

PERN

Pablo Leon Rodrigues



UPF

PERN

O PERN stack é uma coleção de quatro tecnologias usadas para construir aplicativos web, PostgreSQL, Express, React e Nodejs.

- **PostgreSQL**: um sistema de gerenciamento de banco de dados relacional de objeto (ORDBMS) de código aberto que oferece suporte a SQL para dados relacionais e JSON para dados não relacionais.
- **Express.js**: uma estrutura da web JavaScript popular desenvolvida para Node.js que simplifica a construção de aplicativos da web e APIs.
- **React.js**: uma biblioteca JavaScript para construir interfaces de usuário dinâmicas.
- **Node.js**: um ambiente de tempo de execução JavaScript que permite executar JavaScript no lado do servidor.

P E R N

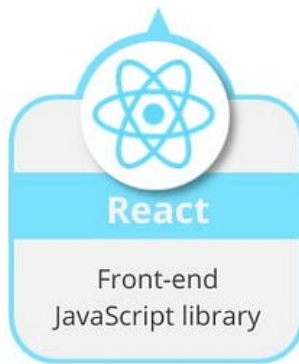
S

t

a

c

k



Prós

- JavaScript full-stack: usar JavaScript tanto para front-end quanto para back-end pode simplificar o desenvolvimento para aqueles que já estão familiarizados com a linguagem.
- Comunidade: as grandes comunidades por trás de cada tecnologia da pilha fornecem amplos recursos de aprendizagem e suporte.
- Escalabilidade: O stack PERN pode lidar com aplicações pequenas e grandes devido à escalabilidade inerente de seus componentes.
- Flexibilidade: Os recursos de manipulação de dados do PostgreSQL oferecem flexibilidade no armazenamento e gerenciamento de dados.

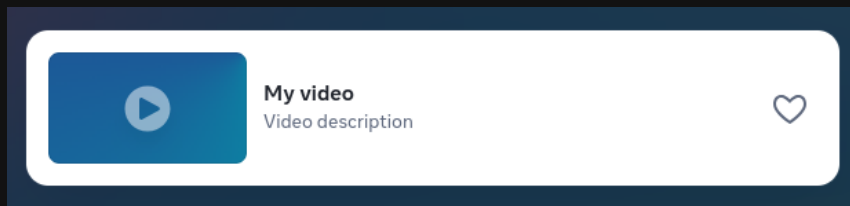
Cons

- Complexidade: embora o uso de JavaScript em toda a pilha possa ser vantajoso, ele também pode adicionar complexidade para desenvolvedores que não estão familiarizados com JavaScript no back-end.
- Escolha de banco de dados: o PostgreSQL pode ser um exagero para aplicativos mais simples que não exigem seus recursos avançados.
- Considerações de segurança: como acontece com qualquer desenvolvimento web, a segurança precisa ser cuidadosamente considerada. Porém, o mesmo pode ser dito para qualquer outro stack.

REACT

REACT é uma biblioteca Javascript utilizada para criar interfaces de usuário. Mais utilizado para single-page apps, foi desenvolvido pelo Facebook.

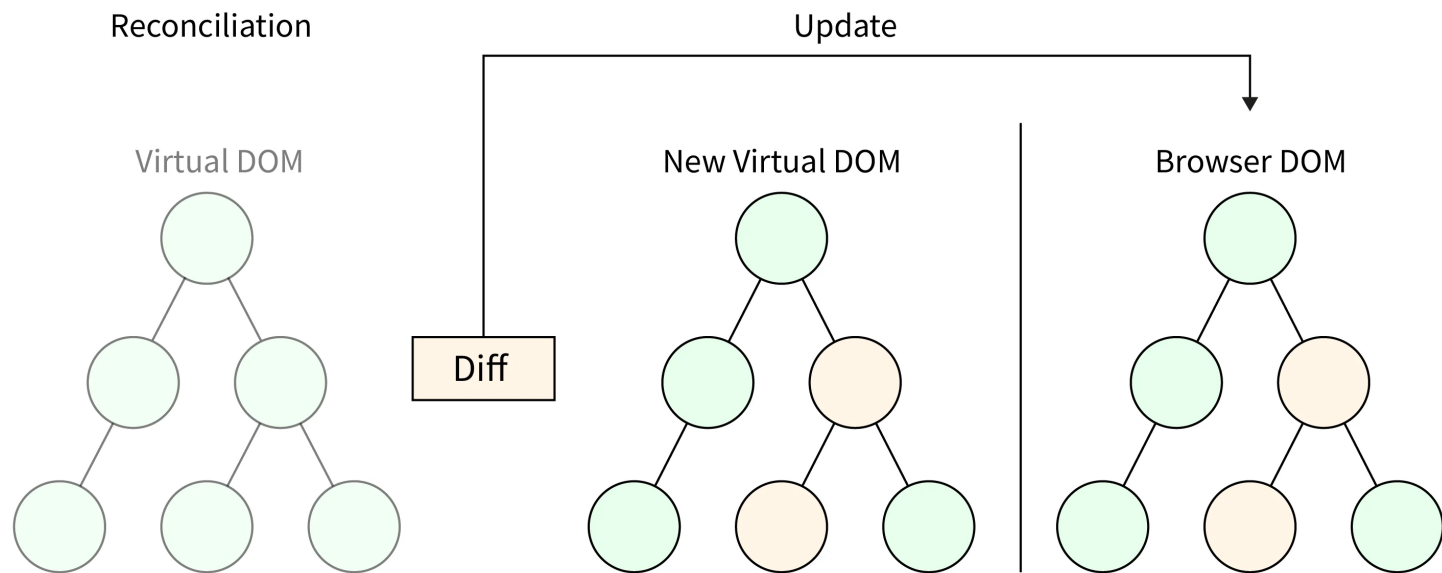
O objetivo do react é simplificar o processo de desenvolver interfaces de usuário utilizando uma abordagem baseada em componentes.



JSX, uma extensão de sintaxe para JavaScript, permite aos desenvolvedores escrever componentes de UI em um formato semelhante a XML ou HTML. Tornando o código mais legível e expressivo.

```
function Video({ video }) {  
  return (  
    <div>  
      <Thumbnail video={video} />  
      <a href={video.url}>  
        <h3>{video.title}</h3>  
        <p>{video.description}</p>  
      </a>  
      <LikeButton video={video} />  
    </div>  
  );  
}
```

React usa um DOM virtual (Document Object Model). Em vez de atualizar todo o DOM quando os dados mudam, o React primeiro cria uma representação virtual do DOM na memória. Em seguida, ele calcula a maneira mais eficiente de atualizar o DOM real, reduzindo a necessidade de recarregamentos.



Next.js

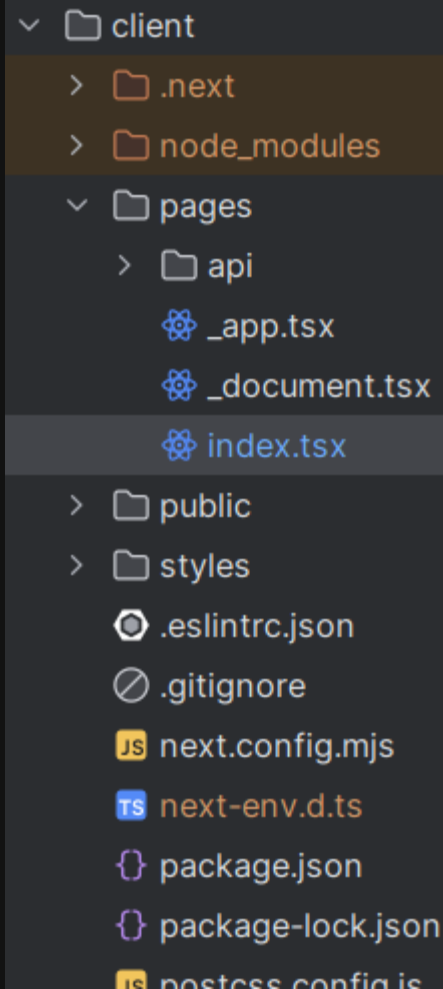
O react recomenda a utilização de frameworks para construção de app, vamos utilizar o [Next.js](#), desenvolvido pela [Vercel](#).

O Nextjs, utiliza [Tailwind](#) css como padrão para folhas de estilo.

Para instalar o Nextjs:

```
npm install -g nextjs
```

```
leon@leon-dev ~/repos/nodejs npx create-next-app client
✓ Would you like to use TypeScript? ... No / Yes
✓ Would you like to use ESLint? ... No / Yes
✓ Would you like to use Tailwind CSS? ... No / Yes
✓ Would you like to use `src/` directory? ... No / Yes
✓ Would you like to use App Router? (recommended) ... No / Yes
? Would you like to customize the default import alias (@/*)? > No / Yes
```



```

v client
  > .next
  > node_modules
  v pages
    > api
      _app.tsx
      _document.tsx
    index.tsx
  > public
  > styles
.eslintrc.json
.gitignore
JS next.config.mjs
TS next-env.d.ts
package.json
package-lock.json
JS postcss.config.js
```

Componentes

Um componente é um bloco de código reutilizável e independente, que divide a interface do usuário em partes menores.



Componentes de classe

Os componentes de classes têm acesso a recursos adicionais, como o ciclo de vida do componente. Isso permite que os desenvolvedores controlem o comportamento do componente em diferentes estágios, `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.

Eles também têm suporte nativo ao gerenciamento de estado usando o objeto `state`. Isso permite que os desenvolvedores armazenem e atualizem o estado interno do componente de forma fácil e intuitiva.

Esses componentes precisavam de um método `render()` para poder retornar o JSX.

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}</h1>;  
  }  
}
```

Componentes funcionais

Um componente funcional, basicamente, é uma função em JavaScript/ES6 que retorna um elemento do React (JSX).

Os componentes funcionais oferecem uma sintaxe mais concisa, facilidade de reutilização e melhor desempenho, sendo a escolha preferida para novos projetos e aqueles que buscam uma abordagem mais moderna de desenvolvimento. Eles possibilitam gerenciar o lifecycle através de `hooks`.

- é uma função em JavaScript/ES6
- deve retornar um elemento em React (JSX)
- sempre começa com letra maiúscula (convenção dos nomes)
- aceita props como parâmetro, se necessário

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}  
export default Welcome;
```

```
import Welcome from './Welcome';  
  
function App() {  
  return  
    <div className="App">  
      <Welcome name="John"/>  
    </div>  
}
```

`props` são a forma de comunicação entre componentes react. As props transportam dados apenas do elemento pai para os elementos filhos.

useState hook

Você chama `useState` dentro do seu componente funcional para declarar uma variável `state`. Essa variável deve ser criada no escopo do componente, ao tentar criar um `state` dentro de uma função, bloco condicional ou loop irá causar um erro.

```
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0)
  return (<div>{count}</div>);
}
```

- A chamada `{useState(0)}` declara a variável de estado `count` com um valor inicial de 0.
- A função `setCount` é usada para atualizar a variável.
- `useState` pode ser usado para gerenciar vários tipos de dados, incluindo números, strings, booleanos, arrays ou objetos.
- As atualizações de estado sempre devem ser feitas usando a função `setter` retornada por `useState`.
- Nunca modifique diretamente a variável de estado.
- Os componentes do React são renderizados novamente sempre que seu estado muda. Isso garante que a IU reflita os valores de estado mais recentes.

useEffect hook

Hooks são funções que possibilitam gerenciar efeitos em componentes funcionais.

- Buscar data de API's
- Manipular DOM
- Criar temporizadores

```
useEffect(function, dependencies)
```

```
useEffect(() => {  
  
}, []);
```

- O primeiro argumento `function` é uma função que contém a lógica do efeito colateral.
- O segundo argumento `dependencies` (opcional) é uma matriz de dependências. Isso informa ao React quando executar novamente o efeito, se nenhuma dependência for especificada, o efeito será executado após cada renderização.

Ao executar, por padrão o `effect` vai rodar depois do render inicial e depois de cada update subsequente. Esse padrão pode ser alterado usando a um `array` de dependências. Se o array estiver vazio essa função só vai executar uma vez, se tiver um valor(es), toda vez que esse valor for alterado a função vai ser executada.

```
const [count, setCount] = useState(0)
return(
  <div>
    <p>{count}</p>
    <button onClick={() => setCount(count + 1)}>mais</button>
    <button onClick={() => setCount(count - 1)}>menos</button>
  </div>
)
```

```
const [hello, setHello] = useState("Hello")
const [bool, setBool] = useState(true)
return(
  <div>
    <p>{hello}</p>
    <button onClick={() => setHello("goodbye")}>bye</button>
    {bool ? <p>verdadeiro</p> : <p>false</p>}
    <button onClick={() => bool ? setBool(false) : setBool(true)}>toogle</button>
  </div>
)
```


Fetch API

```
useEffect(() => {  
  fetch("https://dog.ceo/api/breeds/image/random")  
    .then((resp) => resp.json())  
    .then((apiData) => {  
      setData(apiData.message);  
    });  
}, [update]);
```

Axios

Axios é um cliente HTTP baseado em promessa para `node.js` e navegador. É isomórfico (pode rodar no navegador e nodejs com a mesma base de código). No lado do servidor utiliza o módulo http nativo node.js, enquanto no cliente (navegador) utiliza XMLHttpRequests. `npm install axios`

```
import axios from "axios";

const api = axios.create({
  baseURL: "https://api.github.com",
});

export default api;
```

```
axios({
  method: 'post',
  url: '/login',
  data: {
    firstName: 'Finn',
    lastName: 'Williams'
  }
});
```

- `axios.request(config)`
- `axios.get(url[, config])`
- `axios.delete(url[, config])`
- `axios.head(url[, config])`
- `axios.options(url[, config])`
- `axios.post(url[, data[, config]])`
- `axios.put(url[, data[, config]])`
- `axios.patch(url[, data[, config]])`

Axios Post

Depois que uma solicitação HTTP POST é feita, o Axios retorna uma promessa que é cumprida ou rejeitada, dependendo da resposta do serviço de backend

```
axios.post('/login', {
  firstName: 'Finn',
  lastName: 'Williams'
})
.then((response) => {
  console.log(response);
}, (error) => {
  console.log(error);
});
```

Se a promessa for cumprida, o primeiro argumento de `then()` será chamado; se a promessa for rejeitada, o segundo argumento será chamado.

```
{
  // `data` é a resposta(*response*)
  data: {},
  // `status` é o HTTP status code
  status: 200,
  // `statusText` é a mensagem de status
  statusText: 'OK',
  // `headers` headers da resposta
  headers: {},
  // `config` configuração para o axios
  config: {},
  // `request` é o request dessa response
  request: {}
}
```

SPA

Uma das características do REACT é a possibilidade de criar SPA(Single Page Applications)

Usando o pages router, cada página criada dentro da pasta pages, vai redirecionar para uma url de acordo com o nome do arquivo. Porém essa navegação envolve o load normal das páginas.

Por exemplo a página `listarPessoas.tsx` vai refletir com a url

`localhost:3000/listarPessoas` e a página default `index.tsx` é associada a url `localhost:3000/`.

Router

Diversas libs podem ser usadas para gerenciar as rotas de páginas, uma delas é o `react-router-dom`.

Para instalar

```
npm install react-router-dom
```

Para configurar o router no projeto next precisamos alterar a estrutura da aplicação.

_app.tsx

```
import "@styles/globals.css"
import {AppProps} from 'next/app'
import {useEffect, useState} from 'react'

function App({Component, pageProps}: AppProps) {
  const [render, setRender] = useState(false)
  useEffect(() => setRender(true), [])
  return render ? <Component {...pageProps} /> : null
}
export default App
```

index.tsx

```
import {BrowserRouter as Router, Routes, Route} from 'react-router-dom';
import ListarPessoas from "@pages/listarPessoas";
import "bootstrap/dist/css/bootstrap.min.css";

export default function Home() {
  return (
    <Router>
      <main className="container">
        <Routes>
          <Route path="/" element={<h1>Home</h1>}/>
          <Route path="/listarPessoas" element={<ListarPessoas/>}/>
        </Routes>
      </main>
    </Router>
  );
}
```

Autenticação

Os dois principais métodos de autenticação na web são através de `sessions` e `tokens`. Cada um tem suas características pontos fracos e fortes, como sempre a utilização desses métodos varia conforme com o projeto e o escopo.

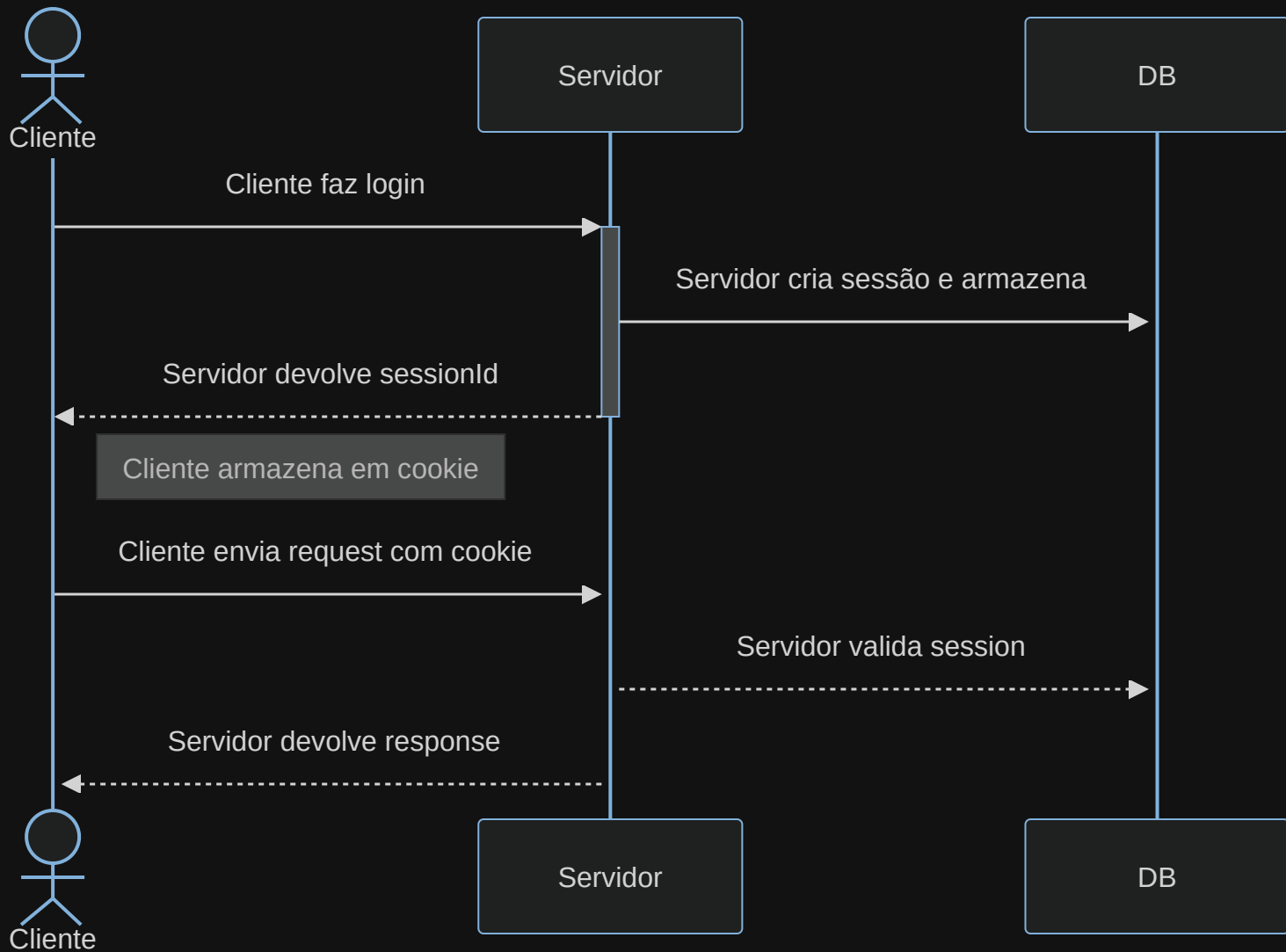
Ou seja `depende` ...

Sessions

O método de `sessions` é o tradicional na web sendo utilizado em diversos tipos de aplicativos...

Ele consiste em:

- Usuário faz o login
- O server cria uma sessão, essa sessão é armazenada em memória no server ou em um banco de dados
- O server devolve o response do login e com o
- O cliente armazena essa sessão em um cookie no browser
- O cliente faz uma requisição enviando juntamente o cookie
- O servidor busca essa sessão para checar se ela é válida
- Se estiver tudo certo o servidor devolve o response



Essa abordagem foi muito utilizada e existe debate entre os prós e cons de utilizar sessões. Dentre os cons podemos destacar os dois principais:

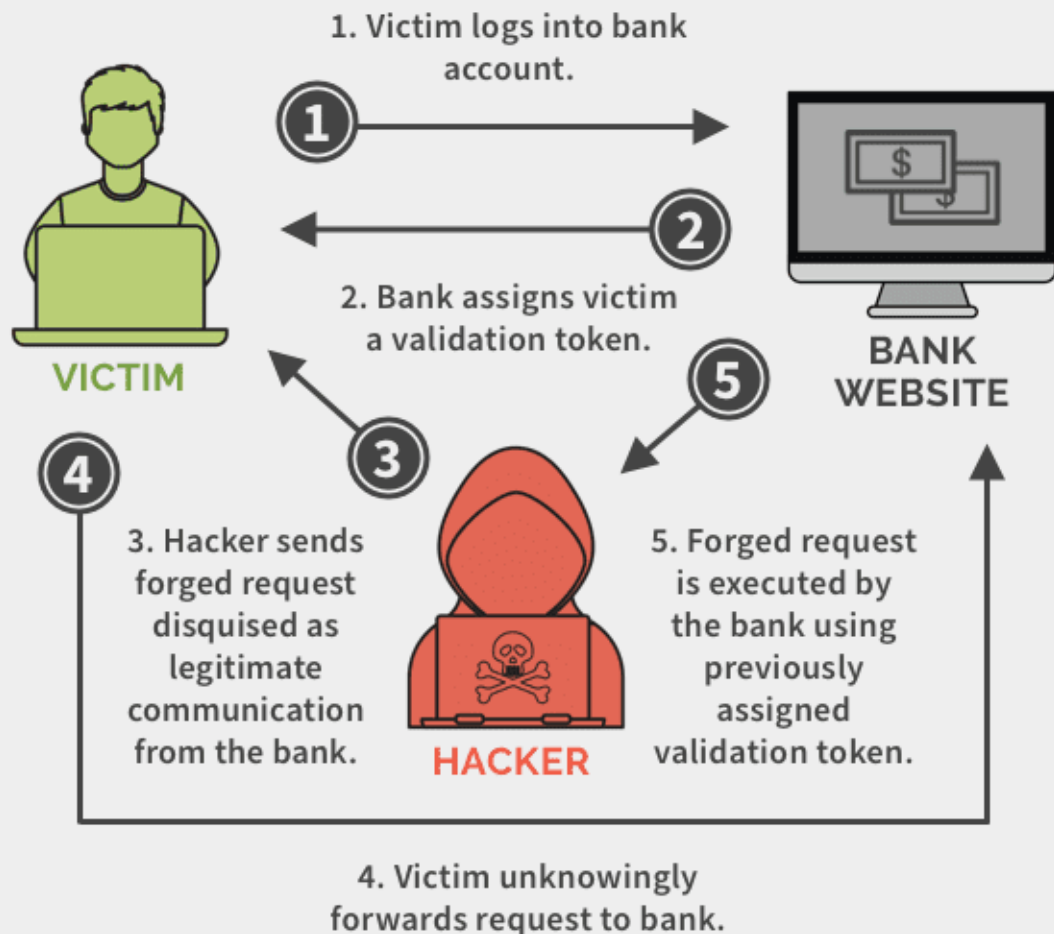
Segurança:

O principal ponto de falha de segurança nessa abordagem são os ataques de `cross site request forgery` , [CSRF](#)

Esse tipo de ataque consiste em utilizar uma sessão atualizada em um cookie no navegador da vítima, e enviar um request malicioso que usa essa sessão armazenada para fazer alguma ação no servidor verdadeiro.

Se a aplicação for desenvolvida com recursos modernos e frameworks para validação, esse risco diminuí, também é necessária uma boa engenharia social por parte do atacante...

Cross-Site Request Forgery



Performance issues

O maior problema atual na utilização de sessões é a sessão ser armazenada ou em memória no servidor, ou em um banco de dados, não parece um problema grande né..., mas pense da seguinte forma...

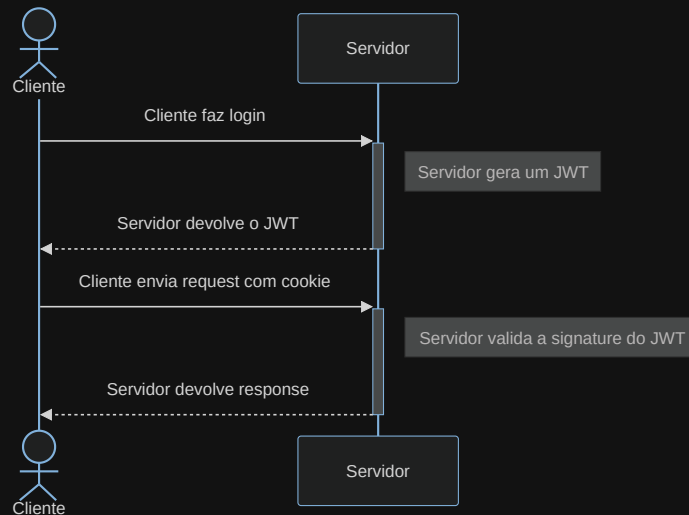
1. cada usuário que faz login cria uma sessão.
2. a sessão é armazenada
3. quando um usuário faz qualquer request o server deve:
 - buscar no banco de dados ou em memória essa sessão e validar se é válida
4. agora imagine que temos 10 instâncias desse servidor(API) em execução, uma API escalada horizontalmente em um cloud server
5. e imagine que temos 1 milhão de usuários logados em cada instância e cada usuário vai fazer em média 5 requisições

JWT

Uma alternativa para o uso de sessions é a utilização de tokens, o mais utilizado hoje é o JWT **Json web token**.

O JWT é um padrão de autenticação definido pela **RFC7519**. No JWT é utilizado um token Base64 que pode ser usado com par de chaves ou assinatura(public/private).

Usando JWT o servidor não precisa armazenar nada, ele gera o JWT e devolve para o cliente.



Um JWT é uma string com três partes separadas por um `.`, as três partes são `header`, `payload`, `signature`.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjEiLCJlbWFpbCI6InRlc3RlQHRlc3RlLmNvbSJ9.SnpFarLPRcuEFZ-bnUC-2PLhEAYzgdSYrS4oNcr6v5Q

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.

header

eyJpZCI6IjEiLCJlbWFpbCI6InRlc3RlQHRlc3RlLmNvbSJ9.

payload

SnpFarLPRcuEFZ-bnUC-2PLhEAYzgdSYrS4oNcr6v5Q

signature

header

Headers é o cabeçalho do token onde passamos basicamente duas informações: o `alg` que informa qual algoritmo é usado para criar a assinatura e o `typ` que indica qual o tipo de token.

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

payload

É onde os dados são armazenados. Pode conter informações como o identificador do usuário, permissões, expiração do token, etc. O JWT é assinado digitalmente, mas isso não é o mesmo que criptografia, não é aconselhável utilizar dados sensíveis em um JWT.

```
{  
  "id": "1",  
  "email": "teste@teste.com"  
}
```

signature

A assinatura do token (signature) é composta pela codificação do header e do payload somada a uma chave secreta gerada pelo algoritmo especificado no header.

```
HS256SHA256(  
    base64UrlEncode(header) + "." + base64UrlEncode(payload), secret_key)
```

Outros atributos que são comuns no `payload` são:

- `sub` : usado para representar o `subject` ou id do usuário
- `iat` : usado para definir o `inserted at` do token
- `exp` : usado para definir o `expire at` do token



Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjEiLCJlbWFpbCI6InRlc3R1QHRlc3R1LmNvbSJ9.SnPFarLPRcuEFZ-bnUC-2PLhEAyzgdSYrS4oNcr6v5Q
```

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "id": "1",
  "email": "teste@teste.com"
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

 Signature Verified

SHARE JWT


```
export const doLogin = async (req: Request, res: Response) => {
  const { username, password } = req.body
  if (!username || !password) {
    return res.status(400).json({ error: 'Username e password são obrigatórios' })
  }
  const usuarioRepository = AppDataSource.getRepository(Usuario)
  try {
    const usuario = await usuarioRepository.findOneBy({ username: username })
    if (!usuario) {
      return res.status(401).json({ error: 'Usuário não encontrado' })
    }
    const isPasswordValid = await bcrypt.compare(password, usuario.password)
    if (!isPasswordValid) {
      return res.status(401).json({ error: 'Credenciais inválidas' })
    }
    return res.status(200).json({ message: 'Login com sucesso' })
  } catch (error) {
    console.error('Erro durante login:', error)
    return res.status(500).json({ error: 'Internal server error' })
  }
}
```

Vamos adaptar a api para utilizar jwt. Primeiro precisamos adicionar o jsonwebtoken ao projeto.

```
npm install jsonwebtoken  
npm i --save-dev @types/jsonwebtoken
```

Agora nosso processo de login ao verificar um usuário no banco, devemos gerar um token para enviar no response, esse token vai ser utilizado para autenticar outras requisições.

No método de login vamos importar a lib do jsonwebtoken

```
import jwt from "jsonwebtoken"
```

Vamos mudar o trecho após a validação do login adicionando a criação do token e o retorno do mesmo no response

```
const token = jwt.sign({  
  username: username  
}, process.env.TOKEN, {  
  expiresIn: '1h'  
})  
  
res.status(200).json({  
  auth: true,  
  token: token }).send()
```

Agora com o token, precisamos de um método para fazer a autenticação desse token.

```
import {Request, Response} from "express";
import jwt from "jsonwebtoken";
class Authentication {
  async hasAuthorization(req: Request, res: Response, next: () => void) {
    const bearerHeader = req.headers.authorization
    if (!bearerHeader) {
      res.status(403).json({auth: false, message: 'Nenhum token fornecido.'})
    }
    const bearer = bearerHeader.split(' ')[1]
    jwt.verify(bearer, process.env.TOKEN, function (err, decoded) {
      if (err) return res.status(500).json({
        auth: false,
        message: 'Failed to authenticate token.'
      });
      req.params.token = bearer;
      next();
    });
  }
}
export default new Authentication()
```

Depois precisamos adicionar a função que criamos para autenticação nas rotas que queremos "proteger".

```
routerUsuario.get("/usuarios/listar", Auth.hasAuthorization ,getUsuarios)
```

CRUD - Login / get

GET http://localhost:3001/usuarios/listar

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Headers Hide auto-generated headers

Key	Value
<input checked="" type="checkbox"/> Postman-Token	<calculated when request is sent>
<input checked="" type="checkbox"/> Host	<calculated when request is sent>
<input checked="" type="checkbox"/> User-Agent	PostmanRuntime/7.37.3
<input checked="" type="checkbox"/> Accept	*/*
<input checked="" type="checkbox"/> Accept-Encoding	gzip, deflate, br
<input checked="" type="checkbox"/> Connection	keep-alive
<input checked="" type="checkbox"/> Authorization	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybWFTZSI6InRlc3RlliwiaWF0IjoxNzE1MTEzODc0LCJleHAiOiE3MT...
Key	Value

Body Cookies Headers (8) Test Results

Testando a API

Para implementarmos testes na API vamos utilizar duas bibliotecas, [Jest](#) e [Supertest](#)

- "Jest é um framework de teste de JavaScript encantador com foco na simplicidade." Jest website
- SuperTest é uma biblioteca de asserções HTTP que permite testar seus servidores HTTP Node.js.

```
npm install --save-dev jest
npm install --save-dev supertest ts-jest
npm i --save-dev @types/jest
npm i --save-dev @types/supertest
```

No `package.json`, vamos adicionar o script de teste. E vamos criar uma pasta para armazenar todos os testes, `__test__`.

```
{
  "scripts": {
    "test": "jest"
  },
  "jest": {
    "testEnvironment": "node",
    "coveragePathIgnorePatterns":
      ["/node_modules/"]
  }
}
```

Criamos um arquivo na raiz do projeto chamado `jest.config.ts` que vai conter configurações do jest.

```
module.exports = {
  preset: 'ts-jest',
  testEnvironment: 'node',
  testMatch: ['<rootDir>/**/*.test.ts'],
  verbose: true,
  forceExit: true,
  clearMocks: true,
}
```

Essa configuração determina que o jest vai utilizar o ts-jest para typescript, que o ambiente de teste é um container node, e que vai buscar por todos os arquivos de teste dentro das pastas do projeto que sigam a nomenclatura `*.test.ts`.

O `verbose` é utilizado para "logar" mais informações sobre os testes em execução para facilitar a interpretação do resultado.

O `clearMocks` opção é uma configuração que limpa automaticamente os dados simulados entre cada teste. Garantindo que seus testes sejam executados isoladamente e que as afirmações sobre o comportamento simulado sejam precisas.

O jest cria blocos de teste utilizando o `describe` dentro dele existem quantos casos de teste forem necessários. A nomenclatura para nome de arquivos de teste normalmente segue o nome da classe que está sendo testada, seguido de `.test`, por exemplo dentro da pasta `__test__` temos a arquivo `login.test.ts`.

```
const request = require('supertest');
const server = require('./server');

describe('GET /usuarios/listar', () => {
  it('deve retornar uma lista de usuários', async () => {
    const response = await request(server).get('/usuarios/listar');
    expect(response.statusCode).toBe(200);
    expect(response.body).toBeInstanceOf(Array);
  });
});
```

Adicionando mais segurança

Além de testes e da validação utilizando jwt tokens, existem outras medidas de segurança que podemos adotar para diminuir riscos.

Restringindo requests

Dependendo do escopo da api podemos restringir a origem das chamadas garantindo que apenas um determinado domínio tenha acesso a api.

Para isso vamos utilizar a lib `CORS` e definir os domínios permitidos, requests externos vão ser bloqueados.

```
const cors = require('cors');
const app = express();
const allow = ['https://localhost'];
const corsOptions = {
  origin: function (origin, callback) {
    if (allow.indexOf(origin) !== -1
        || !origin) {
      callback(null, true);
    } else {
      callback(new Error());
    }
  }
};
app.use(cors(corsOptions));
```


Headers

Para melhorar a proteção contra Cross-Site Scripting entre outros, podemos utilizar o middleware

`helmet`.

```
npm install helmet
```

Depois utilizar como middleware no app

```
import helmet from "helmet"

const app = express()

app.use(helmet())
```

O helmet seta diversos headers na api para mitigar alguns tipos de ataque.

Permissões

Uma medida que podemos tomar é definir permissões específicas para a api conforme o acesso do usuário.

Por exemplo: apenas usuários com permissão de admin podem fazer determinada chamada para a api. Nesse caso verificamos as permissões do usuário que fez o request. Isso pode e é normalmente feito no frontend mas também podemos adicionar a validação na api.

Aqui podemos causar um problema de lentidão ao verificar diversas vezes a permissão do usuário no banco, esse tipo de medida pode causar uma lentidão conforme o tamanho do sistema.

rate limit

Alguns ataques consistem em derrubar a aplicação utilizando força bruta(ddos), nesses casos podemos delimitar a quantidade de requisições por IP. Uma biblioteca que possibilita isso é a `express-rate-limit`.

```
npm install express-rate-limit
```

```
import rateLimit from 'express-rate-limit'

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000,
  // 15 minutes
  max: 100,
  // Limit to 100 requests by ID
  standardHeaders: true,
  // Return rate limit info in the
  // `RateLimit-*` headers
  legacyHeaders: false,
  // Disable the `X-RateLimit-*`
  // headers
})

app.use(limiter)
```

Sanitization

Uma das mais recomendadas boas práticas em apis é a [sanitização](#) de parâmetros. Existem muitas bibliotecas que ajudam nessa tarefa, aqui vamos usar o `zod`.

```
npm install zod
```

Com o zod criamos "schemas" (`schema`) onde passamos o body ou parâmetros que queremos validar para um objeto validador.

```
const userSchema = z.object({  
  name: z.string().min(1).required(),  
  email: z.string().email().optional(),  
  password: z.string().min(8).max(100),  
  limit: z.number().int().positive(),  
});
```

```
import { z } from 'zod'  
  
const loginSchema = z.object({  
  username: z.string().min(1).max(20),  
  password: z.string().min(5).max(12),  
})  
  
const validate = (schema, body) => {  
  try {  
    schema.parse(body);  
    return {  
      isValid: true,  
      errors: null  
    }  
  } catch (e) {  
    return {  
      isValid: false,  
      errors: e.errors  
    }  
  }  
}
```

Desta forma podemos chamar a função validate conforme o schema utilizado

```
const { isValid, errors } = validate( userSchema, req.body )

if (!isValid) {
  return res.status(400).json(
    { errors }
  )
}
```

<https://www.dio.me/articles/react-componentes-de-classes-x-componentes-funcionais>

<https://www.freecodecamp.org/portuguese/news/componentes-funcionais-props-e-jsx-em-react-tutorial-de-react-js-para-iniciantes/>

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

https://axios-http.com/docs/api_intro

<https://www.scaler.com/topics/react/virtual-dom-in-react/>

<https://dev.to/vinibgoulart/how-to-protected-a-route-with-jwt-token-in-react-using-context-api-l38>

<https://www.atatus.com/blog/cross-site-request-forgery-a-threat-to-open-web-applications/>

<https://owasp.org/www-community/attacks/csrf>

<https://jwt.io/>

<https://www.alura.com.br/artigos/o-que-e-json-web-tokens>

<https://dev.to/gimnathperera/yup-vs-zod-vs-joi-a-comprehensive-comparison-of-javascript-validation-libraries-4mhi>

<https://www.alura.com.br/artigos/validacao-de-formulario-com-javascript>