Aula 5 - Segurança e JWT

Docupedia Export

Author:Ferro Alisson (CtP/ETS)
Date:21-Aug-2023 13:16

Table of Contents

1 CORS 3

2 Segurança 4

1 CORS

Cross-Origin Resource Sharing (CORS) é um mecanismo que usa cabeçalhos adicionais HTTP para informar a um navegador que permita que um aplicativo Web seja executado em uma origem (domínio) com permissão para acessar recursos selecionados de um servidor em uma origem distinta. é uma politica de segurança dos servidores, protegendo assim as requests de urls indesejadas, para isso usaremos uma biblioteca chamada cors do npm

```
npm i cors
```

E adicionarmos o código no index.js

```
const cors = require('cors');

app.use(cors({
    origin: '*'
}));
```

Onde, para nosso teste a origin está com * para responder de todos endpoints do front-end.

2 Segurança

Nas API é importante criptografar nossos dados, isso pois caso ocorra algum vazamento de dados, o usuário mantem seus dados criptografados e não expostos, para isso iremos instalar algumas bibliotecas

```
npm i dotenv bcryptjs jsonwebtoken
```

E iremos criar um arquivo de rotas para registar e autenticar em *routes/auth.js*

```
const express = require('express');
const AuthController = require('../controller/AuthController');
const router = express.Router();

router
    .post('/register', AuthController.register)
    .post('/login', AuthController.login)

module.exports = router;
```

Precisamos agora importar o arquivo de rotas no *routes/index.js*

```
const bodyParser = require('body-parser');
const person = require('./person');
const auth = require('./auth');

module.exports = (app) => {
    app.use(
        bodyParser.json(),
        person,
        auth
    )
}
```

E no arquivo *startup/routes.js* vamos definir no endpoint

```
const express = require('express');
const person = require('../routes/person');
const auth = require('../routes/auth');


module.exports = function(app) {
    app
        .use(express.json())
        .use('/api/person', person)
        .use('/api/auth', auth)
}
```

Agora nosso que terminamos as configurações iniciais, podemos assim criar o controller para autenticação.

```
const User = require('../models/User');
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');
require('dotenv').config();

class PersonController {
    static async register(req, res){

    }

    static async login(req, res){
    }
}

module.exports = PersonController;
```

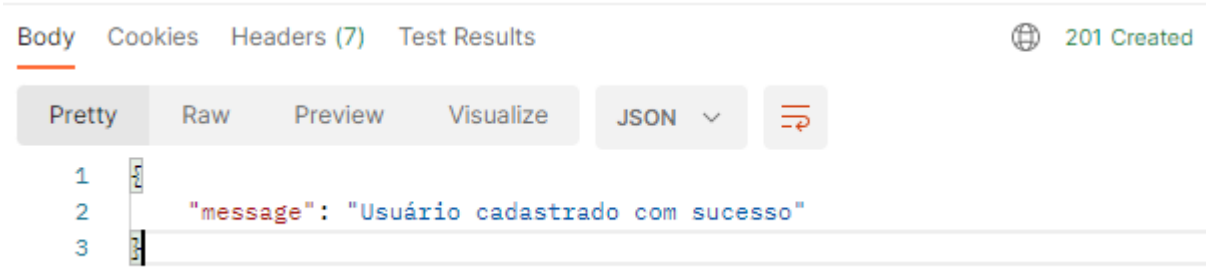
Inicialmente vamos fazer o método register e deixa o login criado somente para não dar erro de compilação.
Após feitas as verificações de nome, e-mail, senha, confirmar senha e verificar se o e-mail não esta cadastrado, vamos cadastrar no banco de dados o usuário

```
        const user = new User({
            name,
            email,
            password,
        });

        try {
            await user.save();
            res.status(201).send({ message: "Usuário cadastrado com sucesso" });
        } catch (error) {
            return res.status(500).send({ message: "Something failed" })
        }
```

Agora vamos testar no Postman e ver a resposta

```
{
    "name": "Teste",
    "email": "teste@teste.com",
    "password": "teste",
    "confirmPassword": "teste"
}
```



O usuário foi cadastrado corretamente, vamos ver as informações no banco de dados

```
_id: ObjectId('64709c949bf74aaca1854894')
name: "Teste"
email: "teste@teste.com"
password: "teste"
__v: 0
```

Temos a senha e o e-mail expostos no banco de dados, e isso é uma falha de segurança, pois qualquer pessoa que tiver acesso irá saber o login e senha, vamos modificar isso
Para isso iremos utilizar a biblioteca do bcrypt para criptografar a senha,
O primeiro passo é criar um salt, um número de caracteres aleatório que será gerado junto com a senha.
O segundo passo é gerar a senha com hash.
E por fim, onde antes passávamos a senha para salvar, vamos agora passar a senha com hash

```
        const salt = await bcrypt.genSalt(12);
        const passwordHash = await bcrypt.hash(password, salt);

        const user = new User({
            name,
            email,
            password: passwordHash
        });
```

Agora iremos testar no Postman e ver o resultado no banco de dados

```
_id: ObjectId('64709f846397a0c5bd98fef3')
name: "Teste"
email: "teste2@teste.com"
password: "$2a$12$fDHAZmiV12GaF/qOhQ62fuadFYmqMrbfNJvY3JEtOyoJAANYLcynm"
__v: 0
```

Agora sim, a senha esta criptografada.

Seguindo, vamos para o método de *login*,
Vamos fazer as verificações de usuário e senha e por fim gerar um JWT *token*.
JWT resumidamente, é uma string de caracteres que, caso cliente e servidor estejam sob HTTPS, permite que somente o servidor que conhece o 'segredo' possa validar o conteúdo do token e assim confirmar a autenticidade do cliente.
Em termos práticos, quando um usuário se autentica no sistema ou web API (com usuário e senha), o servidor gera um *token* com data de expiração pra ele.
Após as verificações iremos colocar o código

```
        try {
            const secret = process.env.SECRET
            const token = jwt.sign(
                {
                    id: user._id,
                },
                secret,
                {
                    expiresIn: '2 days'
                }

            );

            return res.status(200).send({token: token})
        } catch (error) {
            return res.status(500).send({ message: "Something failed" })
        }
```

Desafio: no projeto com os produtos, crie um endpoint para que o usuário possa se cadastrar, logar e deletar o usuário