hash = bcrypt.compareSync(this.user.senha, userFormat.password_hash);
22     if(!hash){
23       this.error.push('Senha incorreta');
24       return
25     }
26     return userFormat;
27   }
28   async inserir(){
29     this.user = {
30       email: this.body.email,
31       senha: this.body.senha
32     }
33   }
34
35 }
36 export default Login;
37 //criar duas funções: validar e inserir

```

Ah... um detalhe, na rota que criamos o usuário, adicionamos o hash e o salt de uma vez com a função .hash()

```

async createUser(nome, email, pass){
  try{
    const pass_hash = await bcrypt.hash(pass, 10);
    const [createdResult] = await db.execute(`
      INSERT INTO user(user_name, email, password_hash) VALUES (?, ?, ?)
    `, [nome, email, pass_hash]);
    const [showUser] = await db.execute(`
      SELECT * FROM user WHERE id = ?
    `, [createdResult.insertId]);
    return showUser[0];
  }catch(e){
    console.log('erro em criar usuario ', e);
  };
};

```

No projeto passado, havíamos usado o `.genSaltSync()` para gerar o salt e o `hashSync` para gerar a hash junto com o salt.

Depois disso, iremos agora criar um arquivo de controlador específico para o token que após o login, a pessoa adquire um token.

Primeiro, iremos baixar a biblioteca `JSONWebToken` e importar para nosso projeto

```
src > controllers > JS tokenControll.js > ...
1  import login from '../models/tokenModel.js';
2  import jwt from 'jsonwebtoken';
3
4  class TokenController{
5    async store(req, res){
6      const user = new login(req.body);
7      const userInsert = await user.autenticar()
8      if(user.error.length > 0){
9        res.json({Message: user.error});
10       return;
11     }
12     const {id, email} = userInsert
13     const token = jwt.sign({id, email}, process.env.TOKEN_SECRET, {
14       expiresIn: process.env.TOKEN_EXPIRATION
15     });
16     res.json({Message: 'Login realizado', Token: token});
17   }
18 }
19 export default new TokenController();
```

OBS: o dados do token está no arquivo `.env`

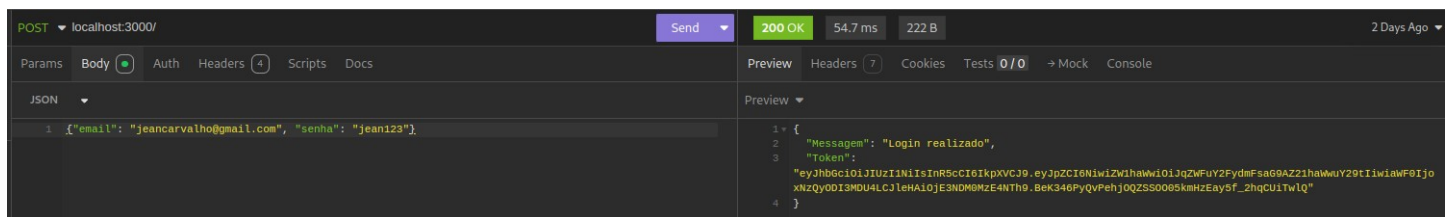
```
1  DATABASE=aluno
2  DATABASE_HOST=localhost
3  DATABASE_PORT=3307
4  DATABASE_USERNAME=root
5  DATABASE_PASS=2002
6
7  TOKEN_SECRET=ASd1123HZ870ASDBalhdsFAGkjdSauu!_q1we123k87ip
8  TOKEN_EXPIRATION=7d
```

`Token_secret` armazena tipo uma assinatura, palavra-chave do nosso token, enquanto o `token_expiration` só informa o tempo de expiração


```
const token = jwt.sign({id, email},
  process.env.TOKEN_SECRET, {
```

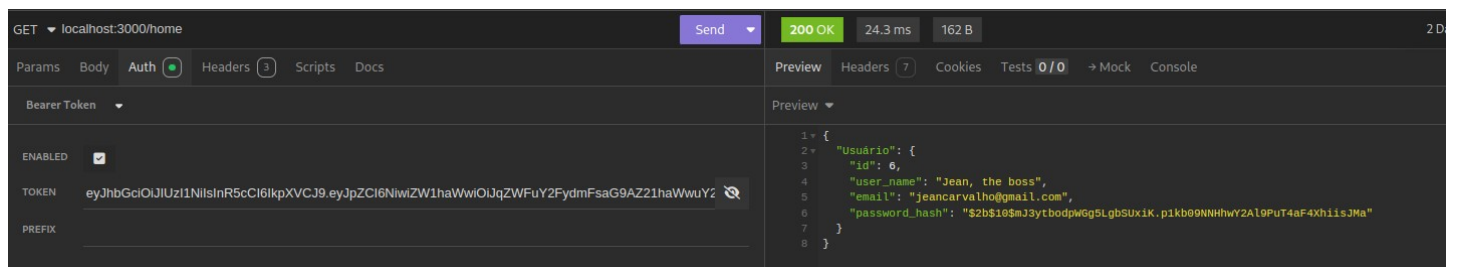


Nela será obtido todas informações no qual pode ser extraído



Criando um middleware

```
src > middleware > .js LoginRequired.js > ...
1 import jwt from 'jsonwebtoken';
2
3 export default (req, res, next) =>{
4   const {authorization} = req.headers;
5   if(!authorization) return res.status(401).json({Error: 'Necessário fazer login'});
6   const [bearer, token] = authorization.split(' ');
7   try{
8     const dados = jwt.verify(token, process.env.TOKEN_SECRET);
9     const {id, email} = dados;
10    req.id = id;
11    req.email = email;
12    return next();
13  }catch(e){
14    res.status(401).json({Error: 'Token invalido'});
15  }
16 }
```



```
src > controllers > dbControll.js > userHome
1 import userModel from '../models/dbModel.js';
2
3 const userHome = async(req, res) => {
4   try{
5     const user = new userModel()
6     const idUser = await user.home(req.id);
7     res.json({Usuário: idUser});
8   }catch(e){
9     res.status(404).json({error: 'usuário não encontrado'});
10  }
11 }
```

O token é passado no cabeçalho da requisição, nesse cabeçalho existe uma propriedade que contém authorization, nela iremos extrair o token, mas com ela vem uma palavra chamada “bearer” que significa “portaria”, então iremos criar uma lógica para extrair só a chave. Feito isso, usaremos a função .verify() do JWT, iremos passar o token que extraímos junto com a palavra-chave Com isso, iremos conseguir extrair o id e o e-mail do usuário

```
try{
  const dados = jwt.verify(token, process.env.TOKEN_SECRET);
  const {id, email} = dados;
  req.id = id;
  req.email = email;
  return next();
}catch(e){
  res.status(401).json({Error: 'Token invalido'});
}
```

Com isso, iremos criar um propriedades na requisição que irá conter o id e o e-mail para que nós possamos acessar em nosso controll

```
const userHome = async(req, res) => {  
  try{  
    const user = new userModel()  
    const idUser = await user.home(req.id);  
    res.json({Usuário: idUser});  
  }catch(e){  
    res.status(404).json({error: 'usuário não encontrado'});  
  }  
}
```