

Bcrypt e JWT

**Guia de Implementação de Segurança em APIs
REST com Bcrypt e JWT (JSON Web Token)**

Introdução – Por que segurança no back-end é importante?

Dados de usuários (como senhas, e-mails e informações pessoais) são **alvo de ataques constantes**. Sem proteção, qualquer invasor que tenha acesso ao banco de dados pode visualizar todas as senhas em texto puro.

Boas práticas de segurança como **hash de senhas** e **autenticação baseada em tokens** são essenciais para proteger dados e evitar acessos não autorizados.

O que é o bcrypt e como será utilizado?

O **bcrypt** é uma biblioteca para gerar **hashes seguros de senhas**, tornando-as ilegíveis no banco de dados. Mas antes, é importante entender o conceito de hash: um processo unidirecional que transforma uma senha em uma sequência única e irreversível.

O bcrypt aplica o **salt**, que adiciona aleatoriedade ao hash, dificultando ataques como **rainbow tables**.



```
1 bcrypt.hash()      → Criar o hash antes de salvar no banco
2 bcrypt.genSalt()   → Gerar um hash mais seguro
3 bcrypt.compare()   → Verificar se a senha informada corresponde ao hash armazenado
```

O que é JWT e como será utilizado?

O **JWT (JSON Web Token)** é um padrão para **autenticação e troca segura de informações** entre cliente e servidor, sendo composto por:

- **Header** — tipo do token e algoritmo usado
- **Payload** — informações do usuário (id, e-mail, etc.)
- **Signature** — garante que o token não foi alterado

Após o login, o servidor gera um **token JWT** que será enviado pelo cliente em cada requisição, permitindo acesso a **rotas protegidas** sem precisar reenviar login e senha a cada vez.

```
1  jwt.sign()    → Gerar o token
2  jwt.verify()  → Validar o token recebido do cliente
3  jwt.decode()  → Retornar o payload
```

Guia de Desenvolvimento - Entendendo o bcrypt

1. Baixe a biblioteca bcryptjs:



```
1 npm install bcryptjs
```

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

Iniciando o projeto

Iniciaremos definindo a função responsável por fazer o **hash da senha do usuário** e em seguida a função que **verificará se a senha informada corresponde ao do hash**

```
1 const bcrypt = require("bcryptjs");
2
3 // Função para gerar o hash
4 async function hashSenha(senha) {
5   // A função do salt gera um hash mais seguro, quanto maior o salt, mais seguro.
6   // No entanto salts maiores exigem mais processamento.
7   const salt = await bcrypt.genSalt(10);
8   return bcrypt.hash(senha, salt);
9 }
```

```
1 async function verificarSenha(senha, hash) {
2   // Verifica se a senha informada pelo usuário corresponde ao hash armazenado
3   return await bcrypt.compare(senha, hash);
4 }
```

Simulação de base de dados em array

A fim de simular uma base de dados criaremos um array de usuário



```
1 //Usuários de exemplo contendo apenas login e senha
2 const users = [
3   { login: "user1@gmail.com", senha: "123" },
4   { login: "user2@gmail.com", senha: "456" },
5   { login: "user3@gmail.com", senha: "678" },
6 ];
```

Função de cadastro

Agora criaremos uma função de cadastro que simule tal operação

```
1  async function cadastroUsuario(login, senha) {
2    const user = users.find((user) => user.login === login);
3    if (user && user.login === login) {
4      throw new Error("Usuário já cadastrado");
5    }
6
7    // Adiciona o novo usuário ao array
8    users.push({ login, senha: await hashSenha(senha) });
9  }
```


Função de login

Agora criaremos uma função de login

```
1  async function loginUsuario(login, senha) {  
2    const user = users.find((user) => user.login === login);  
3    if (!user) {  
4      throw new Error("Usuário inexistente");  
5    }  
6  
7    const ok = await verificarSenha(senha, user.senha);  
8    if (!ok) {  
9      throw new Error("Senha incorreta");  
10   }  
11  
12   return "Login efetuado com sucesso";  
13 }
```

Fluxo do usuário

E por fim, uma função que tem como finalidade simular fluxos comuns de um usuário, realizando operações de cadastro e login, informando um cadastro anteriormente realizado e um teste de login incorreto

```
1  async function fluxoUsuario() {
2    try {
3      // Cadastro
4      await cadastroUsuario("user4@gmail.com", "123456");
5
6      // Login correto
7      const loginCerto = await loginUsuario("user4@gmail.com", "123456");
8      console.log("Retorno: ", loginCerto);
9
10     // Exemplo de login com usuário inexistente (descomente para testar)
11     //const loginErrado = await loginUsuario("userInexistente@gmail.com", "876543");
12
13     // Listar usuários cadastrados
14     users.forEach((user, i) => console.log(`user${i}: `, user));
15
16   } catch (err) {
17     console.log("Erro:", err.message);
18   }
19 }
20
21 fluxoUsuario();
```

Conclusão sobre o Bcrypt

- Nunca armazene senhas em texto puro no banco de dados.
- Sempre utilize hashing com salt para aumentar a segurança.
- O bcrypt é amplamente usado no mercado e considerado seguro para a maioria das aplicações.

Links Uteis:

- [bcryptjs — https://www.npmjs.com/package/bcryptjs](https://www.npmjs.com/package/bcryptjs)
- [OWASP Authentication Cheat Sheet — https://cheatsheetseries.owasp.org/](https://cheatsheetseries.owasp.org/)

Guia de Desenvolvimento - Integrando com JWT para autenticação

1. Baixe a biblioteca do bcryptjs e do JWT (jsonwebtoken):



```
1 npm install bcryptjs
```



```
1 npm install jsonwebtoken
```

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

Iniciando o projeto


Antes de iniciarmos é preciso entender que esse projeto é uma continuação direta do guia "Guia de Desenvolvimento — Entendendo o bcrypt", caso não o tenha visto antes volte e veja os seus conceitos antes de continua.

```
1  const jwt = require("jsonwebtoken");
2  const SECRET = "minhaChaveSecreta"; // Armazene isso em variáveis de ambiente
3
4  async function loginUsuario(login, senha) {
5    const user = users.find((user) => user.login === login);
6    if (!user) {
7      throw new Error("Usuário inexistente");
8    }
9
10   const ok = await verificarSenha(senha, user.senha);
11   if (!ok) {
12     throw new Error("Senha incorreta");
13   }
14
15   // Gerar token com ID ou login do usuário
16   const token = jwt.sign({ login: user.login }, SECRET, { expiresIn: "1h" });
17
18   return { message: "Login efetuado com sucesso", token: token };
19 }
```

A primeira modificação que iremos fazer é **incluir a geração do token** de acesso após o usuário realizar um login com sucesso, ou seja nossa função de login agora também retornará um token, veja:

Token de Autenticação

Veja que agora um login realizado com sucesso retornará algo parecido com isso:



```
1 {  
2   "message": "Login efetuado com sucesso",  
3   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJsb2dpbiI6InVzZXI0QGdtYWlsLmNvbSI6ImIhdCI6MTc1NDcwMjk5NiwiZXhwIjoxNzU0NzA2NTk2fQ.9J2ZH2WZ6ZKE7R8yDzlwvSKH_NmZNCy8ZX80QmKW9gY"  
4 }
```

Validação do Token Gerado

Agora criaremos uma função que servirá para **validar o token gerado**, veja que utilizamos **jwt.verify()** para realizar essa validação, e observe que ao realizarmos um decode - **jwt.decode(token)**, podemos retornar o payload ou seja, o conteúdo utilizado para realizar a geração do token.

```
1 function validarJwt(token) {  
2   const verificar = jwt.verify(token, SECRET);  
3  
4   // O jwt.decode nada mais faz do que decodificar o token devolvendo o payload  
5   // const payload = jwt.decode(token);  
6   // console.log("Payload: ", payload.login);  
7  
8   if (!verificar) {  
9     throw new Error("Token inválido");  
10  }  
11  return true;  
12 }
```

Fluxo do usuário com validação de token

E por fim, acrescentemos a lógica de validação do token gerado a função responsável pela simulação do fluxo de um usuário

```
1  async function fluxoUsuario() {
2    try {
3      await cadastroUsuario("user4@gmail.com", "123456");
4
5      const loginCerto = await loginUsuario("user4@gmail.com", "123456");
6      console.log("Retorno: ", loginCerto);
7
8      // Validando o token de acesso
9      const token = loginCerto.token;
10     const ok = validarJwt(token);
11     console.log("Token Valido: ", ok);
12
13     // Exemplo de login com usuário inexistente
14     //const loginErrado = await loginUsuario("userInexistente@gmail.com", "876543");
15
16     //users.forEach((user, i) => console.log(`user${i}: `, user));
17   } catch (err) {
18     console.log("Erro:", err.message);
19   }
20 }
21 fluxoUsuario();
```


Conclusão sobre o JWT

- 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)
- [OWASP Authentication Cheat Sheet — https://cheatsheetseries.owasp.org/](https://cheatsheetseries.owasp.org/)

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!