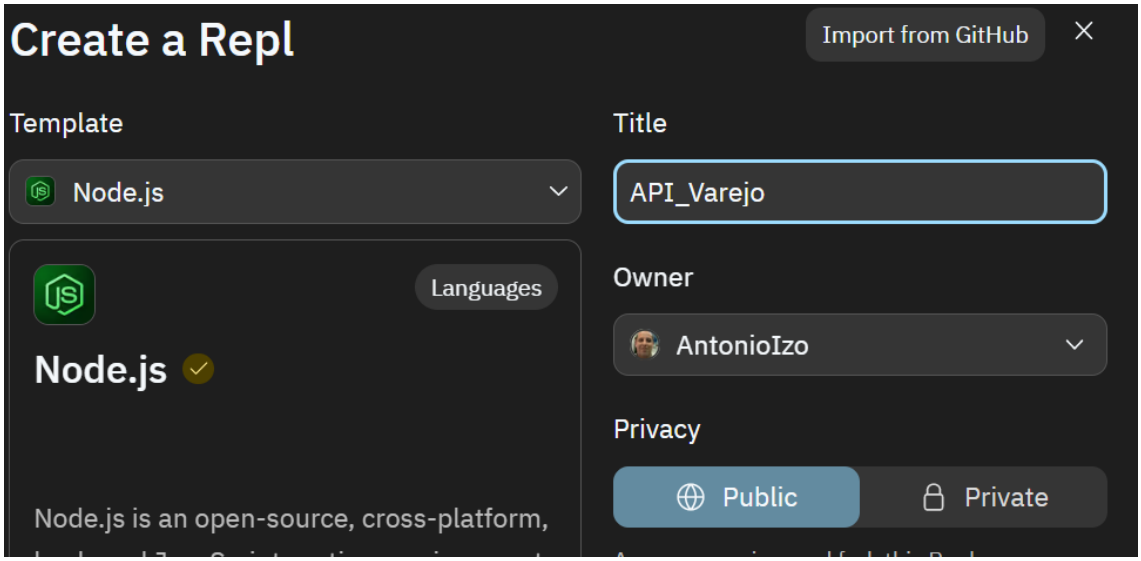
Autenticação com JWT

JSON WEB TOKEN

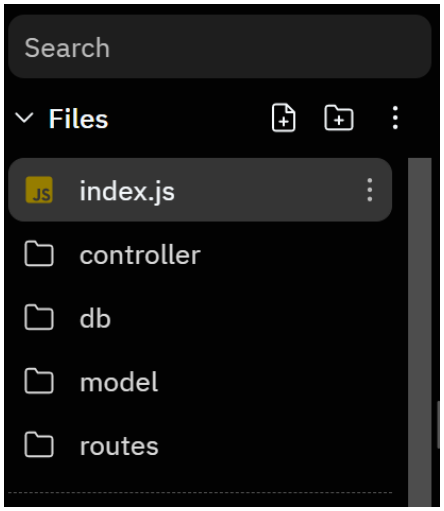
Nesta aula, começaremos a desenvolver uma API com autenticação JWT. Iremos utilizar uma API e aplicação para consumir nossos serviços.

Criando o BackEnd

Coloquei o nome do projeto de API_Varejo para à aplicação BackEnd.



Vamos criar as pasta que iremos utilizar e que terão os nomes de **controller, db, model e routes.**



Siga os passos abaixo para realizar a codificação da nossa API.

Passo 1: Instalar as bibliotecas.

No shell instale as bibliotecas que iremos utilizar na API.

_ Sequelize e sqlite3.

```
~/APIVarejo$ npm install sequelize sqlite3
```

_ express

```
~/APIVarejo$ npm install express
```

Passo 2: Criar as pastas e arquivos com a codificação necessária.

Criação do arquivo **db.js**.

Dentro da pasta db, crie um arquivo chamado db.js com a codificação abaixo.

```
// BIBLIOTECAS/MODULOS UTILIZADOS
const Sequelize = require('sequelize');
//CRIANDO A CONFIGURAÇÃO DO BANCO DE DADOS
const sequelize = new Sequelize({
  dialect: 'sqlite',
  storage: './varejo.sqlite'
})
//TRATANDO POSSÍVEIS ERROS E AUTENTICANDO NO BANCO
try {
  sequelize.authenticate();
  console.log("Banco de dados conectado com sucesso!");
}
catch (erro) {
  console.log("Erro ao conectar ao banco",erro);
}
module.exports = sequelize;
```

O baco de dados terá o nome de **varejo.sqlite.**

O arquivo é igual aos outros que realizamos codificação nas aulas anteriores.

Codificação do **index.js.**

Vamos criar nosso servidor com a criação de acesso a rota principal no arquivo **index.js**.

```
//INSTALAÇÃO BIBLIOTECAS/MÓDULOS
const express = require("express");
const app = express();
const database = require("./db/db");
const routes = require("./routes/routes");
//MODELS


//CODIFICAÇÃO JSON
app.use(express.urlencoded({ extended: true }));
app.use(express.json());


//ROTA PRINCIPAL
app.use("/",routes);

try {
    database.sync().then(() => {
    })
}
catch(erro) {
    console.log("Houve uma falha ao sincronizar com o banco de dados. ", erro);
};

app.listen(3000);
```

Criamos as constantes para express, o databse e routes. Aproveitamos e codificamos os arquivos para uso JSON.

Criação do arquivo **usuarioModel.js**.

Dentro da pasta model, crie um arquivo com o nome **usuarioModel.js** e cole as linhas codificadas abaixo:

```
const Sequelize = require('sequelize');
const database = require('../db/db');
const usuario = database.define('usuario', {
  id_usuario: {
    type: Sequelize.INTEGER,
    autoIncrement: true,
    allowNull: false,
    primaryKey: true
  },
  nome:{
    type: Sequelize.STRING,
```

```
    allowNull: false,
  },
   email:{
    type: Sequelize.STRING,
    allowNull: false,
  },
  senha:{
    type: Sequelize.STRING,
    allowNull:false
  }
}, {database,modelname:'usuario',tableName: 'usuarios'})
module.exports = Usuario;
```

O model do usuário é simples e terá somente os campos nome, email e senha.

Vamos agora a criação do **usuarioController.js**

```
const Usuario = require("../model/usuarioModel");
module.exports = class usuarioController{
  //CREATE
  static async UsuarioCreate(req,res){
    let nome = req.body.nome;
    let email = req.body.email;
    let senha = req.body.senha;
    const usuario = {
      nome: nome,
      email: email,
      senha: senha
    }
    await Usuario.create(usuario);
    res.json({message: "Usuário cadastrado com sucesso!"});
  }
//READ - LISTAR
  static async UsuarioListar(req,res){
    const id_usuario = req.params.id;
    if(id_usuario){
      const usuario = await Usuario.findOne({where: {id_usuario: id_usuario}});
      res.json(usuario);
    }else{
      const usuario = await Usuario.findAll({raw:true});
```

```
      res.json(usuario);
    }
}
//UPDATE
  static async UsuarioUpdate(req, res){
    const id_usuario = req.params.id;
    let nome = req.body.nome;
    let email = req.body.email;
    let senha = req.body.senha;
    const usuario = {
      nome: nome,
      email: email,
      senha: senha
    };
    await Usuario.update(usuario,{where: {id_usuario:id_usuario}});
    res.json({message: "Cadastro atualizado com sucesso! Foram atualizados as
sequintes informações: ", dados: usuario});
  }
//Função UsuarioDelete responsável pela exclusão do usuário.
  static async UsuarioDelete(req,res){
    const id_usuario = req.params.id;
    await Usuario.destroy({where:{id_usuario: id_usuario}});
    res.json({message: "Usuário excluído com sucesso!"});
  }
}
```

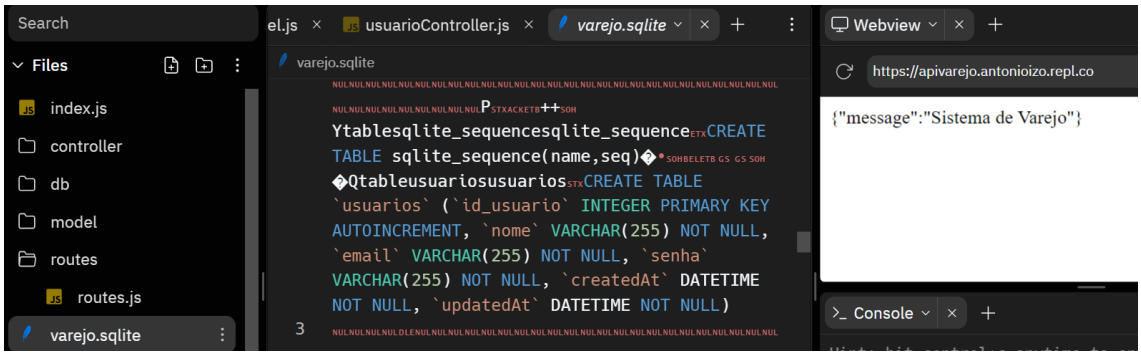
Dentro do usuarioController.js, teremos as funções que são referentes ao CRUD.

Criação do arquivo **routes.js**.

Na pasta routes, crie o arquivo **routes.js.**

```
////////////////MÓDULOS ////////////////////
const express = require("express");
const router = express.Router();
//////CONTROLLERS
const usuarioController = require("../controller/usuarioController");
////////////////Requisições HTTP Principal ////////////////////
router.get("/",(req, res) =>{
  return res.json({message: "Sistema de Varejo"});
})
////////////////Requisições HTTP Usuario ////////////////////

//POST - CADASTRAR
router.post("/add_usuario", usuarioController.UsuarioCreate);

//GET - LISTAR
router.get("/usuarios/:id?", usuarioController.UsuarioListar);

//PUT - UPDATE
router.put("/usuarios/:id", usuarioController.UsuarioUpdate);

// DELETE
router.delete("/usuarios/:id", usuarioController.UsuarioDelete);

module.exports = router;
```

Vamos criar as constantes dos módulos que usaremos e deixar os espaços para as codificações futuras de controllers que serão usados e requisições HTTP dos Controllers.
Finalizamos com o module.exports = router.
Vamos agora executar e verificar se o banco de dados está sendo criado e se nossa tabela usuario foi criada.



Nossa API já está rodando.

Vamos agora configurar o processo de uso do JWT.

Começamos com a instalação das bibliotecas que iremos utilizar.

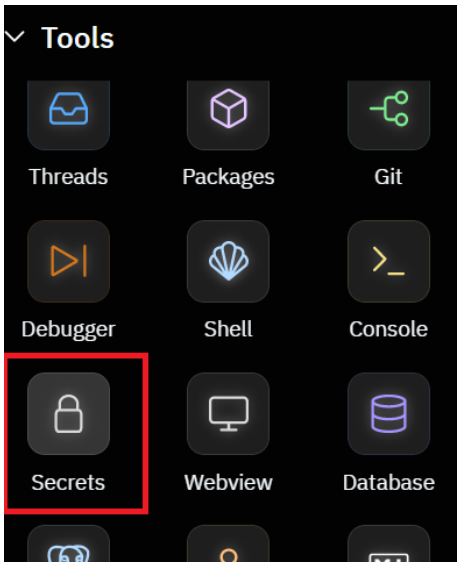

```
npm install jsonwebtoken dotenv-safe
```

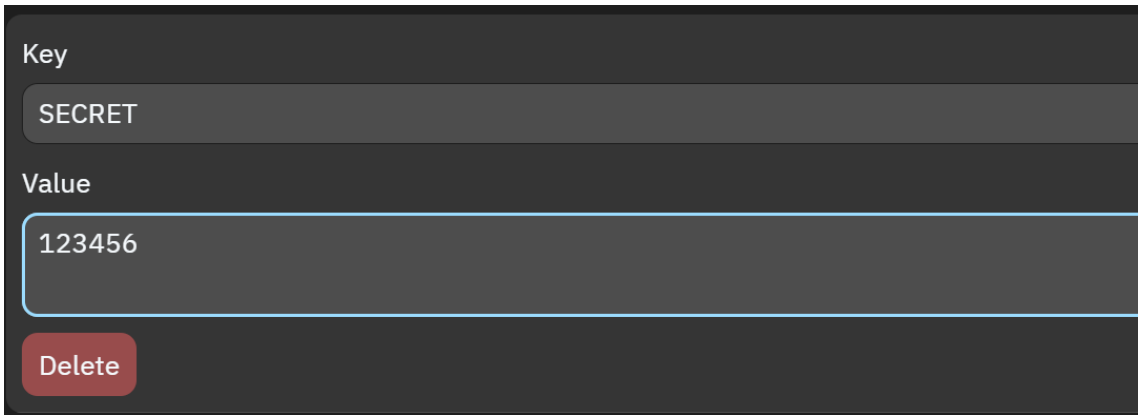
jsonwebtoken: pacote que implementa o protocolo JSON Web Token.

dotenv-safe: pacote para gerenciar facilmente variáveis de ambiente, não é obrigatório para JWT, mas uma boa prática para configurações em geral.

Agora vamos criar nossa variável de ambiente no replit para guardar nosso valor secreto do token.

Dentro de tools, temos o ícone de secrets.





Coloque o nome de SECRET e o valor de 123456 para nossa variável de ambiente.

Este segredo será utilizado pela biblioteca jsonwebtoken para assinar o token de modo que somente o servidor consiga validá-lo.

Para que esse arquivo de variáveis de ambiente seja carregado assim que a aplicação iniciar, adicione a seguinte linha logo no início do arquivo **index.js** da sua API, aproveitando para inserir também as linhas dos novos pacotes que vamos trabalhar.

```
//JWT
const jwt = require('jsonwebtoken');
//MODELS
```

Coloque também dentro do arquivo **usuarioController.js** a linha acima.

Dentro do usuarioController.js, vamos criar a nossa função para verificar se o usuário está logado.

```
static async UsuarioVerificaLogin(req, res){
  var email = req.body.email;
  var senha = req.body.senha;
  const dados = {
      email: email,
      senha: senha
    };
  const usuario = await Usuario.findOne({where: {email:email,senha:
senha}}).then((usuario) => {
    //esse teste abaixo deve ser feito no seu banco de dados
    if(usuario != undefined){
      const id = usuario.id_usuario; //esse id vira do seu banco de dados
      const token = jwt.sign({ id }, process.env.SECRET, {
```

```
        expiresIn: 300 // expira em 5min
      });
      return res.json({ auth: true, token: token }); //Criação do token
    }
    else{
      res.status(402).json({message: "Erro ao logar no sistema."});
    }
  })
}
```

A função recebe os valores do formulário de login e guarda nas variáveis (email e senha). Criamos um objeto chamado dados com os valores.
Verificamos com a função findOne se existe um usuário com o email e senha informados.
Criamos uma condição com IF para verificar se a constante usuario possui valores, se possuir guardamos o código id_usuario dentro da constante id.
O jwt cria um token e armazena na nossa chave secreta com o id do usuário com expiração para 5 minutos.
Retornamos nossa json com o token autenticado.
Vamos criar agora a função para realizar a segurança das nossas rotas. A ideia é proteger todas as rotas que só podem ser acessadas com verificação de token.

```
// VERIFICA SE O TOKEN FOI CRIADO
static async verificaJWT(req, res, next){
    const token = req.headers['x-access-token'];
    if (!token) return res.status(401).json({ auth: false, message: 'Nenhum token criado.' });
    jwt.verify(token, process.env.SECRET, function(err, decoded) {
      if (err) return res.status(500).json({ auth: false, message: 'Falha na autenticação com o token.' });

      // Salva no request para uso posterior
      req.userId = decoded.id;
      next();
    });
}
```

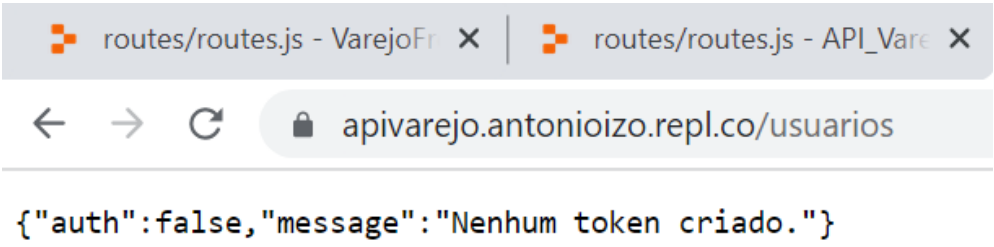
A função acima verifica se existe um token ativo na nossa conexão. Caso exista é guardado na constante token com o req.headers. O req.headers auxilia quando precisamos recuperar valores de cabeçalhos.

Primeiro verificamos se existe o token e depois verificamos se possui autorização para acesso ou expirou.
Salvamos com o req.
Utilizei o tipo x-access-token que é nativo do express para recuperar o cabeçalho caso exista.
Por fim, salvamos o id do usuário e solicitamos afunção next para passar para a próxima verificação caso exista.
Vamos aproveitar e criar no arquivo routes.js as rotas para nossas funções.
Crie a rota abaixo da última rota de usuários. (delete)

```
router.post("/login", usuarioController.UsuarioVerificaLogin);
```

Vamos incluir na nossa rota GET LISTAR de usuários a função para verificar se ela funciona.

```
//GET - LISTAR
router.get("/usuarios/:id?",usuarioController.verificaJWT,
usuarioController.UsuarioListar);
```



Nossa validação de proteção das rotas está funcionando.

Na próxima aula, vamos desenvolver nosso frontEnd.

Até a próxima aula.