Implementando Autenticação na API

Objetivos da Aula:

- Compreender o que é autenticação e por que ela é necessária.
- Implementar a autenticação de usuários utilizando **JSON Web Tokens (JWT).**
- Criar rotas para registro e login de usuários.
- Proteger rotas com middleware de autenticação.

O que é Autenticação?

Conceito:
Autenticação é o processo de verificar a identidade de um usuário. Ela é essencial para garantir que apenas usuários autorizados possam acessar recursos protegidos na aplicação.

Por que é Necessária?
Autenticação é necessária para:

- Proteger dados sensíveis.
- Garantir que ações críticas só possam ser executadas por usuários autorizados.
- Oferecer uma experiência personalizada ao usuário.

Passo 1: Instalando Dependências para Autenticação

Para implementar a autenticação, vamos utilizar **JSON Web Tokens (JWT)** e **bcrypt** para criptografar senhas.

Ação: Execute os seguintes comandos para instalar as dependências necessárias:

npm install jsonwebtoken bcrypt

Passo 2: Gerando a Chave Secreta JWT

Antes de configurar as rotas e controladores, **é essencial garantir que todas as variáveis de ambiente necessárias estejam corretamente definidas no arquivo .env.** Para a autenticação, isso inclui principalmente a chave secreta (JWT_SECRET), que é usada para assinar os tokens JWT.

Como Gerar a Chave Secreta Aleatória:

1. Abra um terminal no seu ambiente de desenvolvimento.
2. Execute o seguinte comando no Node.js para gerar uma chave secreta segura:

node -e "console.log(require('crypto').randomBytes(64).toString('hex'))"

Explicação:

- **require('crypto').randomBytes(64).toString('hex'):** Esse comando usa o módulo crypto do Node.js para gerar 64 bytes de dados aleatórios e converte esses bytes em uma string hexadecimal.

3. O comando acima gerará uma string aleatória, como por exemplo:

   6a2dbb2e4f5e51a3dfe93d2b8a7f7b4d8c4b2f97a30f1e9bb9e9d7f542d3b3c2

4. Copie essa **string** e adicione ao seu arquivo .**env** como o valor de **JWT_SECRET**:

JWT_SECRET=6a2dbb2e4f5e51a3dfe93d2b8a7f7b4d8c4b2f97a30f1e9bb9e9d7f542d3b3c2

Essa chave deve ser mantida em segredo e nunca deve ser compartilhada publicamente, pois é usada para garantir a integridade e autenticidade dos tokens gerados pela sua aplicação.

Passo 3: Configurando o Registro e Login de Usuários

Ação: Crie um novo arquivo **authController.js** dentro da pasta **controllers**:

touch controllers/authController.js

Código para authController.js:

```
const db = require('../config/db'); // Importa a configuração do banco de dados
const bcrypt = require('bcrypt'); // Importa o bcrypt para criptografar senhas
const jwt = require('jsonwebtoken'); // Importa o jsonwebtoken para gerar tokens JWT

// Função para registrar um novo usuário
const registerUser = async (req, res) => {
  const { name, email, password, birth_date } = req.body; // Desestrutura os dados do corpo da requisição

 // Verificar se o usuário já existe no banco de dados
  try {
    const [existingUser] = await db.promise().query('SELECT * FROM users WHERE email = ?', [email]);
    if (existingUser.length > 0) {
      return res.status(400).send('Usuário já registrado');
    }

    // Criptografar a senha usando bcrypt
    const hashedPassword = await bcrypt.hash(password, 10);

    // Inserir o novo usuário no banco de dados
    await db.promise().query(
      'INSERT INTO users (name, email, password, birth_date) VALUES (?, ?, ?, ?)',
      [name, email, hashedPassword, birth_date]
    );

    res.status(201).send('Usuário registrado com sucesso');
  } catch (err) {
    console.error('Erro ao registrar usuário:', err);
    res.status(500).send('Erro ao registrar usuário');
  }
};
```

```
// Função para autenticar um usuário
const loginUser = async (req, res) => {
  const { email, password } = req.body; // Desestrutura os dados do corpo da requisição

 // Verificar se o usuário existe no banco de dados

  try {
    const [user] = await db.promise().query('SELECT * FROM users WHERE email = ?', [email]);
    if (user.length === 0) {
      return res.status(400).send('Credenciais inválidas');
    }

    // Comparar a senha fornecida com a senha criptografada no banco de dados
    const isMatch = await bcrypt.compare(password, user[0].password);
    if (!isMatch) {
      return res.status(400).send('Credenciais inválidas');
    }

    // Gerar um token JWT
    const token = jwt.sign({ userId: user[0].id }, process.env.JWT_SECRET, { expiresIn: '1h' });

    res.json({ token });
  } catch (err) {
    console.error('Erro ao autenticar usuário:', err);
    res.status(500).send('Erro ao autenticar usuário');
  }
};

module.exports = {
  registerUser,
  loginUser
};
```

Explicação do Código:

1. **Estrutura de Funções Assíncronas (async):**
   - async (req, res) => { ... }: Declara uma função assíncrona. As funções assíncronas permitem o uso das palavras-chave await e try-catch para lidar com operações assíncronas de maneira mais legível.
2. **Bloco Try-Catch (try-catch):**
   - try { ... } catch (err) { ... }: O bloco try-catch é usado para capturar e lidar com erros que podem ocorrer durante a execução do código no bloco try. Se algum erro ocorrer, o controle passa para o bloco catch, onde podemos tratar o erro apropriadamente.
3. **Operações Assíncronas com Await (await):**
   - await db.promise().query(...);: A palavra-chave await é usada para esperar que uma Promise seja resolvida ou rejeitada antes de continuar a execução do código. No caso de operações de banco de dados ou qualquer outra operação assíncrona, isso ajuda a escrever código que parece síncrono, mas é não-bloqueante.

4. **Trabalhando com Promises (promise):**
    - db.promise().query(...);: O método promise() do objeto db transforma a operação em uma Promise. Promises são usadas para operações assíncronas e permitem encadear operações com .then() e .catch() ou usar await para um código mais limpo e legível.
5. **Criptografia de Senhas com Bcrypt (bcrypt):**
    - const hashedPassword = await bcrypt.hash(password, 10);: Criptografa a senha do usuário antes de armazená-la no banco de dados. O 10 é o número de salt rounds, que adiciona uma camada extra de segurança à senha criptografada.
    - const isMatch = await bcrypt.compare(password, user[0].password);: Compara a senha fornecida pelo usuário com a senha armazenada no banco de dados de forma segura.
6. **Geração de Tokens JWT (jwt.sign):**
    - const token = jwt.sign({ userId: user[0].id }, process.env.JWT_SECRET, { expiresIn: '1h' });: Gera um token JWT que inclui o ID do usuário no payload. O token é assinado com uma chave secreta (process.env.JWT_SECRET) e tem um tempo de expiração de 1 hora.
7. **Manipulação de Erros (catch):**
    - catch (err) { ... }: Captura qualquer erro que ocorra dentro do bloco try e permite tratar o erro de forma apropriada, como registrar o erro e enviar uma resposta de erro ao cliente.

Passo 3: Definindo Rotas para Registro e Login

Ação: Crie um novo **arquivo auth.js** dentro da pasta **routes**:

touch routes/auth.js

Código para routes/auth.js:

const express = require('express'); // Importa o framework Express
const router = express.Router(); // Cria um novo roteador
const authController = require('../controllers/authController'); // Importa o controlador de autenticação

// Rota para registro de usuário
router.post('/register', authController.registerUser);

// Rota para login de usuário
router.post('/login', authController.loginUser);

module.exports = router; // Exporta o roteador

Explicação do Código:

- Define as rotas **/register** e **/login** e associa essas rotas às funções **registerUser** e **loginUser** do controlador de autenticação.

Passo 4: Atualizando o Servidor para Incluir as Rotas de Autenticação

Ação: Atualize o arquivo **server.js** para incluir as novas rotas de autenticação:

Código para server.js:

```
const dotenv = require('dotenv'); // Importa o pacote dotenv para gerenciar variáveis de ambiente

// Carregar as Variáveis de Ambiente
dotenv.config(); // Carrega as variáveis definidas no arquivo .env para process.env

// Importar as Bibliotecas
const express = require('express'); // Importa o framework Express
const cors = require('cors'); // Importa o pacote cors para permitir requisições de diferentes origens
const bodyParser = require('body-parser'); // Importa o pacote body-parser para analisar o corpo das requisições HTTP

const db = require('./config/db'); // Importa a conexão com o banco de dados

// Inicializar nova aplicação Express
const app = express(); // Inicializa uma nova aplicação Express

// Configurar o CORS e o body-parser
app.use(cors()); // Habilita o CORS para todas as rotas
app.use(bodyParser.json()); // Configura o body-parser para analisar requisições JSON

// Importar as rotas de transações e autenticação
const transactionsRoutes = require('./routes/transactions'); // Importa as rotas de transações
const authRoutes = require('./routes/auth'); // Importa as rotas de autenticação

// Usar as rotas de transações e autenticação para as requisições
app.use('/api/transactions', transactionsRoutes); // Configura o servidor para usar as rotas de transações
app.use('/api/auth', authRoutes); // Configura o servidor para usar as rotas de autenticação

// Rota inicial para testar o servidor
app.get('/', (req, res) => {
  res.send('Servidor está rodando'); // Define uma rota inicial para testar o servidor
});

// Configurar o servidor para uma porta específica
const PORT = process.env.PORT || 3000; // Define a porta a partir da variável de ambiente ou usa a porta 3000 como padrão
app.listen(PORT, () => {
  console.log(`Servidor rodando na porta ${PORT}`); // Loga uma mensagem informando que o servidor está rodando
});
```

Passo 5: Proteger Rotas com Middleware de Autenticação

Ação: Crie um **middleware** para verificar se **o token JWT** está presente e válido.

Código para middlewares/authMiddleware.js:

```
const jwt = require('jsonwebtoken');

const authMiddleware = (req, res, next) => {
  const token = req.header('Authorization').replace('Bearer ', ''); // Obtém o token do cabeçalho da requisição

  if (!token) {
    return res.status(401).send('Acesso negado. Nenhum token fornecido.');
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET); // Verifica a validade do token
    req.user = decoded; // Adiciona as informações do usuário à requisição
    next(); // Passa o controle para a próxima função middleware
  } catch (err) {
    res.status(400).send('Token inválido.');
  }
};

module.exports = authMiddleware; // Exporta o middleware de autenticação
```

Explicação do Código:

- const token = req.header('Authorization').replace('Bearer ', '');: Extrai o token JWT do cabeçalho da requisição e remove a parte "Bearer ".
- if (!token) { ... }: Verifica se o token está presente. Se não estiver, retorna um erro 401.
- const decoded = jwt.verify(token, process.env.JWT_SECRET);: Verifica a validade do token usando a chave secreta definida nas variáveis de ambiente.
- req.user = decoded;: Adiciona as informações decodificadas do usuário à requisição para que possam ser usadas nas rotas protegidas.
- next();: Passa o controle para a próxima função middleware se o token for válido.
- catch (err) { ... }: Captura qualquer erro que ocorra durante a verificação do token e retorna um erro 400 se o token for inválido.

Ação: Atualize as rotas de transações para utilizar o middleware de autenticação.

Código para routes/transactions.js:

```
const express = require('express'); // Importa o framework Express
const router = express.Router(); // Cria um novo roteador
const transactionsController = require('../controllers/transactionsController'); // Importa o controlador de transações
const authMiddleware = require('../middlewares/authMiddleware'); // Importa o middleware de autenticação

// Definindo uma rota para obter todas as transações (protegida)
router.get('/', authMiddleware, transactionsController.getAllTransactions);

// Definindo uma rota para adicionar uma nova transação (protegida)
router.post('/', authMiddleware, transactionsController.addTransaction);
```

// Definindo uma rota para atualizar uma transação existente (substituição completa) (protegida)
router.put('/:id', authMiddleware, transactionsController.updateTransactionPut);

// Definindo uma rota para atualizar uma transação existente (atualização parcial) (protegida)
router.patch('/:id', authMiddleware, transactionsController.updateTransactionPatch);

// Definindo uma rota para deletar uma transação existente (protegida)
router.delete('/:id', authMiddleware, transactionsController.deleteTransaction);

// Exportando o roteador
module.exports = router;

Explicação do Código:

- Adiciona o middleware de autenticação **authMiddleware** a todas as rotas de transações, garantindo que apenas usuários autenticados possam acessar essas rotas.

Parte Prática (1 hora):

Ação:

1. **Registrar um novo usuário**:
   - Método: POST
   - URL: http://localhost:3000/api/auth/register
   - Corpo da requisição (JSON):

     Código JSON

     ```
     {
      "name": "Joana Dark",
      "email": "joanadark@apifinance.com",
      "password": "123456",
      "birth_date": "1990-01-01"
     }
     ```

2. **Fazer login com o usuário registrado**:
   - Método: POST
   - URL: http://localhost:3000/api/auth/login
   - Corpo da requisição (JSON):

     Código JSON

     ```
     {
      "email": "joanadark@apifinance.com",
      "password": "123456"
     }
     ```

3. **Usar o token JWT para acessar rotas protegidas**:
   - Método: GET

- URL: http://localhost:3000/api/transactions
- Cabeçalho da requisição:
  - Authorization: Bearer <token>
- Onde <token> é o token JWT recebido na resposta do login.

Verificação Importante

Certifique-se de que a variável de ambiente **JWT_SECRET** está corretamente definida no arquivo **.env.**

Essa variável é crucial para a geração e verificação dos tokens JWT. Se não estiver definida ou se o valor estiver incorreto, a autenticação não funcionará corretamente.

Dica: Caso queira gerar a chave secreta de forma automática, você pode utilizar o comando **node -e "console.log(require('crypto').randomBytes(64).toString('hex'))"** para gerar uma string hexadecimal segura e única.