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;
```

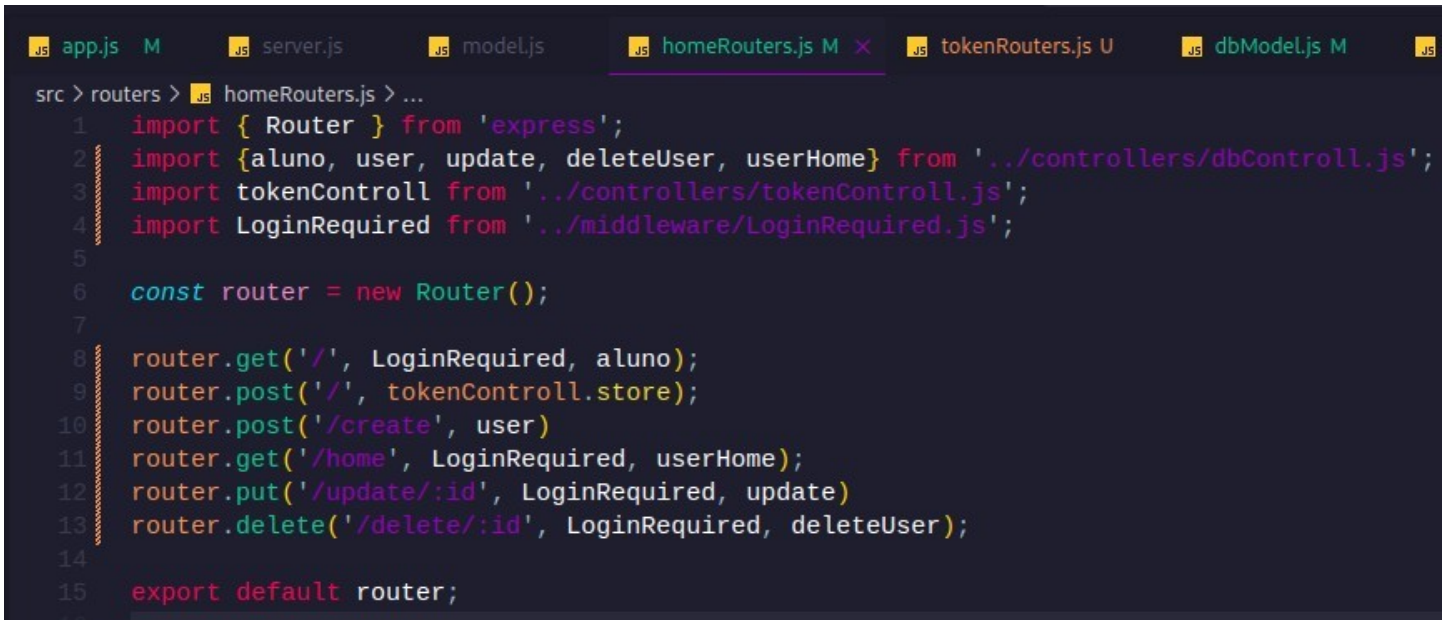
Basicamente, iremos criar um arquivo que contem as funcionalidades do server. Chamaremos de app.js
Com isso, iremos criar um arquivo que server.js que irá só escutar e fazer com que o servidor fique no ar

app.js M | server.js | model.js | homeRouters.js M | tokenR

src > server.js > ...

```
import app from './app.js';

app.listen(3000, ()=>{
  console.log('Server on');
  console.log('Access server in: http://localhost:3000');
})
```

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



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 requirido 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

```
import db from './model.js';
import validator from 'validator';
import bcrypt from 'bcrypt';

class Login{
  constructor(req){
    this.body = req;
    this.error = [];
    this.user = null
  }
  async autenticar(){
    this.inserir();
    const [userDb] = await db.execute(`
      SELECT * FROM user WHERE email = ?
      `, [this.user.email]);
    const userFormat = userDb[0];
    if(!userFormat){
      this.error.push('Email não encontrado');
      return
    }
    const hash = bcrypt.compareSync(this.user.senha, userFormat.password_hash);
    if(!hash){
      this.error.push('Senha incorreta');
      return
    }
    return userFormat;
  }
  async inserir(){
    this.user = {
      email: this.body.email,
      senha: this.body.senha
    }
  }

}
export default Login;
//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, haviamos usados 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 controll especifico para o token que após o login, a pessao adquire um token. Primeiro, iremos baixar a biblioteca JSONWebToken e importar para nosso projeto

src > controllers > tokenControll.js > ...

```
import login from '../models/tokenModel.js';
import jwt from 'jsonwebtoken';

class TokenController{
  async store(req, res){
    const user = new login(req.body);
    const userInsert = await user.autenticar()
    if(user.error.length > 0){
      res.json({Messagem: user.error});
      return;
    }
  const {id, email} = userInsert
  const token = jwt.sign({id, email}, process.env.TOKEN_SECRET, {
    expiresIn: process.env.TOKEN_EXPIRATION
  });
  res.json({Messagem: 'Login realizado', Token: token});
  }
}
export default new TokenController();
```

OBS: o dados do token está no arquivo .env

.env

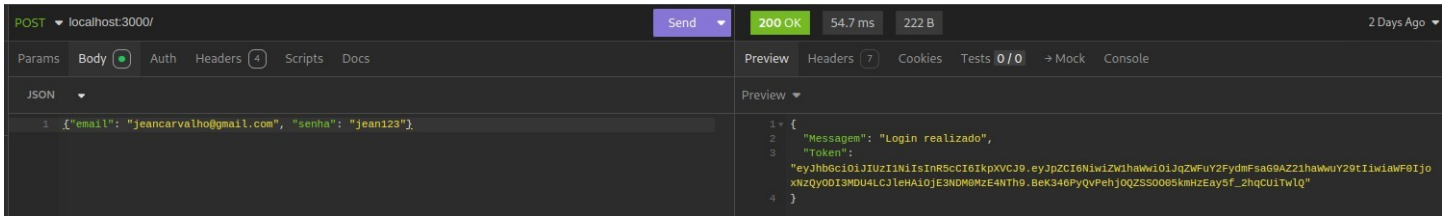
```
DATABASE=aluno
DATABASE_HOST=localhost
DATABASE_PORT=3307
DATABASE_USERNAME=root
DATABASE_PASS=2002

TOKEN_SECRET=ASd1123HZ87OASDBalhsdFAGkjdsauu!_q1we123k87ip
TOKEN_EXPIRATION=7d
```

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

O que acontece, esse jwt.sing() irá criar nossa chave token de acordo com as informações e configurações. Exemplo:

```
const token = jwt.sign({id, email}, process.env.TOKEN_SECRET, {
    expiresIn: process.env.TOKEN_EXPIRATION
});
```

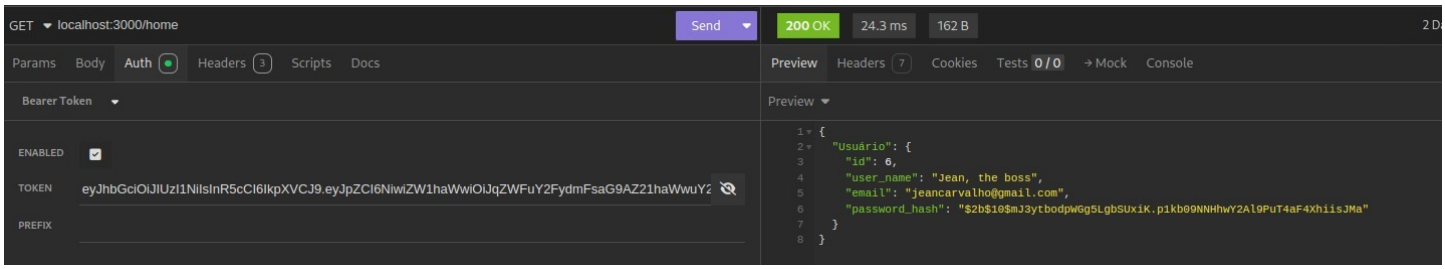
Nota-se que ele recebe três argumentos, primeiro um objeto que irá conter os dados do usuário, segundo é a palavra-chave do token, a terceira é a configuração que também é um objeto, nela contém a propriedade expiresIn: que determina o tempo da validade do token.
Nela será obtido todas informações no qual pode ser extraído



Quando a pessoa fizer o login, adquiriu o token. Agora, como podemos requirir o token nas rotas?
Criando um middleware

src > middleware > LoginRequired.js > ...

```
import jwt from 'jsonwebtoken';

export default (req, res, next) =>{
  const {authorization} = req.headers;
  if(!authorization) return res.status(401).json({Error: 'Necessário fazer login'});
  const [bearer, token] = authorization.split(' ');
  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'});
  }
}
```



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 extraimos junto com a palavra-chave Com isso, iremos conseguir extrair o id e o e-mail do usuário

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

```
  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'});
  }
}
```

1. GET

Função: Solicitar dados de um recurso.

Características:

- Somente leitura (não altera o servidor).
- Parâmetros enviados via URL (query strings).
- Pode ser armazenado em cache

2. POST

Função: Criar um novo recurso ou enviar dados para processamento.

Características:

- Envia dados no corpo da requisição (não na URL).
- Pode ser usado para operações não padronizadas.

3. PUT

Função: Substituir todo um recurso existente.

Características:

- Requer o envio do recurso completo (atualiza todos os campos).
- Se o recurso não existir, pode criá-lo (depende da implementação).

```
Body: { "name": "Alice Updated", "email": "alice_new@example.com" }
```

4. PATCH

Função: Atualizar parcialmente um recurso.

Características:

- Envia apenas os campos a serem modificados.
- Mais eficiente que PUT para pequenas alterações.

```
Body: { "email": "alice_updated@example.com" }
```

5. DELETE

Função: Remover um recurso.

Características:

- Não possui corpo na requisição (geralmente).
- Pode retornar status 204 No Content após exclusão.