

# Linguagem de Programação Visual

RICARDO HENDGES



# Autenticação X Autorização

**Autenticação:** É o processo de verificar a identidade de um usuário. Isso geralmente envolve um processo de login, no qual o usuário fornece suas credenciais (como nome de usuário e senha) para a aplicação, que, em seguida, verifica se essas credenciais estão corretas.

**Autorização:** É o processo de verificar se um usuário tem acesso a certas funcionalidades ou recursos da aplicação. Por exemplo, um usuário pode ser autenticado, mas só ter acesso a certas páginas da aplicação se ele tiver uma determinada permissão.

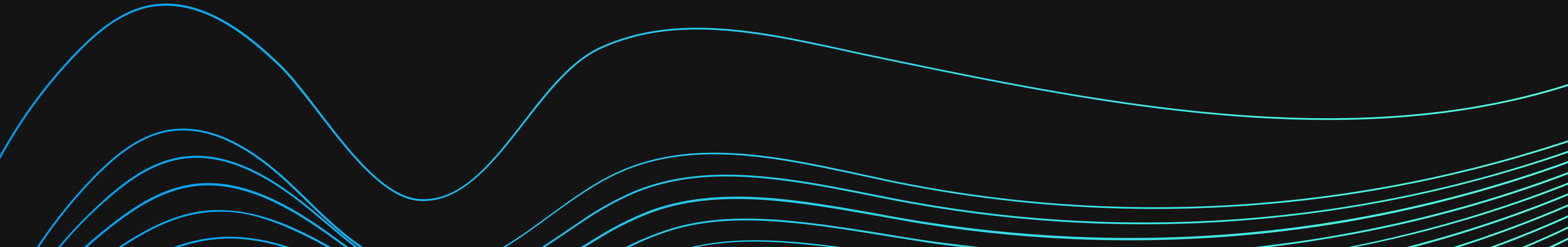
# Implementando

NA IMPLEMENTAÇÃO DA AUTENTICAÇÃO E AUTORIZAÇÃO,  
ALGUMAS COISAS A SEREM CONSIDERADAS INCLUEM:

1. **Armazenamento de credenciais de usuário:** As credenciais de usuário devem ser armazenadas de maneira segura, usando técnicas como criptografia ou hashing.
2. **Registro de usuários:** A aplicação deve permitir que novos usuários se registrem, forneçam suas credenciais e sejam autenticados.
3. **Login:** A aplicação deve permitir que os usuários se loguem, forneçam suas credenciais e sejam autenticados.
4. **Gerenciamento de sessões:** A aplicação deve gerenciar sessões de usuários, para que eles não precisem fazer login todas as vezes que acessarem a aplicação.
5. **Verificação de permissões:** A aplicação deve verificar as permissões de um usuário antes de permitir que ele acesse certos recursos ou funcionalidades.

# Armazenar credenciais de usuário

## TÉCNICAS COMUNS PARA ARMAZENAR CREDENCIAIS DE USUÁRIO

1. **Hashing de senhas:** Em vez de armazenar as senhas de usuário em texto simples, é comum usar técnicas de hashing para criptografar as senhas. Quando o usuário fornece suas credenciais para a aplicação, a senha é convertida em uma "hash" e comparada com a hash armazenada na base de dados. Pacotes como bcrypt ou scrypt podem ser usados para fazer o hashing das senhas em Node.js.
  2. **Salt:** Adicionar um "salt" a uma senha antes de fazer o hashing aumenta a segurança, já que torna mais difícil para um atacante que tenha acesso ao hash da senha descobrir a senha original. Os pacotes de hashing de senhas, como bcrypt, já incluem suporte a salting.
  3. **Criptografia:** Além do hashing, é possível criptografar as senhas usando técnicas de criptografia simétrica ou assimétrica. No entanto, essa abordagem é mais complexa e geralmente não é necessária se o hashing for feito corretamente.
- 

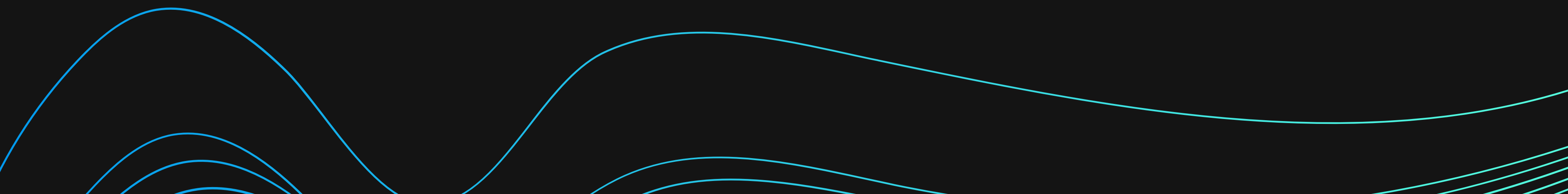
# Armazenar credenciais de usuário

## SALT!!!

Salt é uma string aleatória adicionada a uma senha antes de ser hasheada (transformada em uma string única, irreversível). O objetivo é tornar a senha mais segura e evitar ataques de dicionário, onde os invasores tentam descobrir a senha original comparando as hashes com as palavras em um dicionário.

Quando um usuário cria uma conta, a senha é concatenada com o salt antes de ser hasheada. O salt e a hash da senha são armazenados juntos no banco de dados. Quando o usuário fornece sua senha para fazer login, a senha fornecida é concatenada com o salt e, em seguida, comparada com a hash armazenada. Se as hashes forem iguais, o usuário é autenticado.

Usar um salt diferente para cada senha armazenada torna a tarefa de descobrir as senhas mais difícil, mesmo se o invasor tiver acesso ao banco de dados com as hashes. Além disso, se uma única hash for comprometida, o invasor não terá acesso às outras senhas, já que cada uma delas terá um salt diferente.





# Armazenando Dados Usuário

DB - ESTRUTURA . . .

Tabela users com quatro colunas:

- **id** é uma chave primária **serial**, ou seja, é um campo que será gerado automaticamente e aumentará a cada nova inserção na tabela;
- **username** para armazenar o nome de usuário. Também adicionamos uma restrição **UNIQUE** à coluna username para garantir que não haja dois usuários com o mesmo nome de usuário.
- **salt** é uma coluna do tipo texto que armazenará o salt gerado para cada senha;
- **password** é uma coluna do tipo texto que armazenará a senha criptografada com salt.

```
CREATE TABLE users (  
  id SERIAL PRIMARY KEY,  
  username TEXT NOT NULL UNIQUE,  
  salt TEXT NOT NULL,  
  password TEXT NOT NULL  
);
```

Dessa forma, ao salvar um novo usuário no banco de dados, você precisaria salvar o nome de usuário, gerar o salt, concatená-lo com a senha, criptografá-los juntos, e salvar o nome de usuário, salt e a senha criptografada em suas respectivas colunas na tabela. Ao autenticar um usuário, você precisaria recuperar o salt associado ao nome de usuário fornecido, concatená-lo com a senha digitada, criptografá-los juntos, e comparar o resultado com a senha criptografada armazenada no banco de dados.

# Gerando SALT

## BIBLIOTECA CRYPTO DO NODE.JS

Este código gera uma string aleatória de 16 bytes codificados em hexadecimal. É importante que você gere um salt único para cada senha armazenada.

```
1  const crypto = require('crypto');
2
3  function generateSalt() {
4    return crypto.randomBytes(16).toString('hex');
5  }
6
7  const salt = generateSalt();
8  console.log(salt);
9
10
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

**ricardojh@sk-nb-0111464:**~/estudos/horus/LP23/bacl

Debugger listening on ws://127.0.0.1:33803/20d48f

For help, see: <https://nodejs.org/en/docs/inspect>

Debugger attached.

a5b999bd7b86399ea30a86b9f264762e

Crypto esta incorporado no nodeJS,  
nao há necessidade de instala-lo.

# Concatenando SALT + Senha

Você pode concatenar o salt com a senha fornecida e hasheá-los juntos usando a biblioteca crypto do Node.js. Aqui está um exemplo que usa o algoritmo PBKDF2 para gerar a hash:

```
1  const crypto = require('crypto')
2
3  function generateSalt() {
4    return crypto.randomBytes(16).toString('hex')
5  }
6
7  function hashPassword(password, salt) {
8    return crypto.pbkdf2Sync(password, salt, 1000, 64, 'sha512').toString('hex')
9  }
10
11  const password = 'password'
12  const salt = generateSalt()
13
14  const hashedPassword = hashPassword(password, salt)
15  console.log(salt)
16  console.log(hashedPassword)
17
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
ricardojh@sk-nb-0111464:~/estudos/horus/LP23/back-23/src/utis$ node salt.js
Debugger listening on ws://127.0.0.1:40667/1f075a1a-b76d-4e6e-b6e9-b8cc61049
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
608865e1a8feb38d158072fda5fd2403
27c250efdd185d00a389bf5fa7fdaa6f3ec7d1e53041bdc0d033ce42c3a5734c939b06535337
```

Neste exemplo, a função hashPassword concatena o salt com a senha fornecida e usa o algoritmo PBKDF2 para gerar a hash. A hash resultante é retornada como uma string hexadecimal. É importante que você armazene a hash resultante juntamente com o salt correspondente em seu banco de dados.



# Registro de usuários

```
function criarUsuario (usuario, senha) {  
  const salt = generateSalt()  
  const hashedPassword = hashPassword(senha, salt)  
  // chama banco para salvar {usuario, salt e hashedPassword}  
}
```

Neste exemplo, a função criarUsuario recebe o usuário e senha digitados, gera um salt novo para esse usuário, após isso concatena o salt com sua senha para gerar o hash.

Próximo passo seria a chamada de banco para guardar essas 3 informações no banco de dados para depois poder executar o login utilizando esses dados.

# Login

```
const user = 'RICARDO.HENDGES'
const password = '123456'

const storedPassword = '9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08'
const salt = '2ef97a57a7a0c1c7e26a9ba9bb80f5e8'

function comparePassword(storedPassword, salt, providedPassword) {
  const hash = hashPassword(providedPassword, salt)
  return hash === storedPassword
}

console.log(comparePassword(storedPassword, salt, password))
```

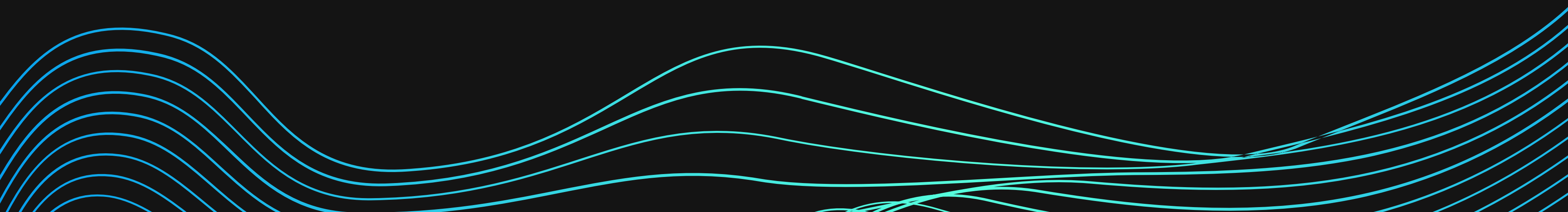
# JWT

O JWT (JSON Web Token) é um padrão de autenticação de informações que permite passar informações autenticadas entre partes como uma forma de autenticação. JWTs são compostos de três partes: cabeçalho, payload e assinatura.

1. **Cabeçalho**: O cabeçalho contém informações sobre o tipo de token (JWT) e o algoritmo de criptografia usado na geração da assinatura.
2. **Payload**: O payload contém as informações que desejamos transmitir, tais como identificadores de usuários, datas de expiração, etc.
3. **Assinatura**: A assinatura é gerada a partir do cabeçalho e do payload, usando uma chave secreta e um algoritmo de criptografia.

O formato geral de um JWT é o seguinte: **Header.Payload.Signature**.

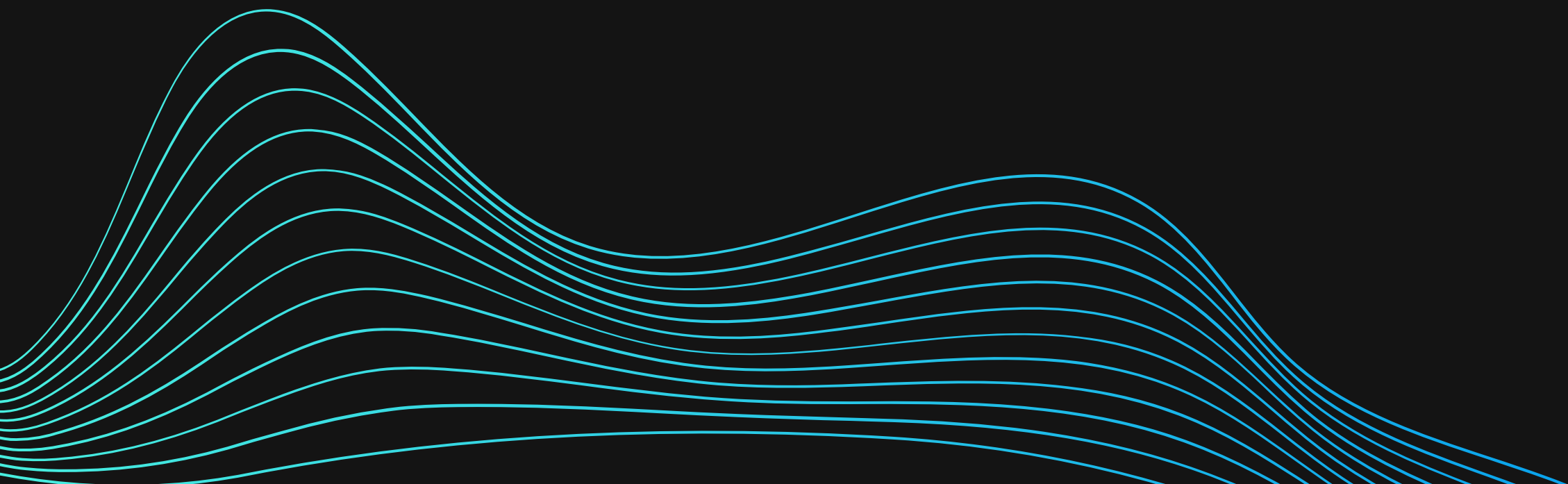
Quanto à criptografia, existem vários algoritmos de criptografia que podem ser usados para gerar a assinatura JWT, incluindo HMAC SHA-256 e RSA. O algoritmo a ser usado é especificado no cabeçalho do JWT. É importante que o algoritmo de criptografia escolhido seja seguro e tenha uma implementação confiável, pois ele é usado para proteger as informações transmitidas no JWT.



# JWT - Assinaturas

Existem três tipos de assinaturas JWT:

1. **HMAC** (Hash-based Message Authentication Code): Este tipo de assinatura usa uma chave secreta compartilhada para gerar a assinatura. É usado em aplicações em que o emissor e o destinatário confiam uns nos outros e podem compartilhar uma chave secreta.
2. **RSA**: Este tipo de assinatura usa uma chave pública e uma chave privada para gerar e verificar a assinatura. O emissor usa a chave privada para assinar o JWT e o destinatário usa a chave pública para verificar a assinatura. É usado em aplicações em que o emissor e o destinatário não confiam uns nos outros, mas confiam na infraestrutura de chave pública.
3. **ECDSA** (Elliptic Curve Digital Signature Algorithm): Este tipo de assinatura é similar ao RSA, mas usa curvas elípticas em vez de primos para gerar as chaves. É mais seguro e mais rápido do que o RSA, mas requer mais recursos computacionais.



# JWT - RSA

Para criar uma chave privada e uma chave pública RSA, você pode seguir os seguintes passos:

1. Instale uma ferramenta de geração de chaves RSA, como o OpenSSL.
2. Abra o terminal e execute o seguinte comando para gerar a chave privada RSA:

```
openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt rsa_keygen_bits:2048
```

3. Execute o seguinte comando para gerar a chave pública RSA a partir da chave privada:

```
openssl rsa -pubout -in private_key.pem -out public_key.pem
```





# JWT - jsonwebtoken - sign

- Instale uma biblioteca para trabalhar com JWT, como o jsonwebtoken para Node.js.
- No seu servidor, crie um objeto com as informações que você deseja incluir no JWT.
- Assine o JWT usando sua chave privada RSA:

```
1  const jwt = require("jsonwebtoken")
2  const fs = require("fs")
3
4  const payload = { userId: 123, permissions: ["read", "write"] }
5
6  const privateKey = fs.readFileSync("../private/private_key.pem")
7
8  const token = jwt.sign(payload, privateKey, { algorithm: "RS256" })
9  console.log(token)
10
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
ricardojh@sk-nb-0111464:~/estudos/horus/LP23/back-23/src/utils$ node
Debugger listening on ws://127.0.0.1:33115/406abe3c-d4b8-4a9e-af89-
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiJlcyMywiczGVybWlzc2
GVsathDikGl66hhhHuCX8Lx2UAWmd2vc5i_WEzRL1KJj0czzBxyCURzpAbIf9Hxibep
KiQvyBb6isytIc62YbescNsRF6bLcvd9BHlsqFh-TwuA298mFXuywXanj17tAcRg5As
ObBztp5Rkwp0wVGvTlxDHpsf05mb20MdJFKasjA
```

# JWT - jsonwebtoken - verify

- Envie o JWT para o cliente (por exemplo, armazene-o em um cookie ou inclua-o em uma resposta HTTP).
- No lado do cliente, verifique a assinatura do JWT usando a chave pública RSA:

Se a assinatura do JWT for válida, você poderá acessar as informações codificadas no payload. Se a assinatura não for válida, o erro "Verification failed" será exibido.

Observe que esses são apenas exemplos gerais e que a implementação exata varia de acordo com a plataforma e biblioteca escolhida.

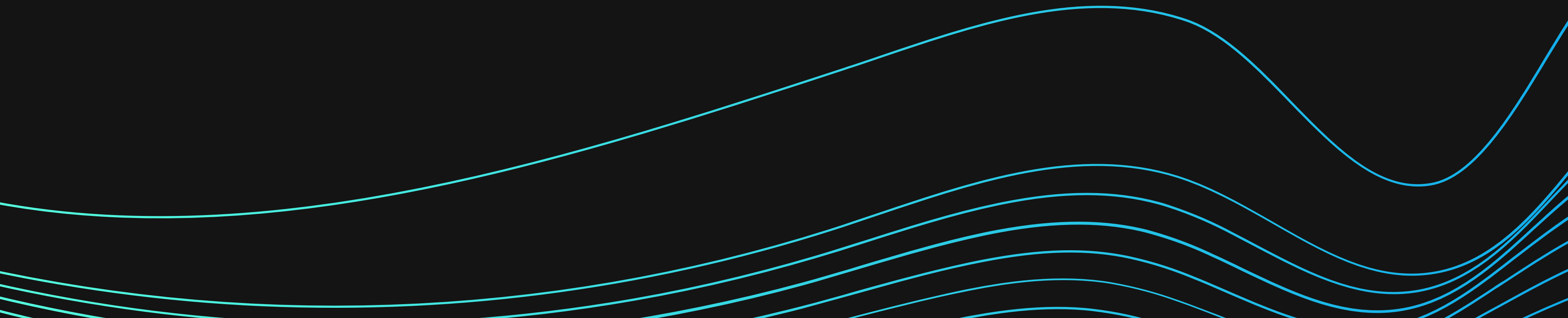
```
1  const jwt = require("jsonwebtoken")
2  const fs = require("fs")
3
4  const publicKey = fs.readFileSync("../private/public_key.pem")
5
6  token = 'eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiJyMywiczGVybWlzc2lvbnR5cy51c2Vybm9udG8iLCJ1aWkiOiJ1c2Vybm9udG8iLCJhdWQiOiJ1c2Vybm9udG8iLCJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9'
7
8  try {
9    const decoded = jwt.verify(token, publicKey, { algorithms: ["RS256"] });
10    console.log("Decoded:", decoded)
11  } catch (err) {
12    console.error("Verification failed:", err)
13  }
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
```

```
ricardojh@sk-nb-0111464:~/estudos/horus/LP23/back-23/src/utills$ node jwt.js
Debugger listening on ws://127.0.0.1:38671/5169d1ce-dace-4f47-b5e8-74d15b84a
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
Decoded: { userId: 123, permissions: [ 'read', 'write' ], iat: 1675715129 }
```

# Cookie

Cookies são pequenos arquivos de texto armazenados em seu navegador de internet pelos sites que você visita. Eles servem para armazenar informações sobre sua navegação, como preferências de usuário, informações de login e histórico de navegação. Isso permite que os sites forneçam uma experiência mais personalizada e facilitem o uso de recursos, como carrinhos de compras e login automático. Além disso, os cookies também são utilizados para fins de análise e publicidade online, ajudando os anunciantes a entender suas preferências e exibir anúncios relevantes.



# Cookie - res.cookie

res.cookie é uma função disponível no Express.js, que permite enviar cookies para o cliente. É uma função adicionada pelo middleware "cookie-parser". Aqui está um exemplo de como usar res.cookie para enviar um cookie para o cliente:

```
const express = require('express');
const cookieParser = require('cookie-parser');

const app = express();

app.use(cookieParser());

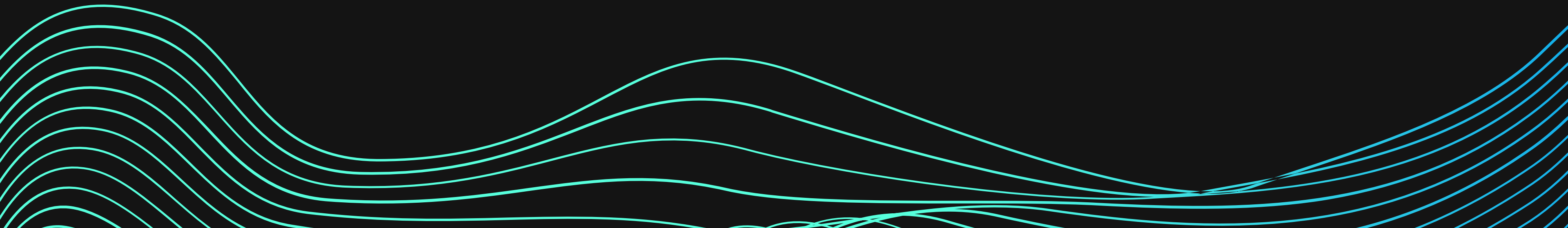
app.get('/', (req, res) => {
  res.cookie('name', 'value');
  res.send('Cookie set!');
});

app.listen(3000);
```

# Cookie - res.cookie

Também estamos especificando as seguintes opções para tornar o cookie mais seguro:

- **secure**: true para indicar que o cookie deve ser enviado apenas em conexões seguras (HTTPS).
- **httpOnly**: true para impedir o acesso ao cookie por scripts client-side, tornando mais difícil para os atacantes acessarem ou modificarem o cookie.
- **sameSite**: 'strict' para impedir que o cookie seja enviado com solicitações cross-site, protegendo contra ataques de roubos de sessão.
- **maxAge**:  $1000 * 60 * 60 * 24$  para definir a validade do cookie como 24 horas.





# routes

usuarios.js

```
const usuariosController = require('../controllers/usuarios')

module.exports = (app) => {
  app.post('/user', usuariosController.newUser)
  app.get('/user', usuariosController.getUser)
  app.delete('/user/:id', usuariosController.deleteUser)
}
```

## index.js

```
const Alunos = require('./alunos')
const Usuarios = require('./usuarios')

module.exports = (app) => {
  Alunos(app)
  Usuarios(app)
}
```

# controllers

usuarios.js

```
const usuariosService = require('../services/usuarios')

const newUser = async (req, res, next) => {
  try {
    const retorno = await usuariosService.newUser(req.body)
    res.status(201).json(retorno)
  } catch (err) {
    res.status(500).send(err.message)
  }
}

const getUser = async (req, res, next) => {
  try {
    const retorno = await usuariosService.getUser()
    res.status(201).json(retorno)
  } catch (err) {
    res.status(500).send(err.message)
  }
}

const deleteUser = async (req, res, next) => {
  try {
    const retorno = await usuariosService.deleteUser(req.params)
    res.status(201).json(retorno)
  } catch (err) {
    res.status(500).send(err.message)
  }
}

module.exports.newUser = newUser
module.exports.getUser = getUser
module.exports.deleteUser = deleteUser
```

# services

usuarios.js

## Valide compatibilidade com

- init.sql - docker
- arquivo salt

```
const db = require('../config/db')
const cript = require('../utils/salt')

const sql_insert = `
  insert into usuarios (usu_name, usu_salt, usu_password)
  values ($1, $2, $3) `

const newUser = async (params) => {
  const {user, pass} = params
  const {salt, hashedPassword} = cript.criarSalt(pass)
  result = await db.query(sql_insert, [user, salt, hashedPassword])
  return result
}

const sql_get = ` select usu_id, usu_name from usuarios `

const getUser = async () => {
  result = await db.query(sql_get, [])
  return {
    total: result.rows.length,
    usuarios: result.rows
  }
}

const sql_delete = ` delete from usuarios where usu_id = $1 `

const deleteUser = async (params) => {
  return await db.query(sql_delete, [params.id])
}

module.exports.newUser = newUser
module.exports.getUser = getUser
module.exports.deleteUser = deleteUser
```

# TESTE!!!

1. insira usuários novos!

a. **POST /user**

2. consulte-os!

a. **GET /user**

3. remova caso necessário!

a. **DELETE /user**

Tema de casa ...

Implemente um patch para atualizar a senha do usuário!

PS.: Desde que ele tenha a senha original . . . .

# Controllers

login.js

```
const loginService = require('../services/login')

const login = async (req, res, next) => {
  if (req.headers && req.headers.authorization && req.headers.authorization.indexOf('Basic') > -1) {
    const basicToken = req.headers.authorization
    token = decodeURIComponent(Buffer.from(basicToken.substr(basicToken.indexOf('Basic') + 6), 'base64'))
    let posPonto = token.indexOf(':')
    req.body.user = token.substr(0, posPonto)
    req.body.pass = token.substr(posPonto+1)

    loginService.login(req.body)
      .then(ret => {
        res.cookie('auth', ret.token, {
          sameSite: 'none',
          secure: true,
          expires: new Date(Date.now() + 1000 * 60 * 60 * 24 * 7)})
        res.status(201).json({status: ret.status, usuario: ret.user})
      })
      .catch(err => res.status(err.status? err.status : 500).json({type:err.type ,message: err.message, detail: err.detail}))
  } else {
    res.status(400).json({type: 'ERRO', message: 'LOGIN WITH BASIC AUTH!'})
  }
}

module.exports.login = login
```



# Service

## login.js

```
const db = require('../config/db')
const jwt = require('jsonwebtoken')
const cript = require('../utils/salt')

const sql_get = `
  select usuarios.usu_name,
         usuarios.usu_salt,
         usuarios.usu_password
  from usuarios
  where usu_name = $1 `

const login = async (params) => {
  const {user, pass} = params
  result = await db.query(sql_get, [user])
  if (!result.rows.length) throw new Error("USUÁRIO NÃO EXISTE!")
  else {
    const salt = result.rows[0].usu_salt
    const password = result.rows[0].usu_password
    if (cript.comparePassword(password, salt, pass)) {
      let perfilAcesso = result.rows[0].usu_name
      let token = jwt.sign({perfilAcesso}, process.env.PRIVATE_AUTH, {algorithm: 'RS256', expiresIn: '7d' })
      return {
        status: 'Logado com sucesso!',
        user: result.rows[0].usu_name,
        token: token
      }
    } else {
      throw { status: 400, type: 'WARN', message: `Senha inválida!`, detail: ''}
    }
  }
}

module.exports.login = login
```