

Projeto Pratico: Rotas de Cadastro e Login Seguro

Guia de Desenvolvimento de Rotas de Cadastro e Login Seguro com Bcrypt e JWT

O que iremos fazer?

Este guia explica passo a passo como implementar rotas seguras de cadastro e autenticação em um projeto **Node.js + Express** usando **bcrypt para hashing de senhas** e **JSON Web Tokens (JWT) para autenticação baseada em token**. Ele também traz dicas de segurança, exemplos de middlewares, e um exemplo de documentação OpenAPI/Swagger..

Objetivo:

Criar rotas REST seguras para:

- **Cadastro de usuário (POST /api/auth/register)** — salvar usuário com senha hasheada
- **Login / Autenticação (POST /api/auth/login)** — verificar credenciais e emitir JWT
- **Rota protegida de exemplo (GET /api/profile)** — exige token válido.

Dependências sugeridas

1. Baixe as biblioteca utilizadas no projeto:

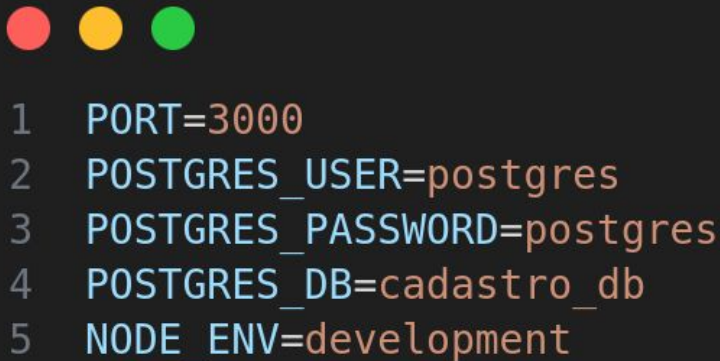


```
1 npm install express bcryptjs jsonwebtoken dotenv knex pg zod
2 # opcional para documentação
3 npm install swagger-ui-express swagger-jsdoc
```

Use **bcryptjs** por compatibilidade; bcrypt (C++) também funciona mas pode exigir build tools.

Iniciando o projeto - Variáveis de ambiente

Iniciaremos criando um `.env` que conterà as variáveis de ambiente de nosso projeto



```
1  PORT=3000
2  POSTGRES_USER=postgres
3  POSTGRES_PASSWORD=postgres
4  POSTGRES_DB=cadastro_db
5  NODE_ENV=development
```

Nunca commit esse arquivo no repositório — **adicione ao `.gitignore`.**

Estrutura mínima

```
1  src/
2    | controllers/
3    |   | auth.controller.js
4    |   | profile.controller.js
5    | db/
6    |   | migrations/
7    |   |   | user.create.js
8    |   | seeds/
9    |   |   | user.seed.js
10   |   | db.js
11   | docs/
12   |   | swagger.docs.json
13   | middlewares/
14   |   | auth.middleware.js
15   |   | validateSchema.middleware.js
16   | models/
17   |   | user.model.js
18   | routes/
19   |   | auth.routes.js
20   |   | profile.routes
21   | repository/
22   |   | user.repository.js
23   | utils/
24   |   | errorHandler.util.js
25   |   | zodSchemas.util.js
26   | app.js
27   | server.js
28  .env
29  docker-compose.yml
30  knexfile.msj
```

Instância do nosso docker

Na raiz do projeto defina nossa instância do postgresQL através do docker, crie o arquivo **docker-compose.yml**, exemplo abaixo:

```
1  services:
2    postgres-seguro:
3      container_name: postgres-seguro
4      image: postgres:17
5      restart: unless-stopped
6      ports:
7        - "5435:5432"
8      environment:
9        POSTGRES_USER: ${POSTGRES_USER}
10       POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
11       POSTGRES_DB: ${POSTGRES_DB}
12      volumes:
13        - postgres-data:/var/lib/postgresql/data
14
15  volumes:
16    postgres-data:
17      driver: local
```

Subindo o Banco de Dados

Para subir o banco, execute o comando correspondente ao seu sistema operacional no terminal:

Windows:

```
1 docker compose up -d #Flag para manter o docker  
ativo (independente do terminal)
```

Linux:

```
1 sudo docker compose up -d
```

Caso seja utilizado outras versões do docker talvez seja necessário acrescentar um "-" (hífen) entre os comandos de docker e composer, como exemplificado abaixo:

```
1 docker-compose up -d
```

Definição do nosso knexfile

Execute o inicializador do knex em seu projeto, verá que um arquivo chamado knexfile será gerado na raiz do projeto



```
1 npx knex init
```

Lembre-se que estamos utilizando **ES6**, estão como boa prática alteraremos o nome do nosso arquivo de knexfile.js para **knexfile.mjs**


Definição do nosso knexfile

Faça a configuração de conexão com o nosso banco de dados, no **knexfile.mjs** faça algo parecido com isso, lembre que estamos usando ES6 então alguns configurações de exportações deve ser alteradas, veja:

```
1 // mude module.export para export default, crie primeiro const config = {...},
2 // e o exporte no final do arquivo
3
4 import dotenv from "dotenv";
5 dotenv.config();
6
7 /**
8  * @type { Object.<string, import("knex").Knex.Config> }
9  */
10 const config = {
11   development: {
12     client: "pg",
13     connection: {
14       host: "127.0.0.1",
15       port: 5435,
16       user: process.env.POSTGRES_USER || "postgres",
17       password: process.env.POSTGRES_PASSWORD || "postgres",
18       database: process.env.POSTGRES_DB || "cadastro_db",
19     },
20     migrations: {
21       directory: "src/db/migrations",
22       extension: "js",
23     },
24     seeds: {
25       directory: "src/db/seeds",
26     },
27   },
28   ci: {
29     client: "pg",
30     connection: {
31       host: "postgres",
32       port: 5435,
33       user: process.env.POSTGRES_USER || "postgres",
34       password: process.env.POSTGRES_PASSWORD || "postgres",
35       database: process.env.POSTGRES_DB || "cadastro_db",
36     },
37     migrations: {
38       directory: "src/db/migrations",
39       extension: "js",
40     },
41     seeds: {
42       directory: "src/db/seeds",
43     },
44   },
45 };
46
47 export default config; // exportação por ES6
```


Conectando ao Banco de Dados

Posterior a isso crie um arquivo chamado **db.js**, dentro da pasta **db/**, que será responsável por fazer justamente essa conexão com nosso banco de dados, veja abaixo:



```
1 import knexConfig from "../../knexfile.mjs";
2 import knex from "knex";
3
4 const nodeEnv = process.env.NODE_ENV || "development";
5 const config = knexConfig[nodeEnv];
6
7 const db = knex(config);
8
9 export default db;
```

Execução dos migrations

Antes de iniciar nosso migrations vamos definir a estrutura da tabela de usuário, execute:

```
1 npx knex migrate:make create_users
```

Um arquivo chamado **<codigo>_create_users.js** será gerado em **db/migrations**, onde definiremos a estrutura da nossa tabela de usuários, veja que também foi necessário adaptações para que o migrations comportasse com ES6:

```
1 /**
2  * @param { import("knex").Knex } knex
3  * @returns { Promise<void> }
4  */
5 export const up = async function (knex) {
6   return await knex.schema.createTable("users", (table) => {
7     table.increments("id").primary();
8     table.string("name").nullable();
9     table.string("email").unique().nullable();
10    table.string("password").nullable();
11  });
12 };
13
14 /**
15  * @param { import("knex").Knex } knex
16  * @returns { Promise<void> }
17  */
18 export const down = async function (knex) {
19   return await knex.schema.dropTable("users");
20 };
```

Em seguida basta executar o migrations:

```
1 npx knex migrate:latest
```

Execução das seeds

Também definiremos seeds para popular nosso banco com alguns usuários iniciais, eles serão importantes para explicarmos como a inclusão das bibliotecas de criptografia atuarão sobre os novos registros, execute:



```
1 npx knex seed:make user.seed
```

Veja que um arquivo chamado **user.seed.mjs** será gerado em **src/db/seeds**, que será onde incluiremos usuários de exemplo, veja:

```
1 export const seed = async (knex) => {
2   // Deletes ALL existing entries
3   await knex("users").del();
4
5   // Inserts seed entries
6   await knex("users").insert([
7     {
8       name: "Alice Souza",
9       email: "alice@example.com",
10      password: "hashed_password_1",
11    },
12    {
13      name: "Bruno Lima",
14      email: "bruno@example.com",
15      password: "hashed_password_2",
16    },
17    {
18      name: "Carla Mendes",
19      email: "carla@example.com",
20      password: "hashed_password_3",
21    },
22  ]);
23 };
```

Automatizando Comando Padrões via package.json

Uma boa prática para projetos back-end node é armazenar/criar scripts que serão executados recorrentemente em nosso servidor para isso criaremos em nosso package.json, uma seção de scripts comuns, veja:

```
1 {
2   "name": "bcrypt_e_jwt_with_express",
3   "version": "1.0.0",
4   "description": "",
5   "main": "app.js",
6   "scripts": {
7     "dev": "node --watch src/server.js",
8     "db:cli": "sudo docker exec -it postgres-seguro psql -U postgres -d cadastro_db",
9     "db:reset": "npm run db:drop && npm run db:create && npm run db:migrate && npm run db:seed",
10    "db:drop": "sudo docker exec -it postgres-seguro psql -U postgres -c 'DROP DATABASE IF EXISTS cadastro_db;',
11    "db:create": "sudo docker exec -it postgres-seguro psql -U postgres -c 'CREATE DATABASE cadastro_db;',
12    "db:migrate": "npx knex migrate:latest ",
13    "db:seed": "npx knex seed:run"
14  },
15  ...
16 }
```

Comando Padrões via package.json

Segue uma tabela explicando cada script:

Script	Comando	Função
dev	<code>node --watch src/server.js</code>	Inicia o servidor em modo de desenvolvimento e reinicia automaticamente quando arquivos mudam.
db:cli	<code>sudo docker exec -it postgres-seguro psql -U postgres -d cadastro_db</code>	Abre o terminal interativo do PostgreSQL dentro do container <code>postgres-seguro</code> conectado ao banco <code>cadastro_db</code> .
db:reset	<code>npm run db:drop && npm run db:create && npm run db:migrate && npm run db:seed</code>	Reseta todo o banco: apaga, recria, aplica migrations e popula dados iniciais.
db:drop	<code>sudo docker exec -it postgres-seguro psql -U postgres -c 'DROP DATABASE IF EXISTS cadastro_db;'</code>	Remove o banco <code>cadastro_db</code> (se existir).
db:create	<code>sudo docker exec -it postgres-seguro psql -U postgres -c 'CREATE DATABASE cadastro_db;'</code>	Cria o banco <code>cadastro_db</code> .
db:migrate	<code>npx knex migrate:latest</code>	Executa todas as migrations pendentes para criar/alterar tabelas.
db:seed	<code>npx knex seed:run</code>	Executa os seeds para popular o banco com dados iniciais.

Desenvolvimento das Rotas de Cadastro e login

Repositories

Iniciaremos pela ordem "inversa", começaremos criando o arquivo de repositório que será responsável por acessar o banco de dados e retornar os usuários, enviar e buscar dados para realizar o login ou realizar o cadastro de um novo usuário. Pra isso crie em **repositories/** um arquivo chamado **user.repository.js**.

```
1 import db from "../db/db.js";
2
3 const userRepository = {
4   findUserByEmail: async (email) => {
5     return await db("users").where("email", email).first();
6   },
7
8   findUserById: async (id) => {
9     return await db("users").where({ id: id }).first();
10  },
11
12  insertUser: async (user) => {
13    return await db("users").insert(user).returning("*");
14  },
15
16  updateUser: async (id, user) => {
17    return await db("users").where("id", id).update(user).returning("*");
18  },
19
20  deleteUser: async (id) => {
21    return await db("users").where("id", id).del();
22  },
23 };
24
25 export default userRepository;
```

Desenvolvimento das Rotas de Cadastro e login

Controllers

Quanto aos nossos controlles iremos gerar um arquivo chama **auth.controller.js** em **src/controllers**, com a seguinte estrutura:

```
1 import userRepository from "../repositories/user.repository.js";
2 import bcrypt from "bcryptjs";
3 import jwt from "jsonwebtoken";
4 import ApiError from "../utils/errorHandler.util.js";
5
6 // Secret key for JWT
7 const SECRET = process.env.JWT_SECRET || "secret";
8
9 // Controllers
10 const login = async (req, res, next) => {
11   try {
12     const { email, password } = req.body;
13
14     const user = await userRepository.findUserByEmail(email);
15
16     if (!user) {
17       return next(
18         new ApiError("User not found", 404, {
19           email: "User not found",
20         })
21       );
22     }
23
24     const isValidPassword = await bcrypt.compare(password, user.password);
25
26     if (!isValidPassword) {
27       return next(
28         new ApiError("Invalid password", 401, {
29           password: "Invalid password",
30         })
31       );
32     }
33
34     const token = jwt.sign({ id: user.id, user: user.name, email: user.email }, SECRET, { expiresIn: "1h" });
35
36     res.status(200).json({
37       message: "User logged in successfully",
38       token: token,
39     });
40   } catch (error) {
41     next(new ApiError("Error logging in", 400, error.message));
42   }
43 };
```

Desenvolvimento das Rotas de Cadastro e login

Controllers

Quanto aos nossos controlles iremos gerar um arquivo chama **auth.controller.js** em **src/controllers**, com a seguinte estrutura:

```
1  const signUp = async (req, res, next) => {
2    try {
3      const { name, email, password } = req.body;
4
5      const user = await userRepository.findUserByEmail(email);
6
7      if (user) {
8        return next(
9          new ApiError("User already exists", 400, {
10             email: "User already exists",
11           })
12        );
13      }
14      const salt = await bcrypt.genSalt(parseInt(process.env.SALT_ROUNDS) || 10);
15      const hashedPassword = await bcrypt.hash(password, salt);
16
17      const newUser = await userRepository.insertUser({
18        name,
19        email,
20        password: hashedPassword,
21      });
22
23      res.status(201).json({
24        message: "User created successfully",
25        user: newUser,
26      });
27    } catch (error) {
28      next(new ApiError("Error creating user", 400, error.message));
29    }
30  };
31
32  export default {
33    login,
34    signUp,
35  };
```


Desenvolvimento das Rotas de Cadastro e login

Routes

Agora iremos definir o arquivo de rotas, **auth.routes.js** e o chamaremos em nosso **app.js**, segue o exemplo abaixo:

```
1 import express from "express";
2 import authController from "../controllers/auth.controller.js";
3 import { signUpSchema, loginSchema } from "../utils/zodSchemas.util.js";
4 import validateSchema from "../middlewares/validateSchemas.middleware.js";
5
6 const router = express.Router();
7
8 router.post("/register", validateSchema(signUpSchema), authController.signUp);
9 router.post("/login", validateSchema(loginSchema), authController.login);
10
11 export default router;
```

Também aproveitaremos para já deixar definida uma rota protegida que logo em seguida implementaremos tal proteção:

```
1 import express from "express";
2 const app = express();
3
4 app.use(express.json());
5 app.use(express.urlencoded({ extended: true }));
6
7 // Middleware de logging
8 app.use((req, res, next) => {
9   console.log(
10     `${new Date().toLocaleString()} | Requisição: ${req.method} ${req.url}`
11   );
12   next();
13 });
14
15 import authRoutes from "../routes/auth.routes.js";
16 import profileRoutes from "../routes/profile.routes.js";
17
18 // Rotas
19
20 // Rotas de autenticação - cadastro e login
21 app.use("/api/auth", authRoutes);
22
23 // Rota protegida - exige token válido
24 app.use("/api/profile", profileRoutes);
25
26 export default app;
```

Desenvolvimento de Rota

Proteção de Middleware

Autenticação

Para a proteção de rotas a primeira coisa que teremos que fazer é a criação de um **middleware** que será responsável por essa proteção, **validando ou não o token** passado pelo usuário

```
1 import jwt from "jsonwebtoken";
2 import ApiError from "../utils/errorHandler.util.js";
3
4 function authMiddleware(req, res, next) {
5   try {
6     // Pega o token do header
7     const tokenHeader = req.headers.authorization;
8
9     // Verifica se o token existe - se não, retorna erro
10    const token = tokenHeader && tokenHeader.split(" ")[1];
11
12    if (!token) {
13      return next(
14        new ApiError("Token not found", 401, { token: "Token not found" })
15      );
16    }
17
18    // Verifica se o token é válido - se não, retorna erro
19    jwt.verify(token, process.env.JWT_SECRET || "secret", (error, decoded) => {
20      if (error) {
21        return next(
22          new ApiError("Error authenticating user", 401, error.message)
23        );
24      }
25      // Se o token é válido, adiciona o user ao request
26      req.user = decoded;
27
28      // Continua para a rota seguinte
29      next();
30    });
31  } catch (error) {
32    return next(new ApiError("Error authenticating user", 401, error.message));
33  }
34 }
35
36 export default authMiddleware;
```

Desenvolvimento de Rota

Controlador Protegido

Agora iremos criar um controller que retornara dados do usuário se o token for válido, segue o exemplo abaixo:

```
1 import ApiError from "../utils/errorHandler.util.js";
2
3 // Controllers
4 const getProfile = async (req, res, next) => {
5   try {
6     const user = req.user;
7
8     if (!user) {
9       return next(
10         new ApiError("Users not found", 404, {
11           user: "Users not found",
12         })
13       );
14     }
15
16     res.status(200).json(user);
17   } catch (error) {
18     next(new ApiError("Error getting Profile user", 500, error.message));
19   }
20 };
21
22 export default {
23   getProfile,
24 };
```

Desenvolvimento de Rota

Rota Protegida

Portanto para implementar essa proteção, ou seja garantirmos que somente usuários logados possam acessar essa rota, basta adicionar o middleware de autenticação entre o início da rota e o final da rota, ou seja entre **/api/profile** e **getProfile**, segue o exemplo abaixo:

```
1 import express from "express";
2 import authMiddleware from "../middlewares/auth.middleware.js";
3 import profileController from "../controllers/profile.controller.js";
4
5 const router = express.Router();
6
7 router.get("/", authMiddleware, profileController.getProfile);
8
9 export default router;
```

O que fizemos?

- **Criptografia de Senha:** Usamos bcrypt para criptografia de senhas.
- **Autenticação:** Usamos JSON Web Tokens (JWT) para autenticação baseada em token.
- **Middlewares:** Usamos middlewares para tratamento de erros assíncronos.
- **Dependências:** Usar npm para gerenciamento de dependências e versões.

Checklist de segurança rápida

- Senha hasheada com salt
- Token com expiração definida
- Validação dos dados de entrada
- Armazenamento seguro das chaves (variáveis de ambiente)

Conclusão

- O JWT permite autenticar usuários de forma prática e segura, evitando o envio repetido de login e senha a cada requisição.
- Com a combinação de bcryptjs para proteção das credenciais e JWT para autenticação baseada em tokens, você estabelece uma base sólida para proteger o back-end da sua aplicação.

Links Uteis:

- [jsonwebtoken](https://www.npmjs.com/package/jsonwebtoken) — <https://www.npmjs.com/package/jsonwebtoken>
- [bcryptjs](https://www.npmjs.com/package/bcryptjs) — <https://www.npmjs.com/package/bcryptjs>

Considerações Finais

Lembre-se que isso não é tudo. Muito mais pode ser explorado.

Com bcrypt e JWT, seu sistema garante senhas seguras e autenticação prática, protegendo os dados dos usuários contra acessos não autorizados.

Acesse os código no GitHub:

<https://github.com/DaviKandido/lab-cadastro-seguro-bcrypt-jwt-express.git>

Esse tutorial foi escrito por Davi Cândido – PUC Minas. Compartilhe com colegas desenvolvedores!