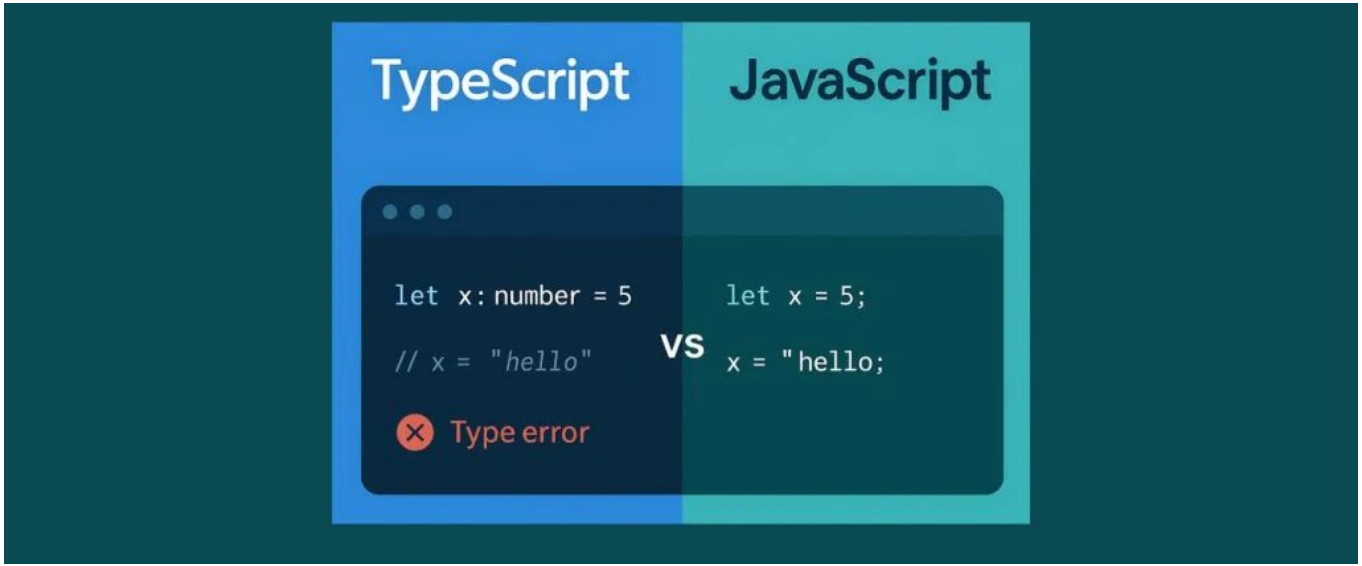
# TypeScript FullStack

Prof. Douglas Andrade

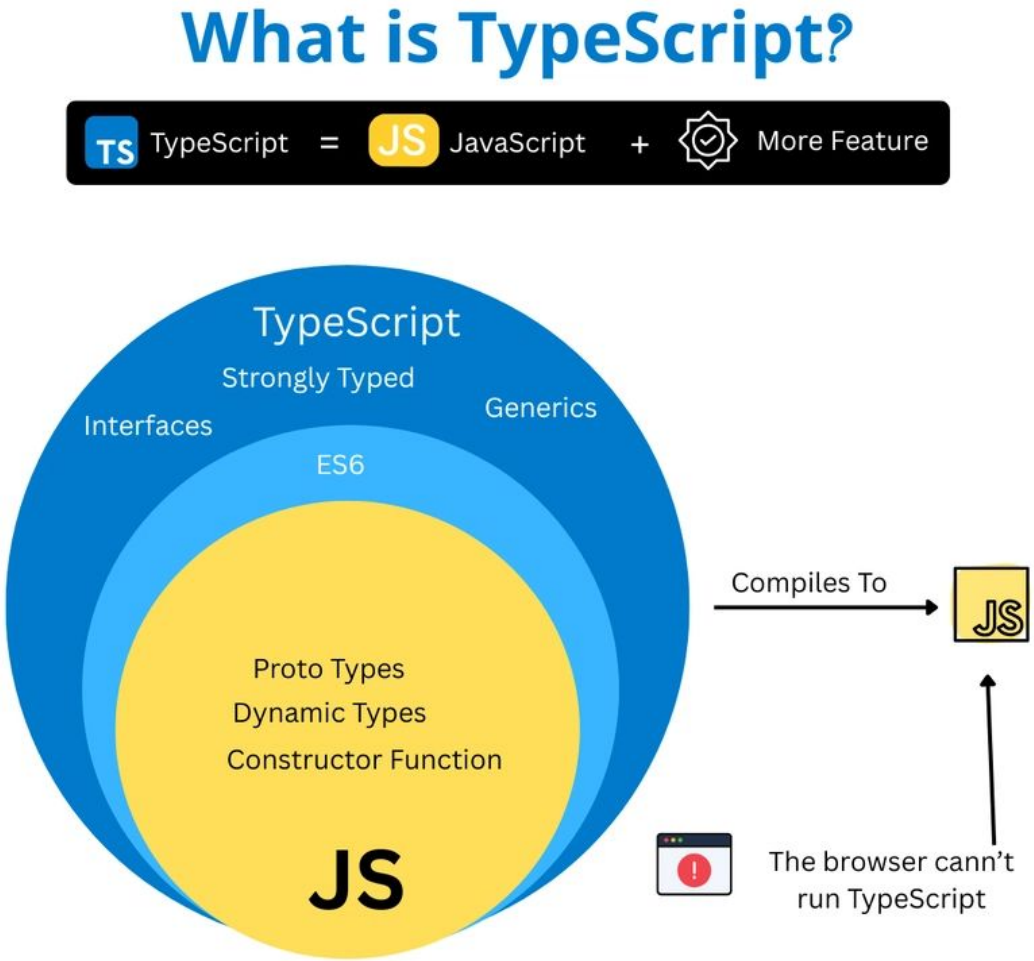
# Revisão de Typescript

O TypeScript é uma linguagem de programação desenvolvida pela Microsoft que estende o JavaScript, adicionando suporte para tipos estáticos opcionais. Isso ajuda a evitar erros comuns durante o desenvolvimento e melhora a qualidade do código.



# Revisão de TypeScript

O TypeScript é compilado para JavaScript antes de ser executado em um navegador ou em um ambiente Node.js. Isso significa que os erros de tipo podem ser capturados antes da execução do código.



# Revisão de TypeScript

- Criação de variáveis
  - tipos primitivos (number, string, boolean)
  - inferência de tipos
  - array(tipo[] ou Array<tipo>)
  - any
  - union type -> tipo | tipo | undefined
  - tipos literais
- Objetos
  - Type alias e interface
- String ‘, “, `
- If, else if, else
- Operador ternário
- Loops

# Revisão de TypeScript

- Funções
  - modos diferentes de declarar
  - tipo dos parâmetros
  - parâmetros opcionais e com valor padrão
  - tipo do retorno e void
- Arrays
  - desestruturação
  - spread operator → cópia
  - for…of
  - forEach, map, filter

# Exercício

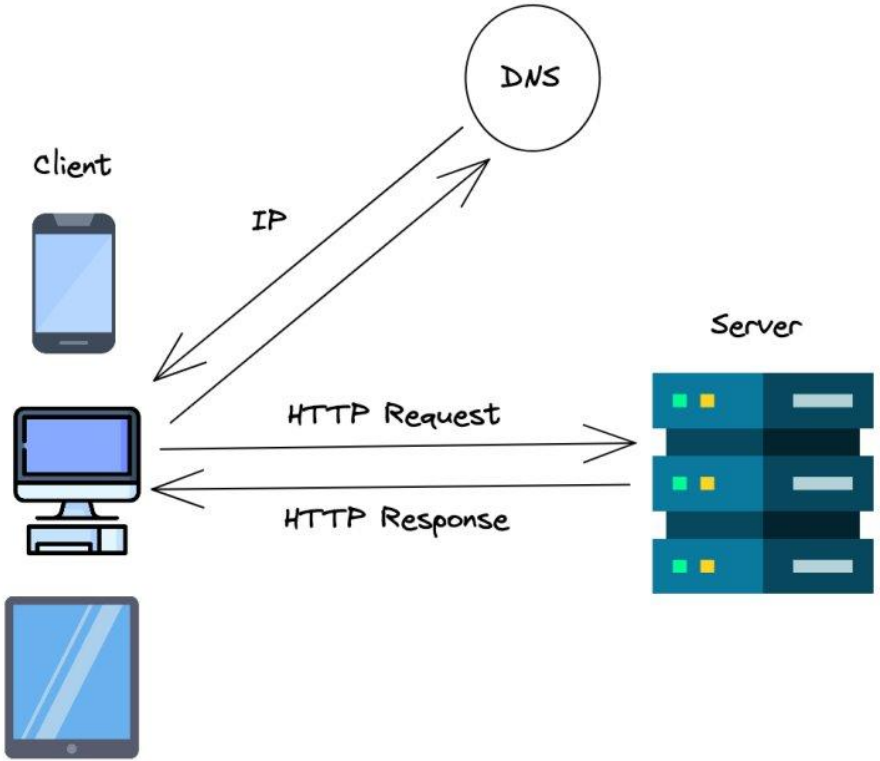
### Exercício: Modelando um Perfil de Usuário

**Instruções:**

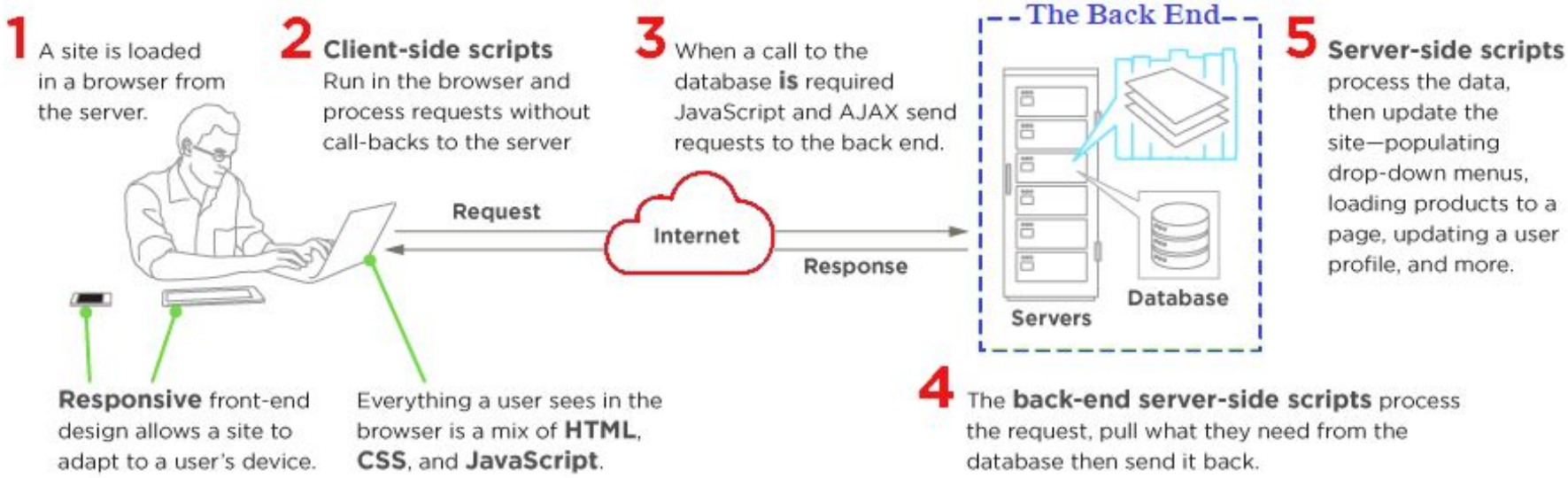
1. Crie um alias de tipo (`type`) chamado `ID` que possa ser `string` ou `number`.
2. Crie um alias de tipo (`type`) chamado `Status` com os valores literais `"ativo"` ou `"inativo"`.
3. Crie uma interface (`interface`) chamada `Perfil` com as seguintes propriedades:
    - `id`: do tipo `ID`.
    - `nome`: do tipo `string`.
    - `idade`: do tipo `number`.
    - `status`: do tipo `Status`, sendo uma propriedade **opcional**.
4. Crie um array chamado `listaDePerfis` tipado como `Perfil[]` e adicione dois objetos a ele.
5. Crie uma função chamada `exibirPerfil` que receba um `Perfil` como parâmetro, não retorne nada (`void`) e imprima os dados no console.
6. Use um laço de repetição para percorrer a `listaDePerfis` e exibir cada perfil

# Arquitetura fullstack

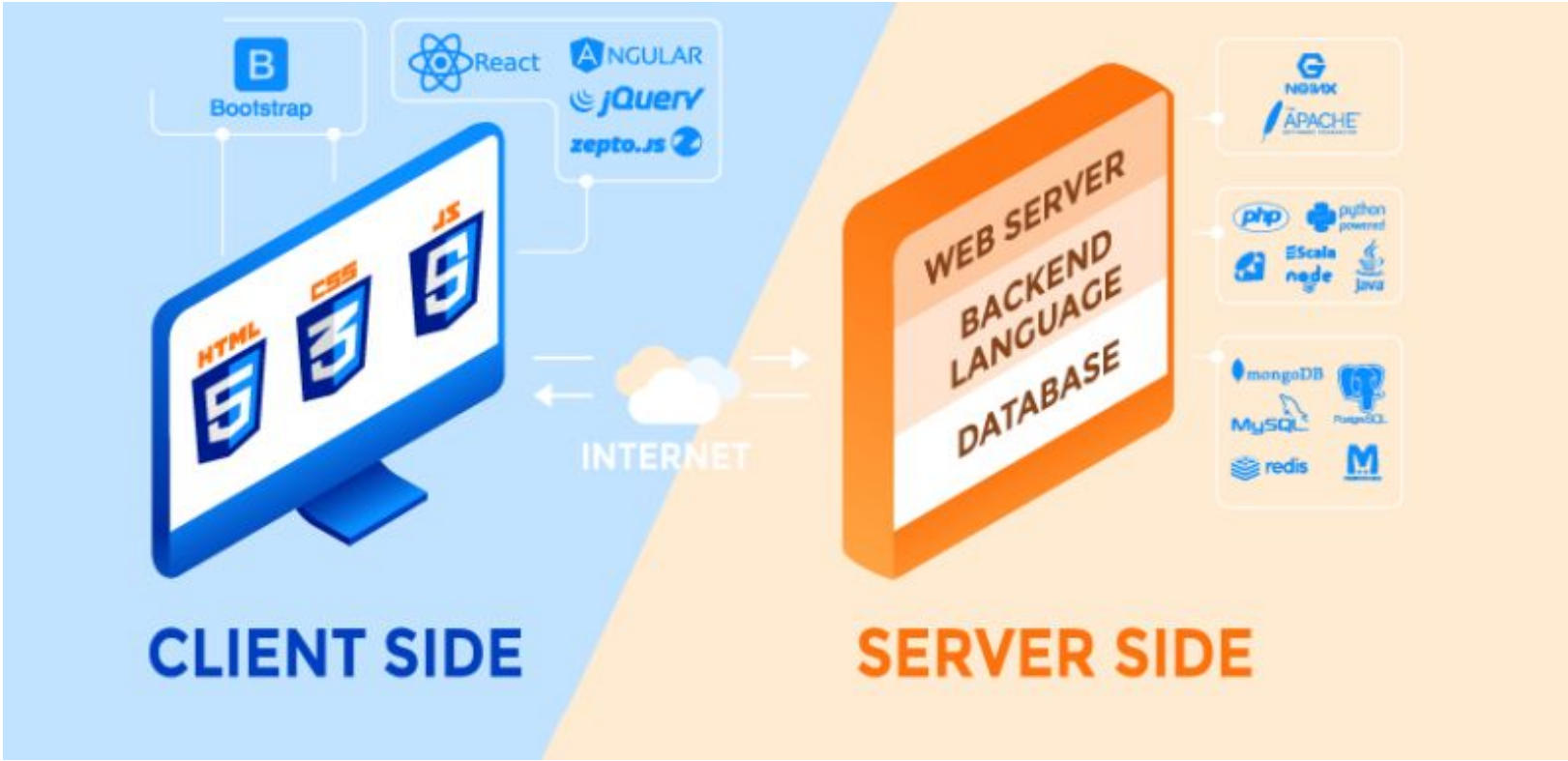
# Arquitetura cliente – servidor



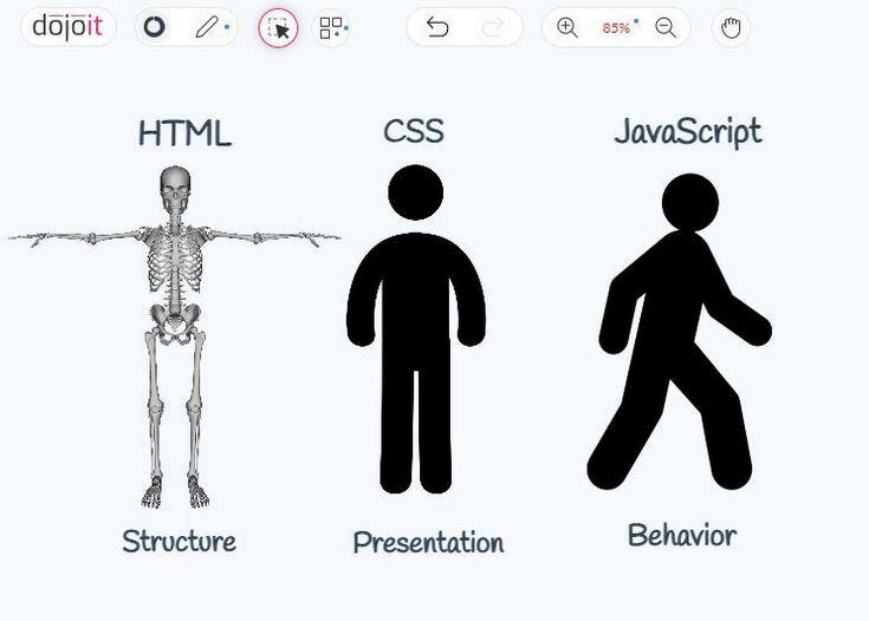
# Arquitetura cliente – servidor

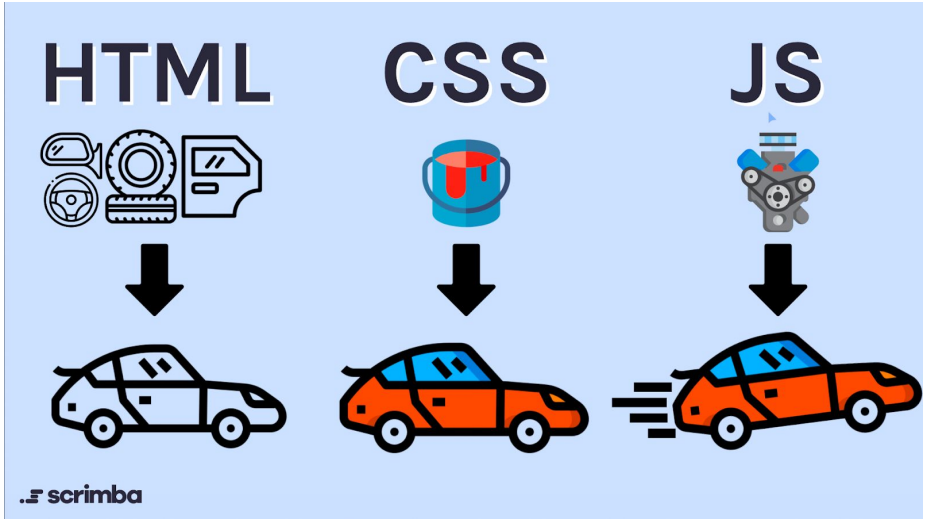


# Arquitetura cliente – servidor

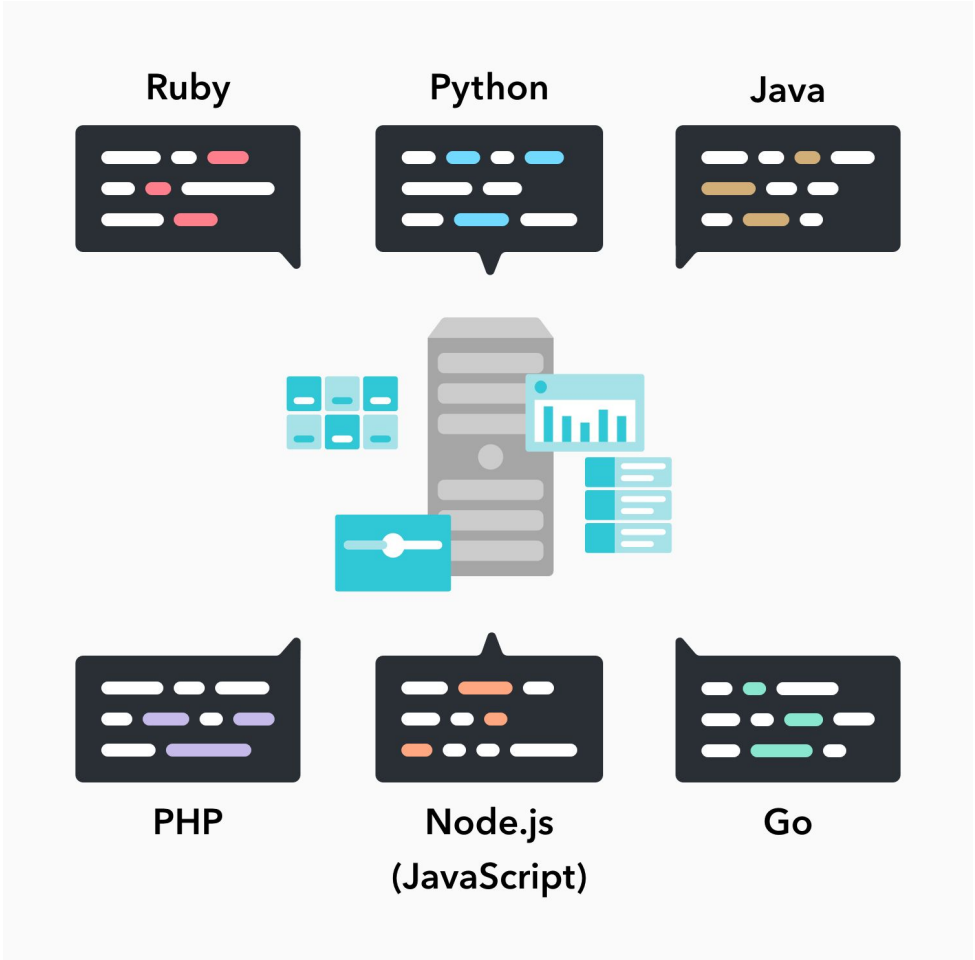


# Client side


dojoit
85%
HTML
CSS
JavaScript
Structure
Presentation
Behavior

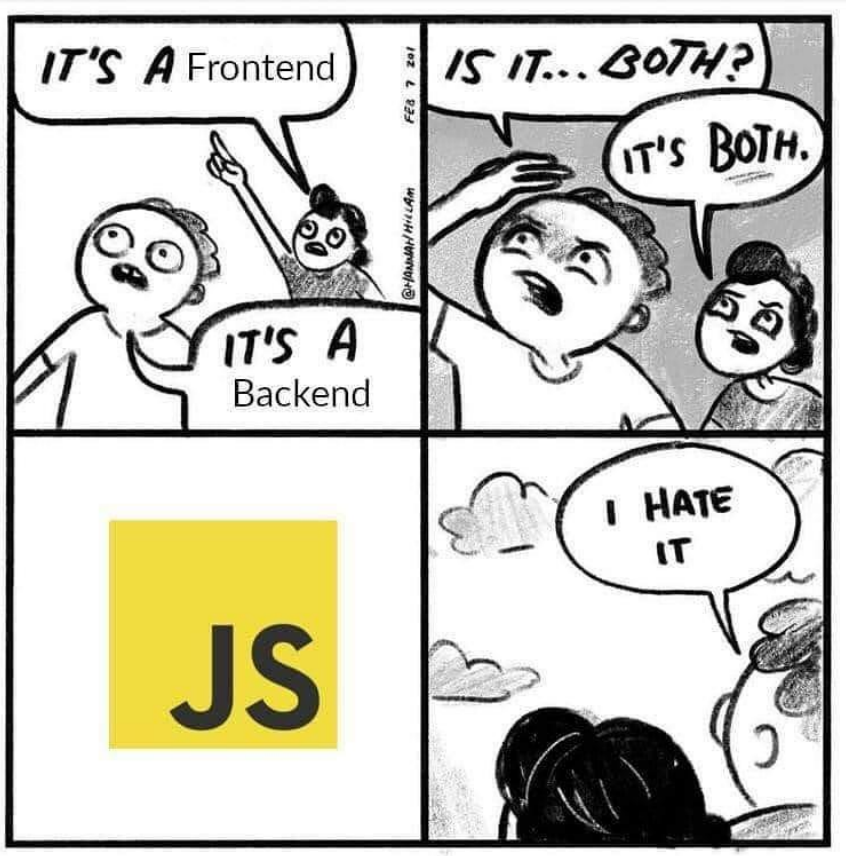

HTML
CSS
JS
scrimba

# Server side



# Frontend vs Backend

# Javascript

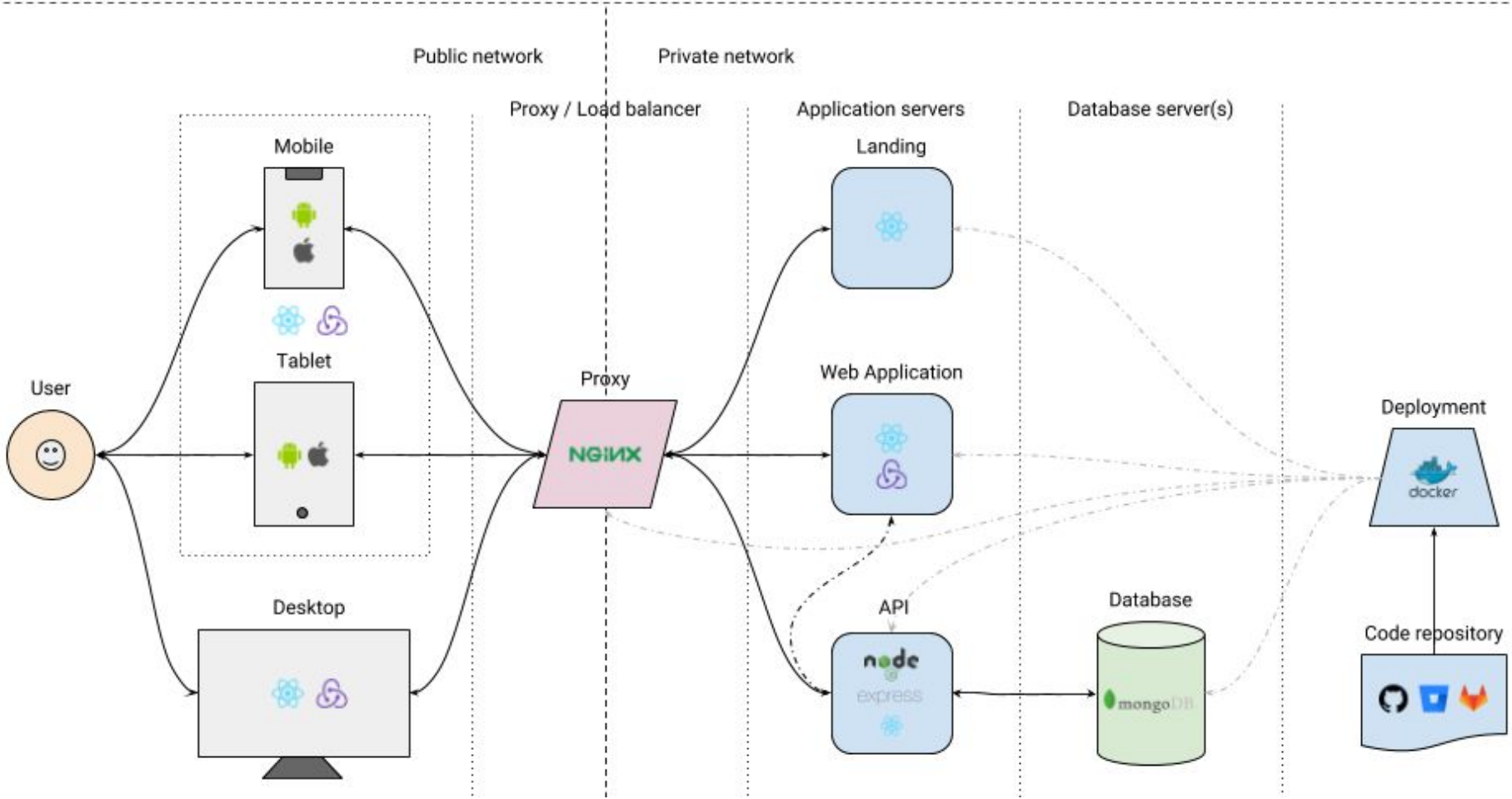


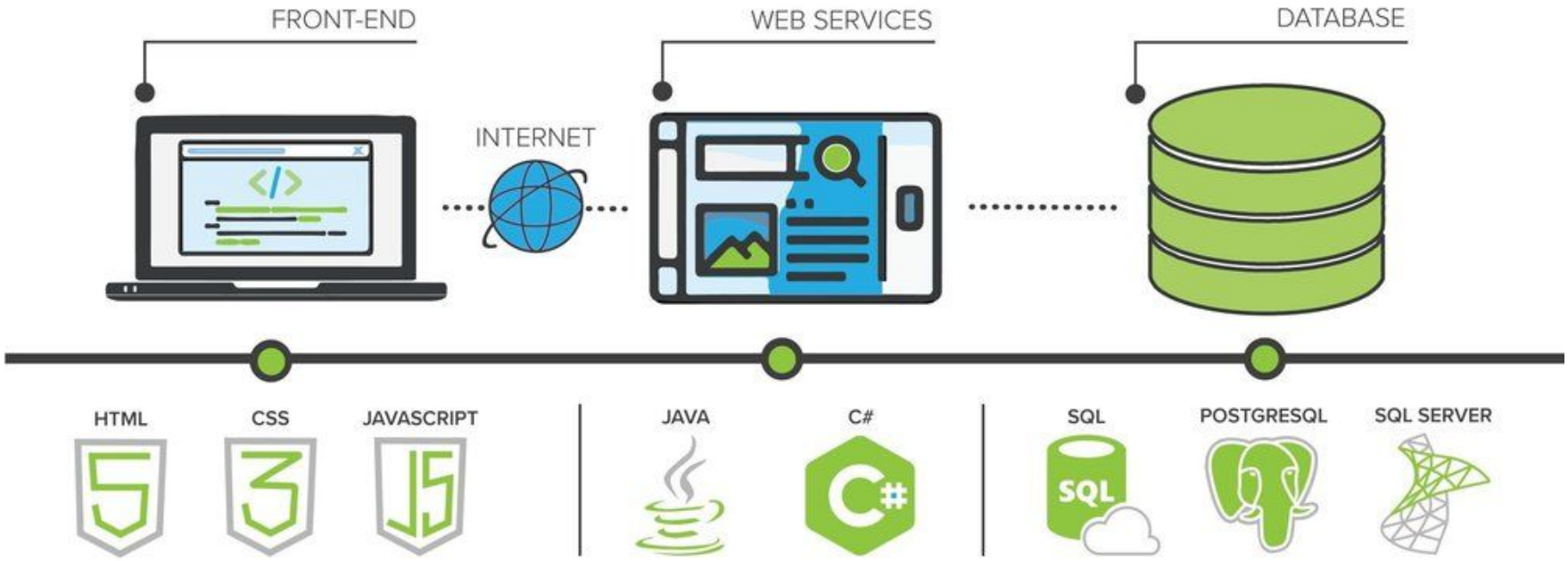
# Javascript

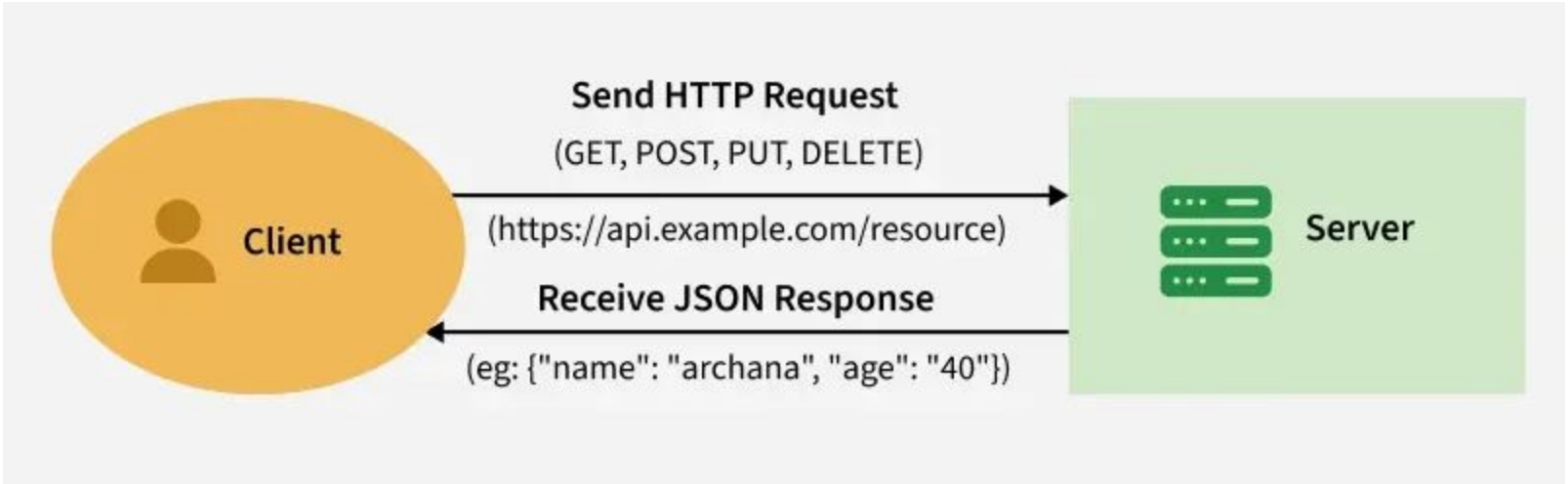
- Web (client-side e server-side)
- Mobile (react native)
- Smartwatch
- TV
- Micro-controladores e IoT

# Full-Stack JavaScript Architecture

github.com/atulmy

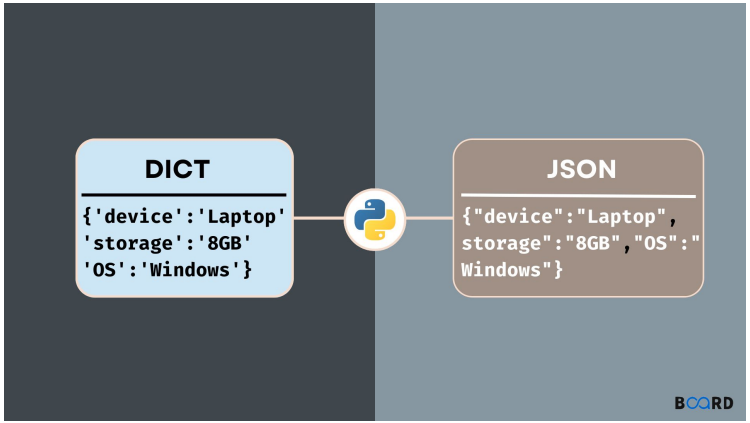


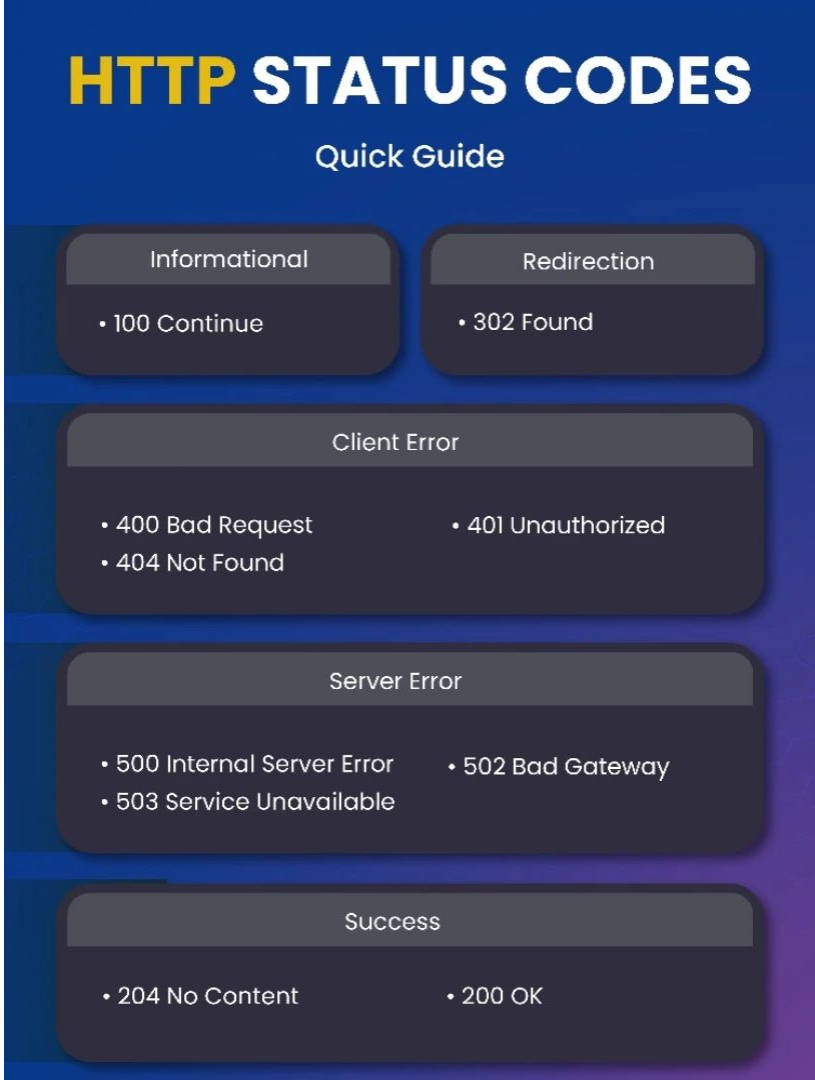

FRONT-END
WEB SERVICES
DATABASE
INTERNET
HTML
CSS
JAVASCRIPT
JAVA
C#
SQL
POSTGRESQL
SQL SERVER


Send HTTP Request
(GET, POST, PUT, DELETE)
Client
(https://api.example.com/resource)
Server
Receive JSON Response
(eg: {"name": "archana", "age": "40"})

# HTTP

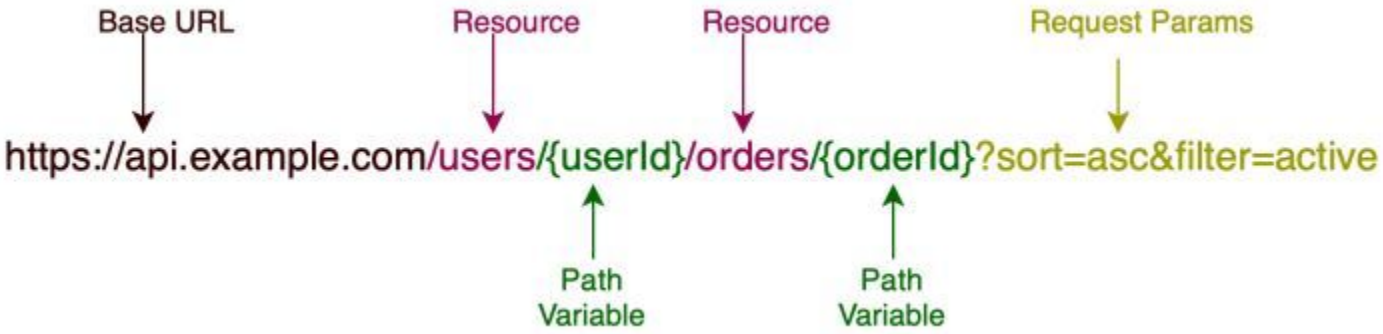
- Request
  - Método (GET, POST, PUT, DELETE, …)
  - URL (.../resource/…)
  - Body (JSON)
- Response
  - Código (200, 201, 204, 400, 404, 500, 503,...)
  - Body (JSON)



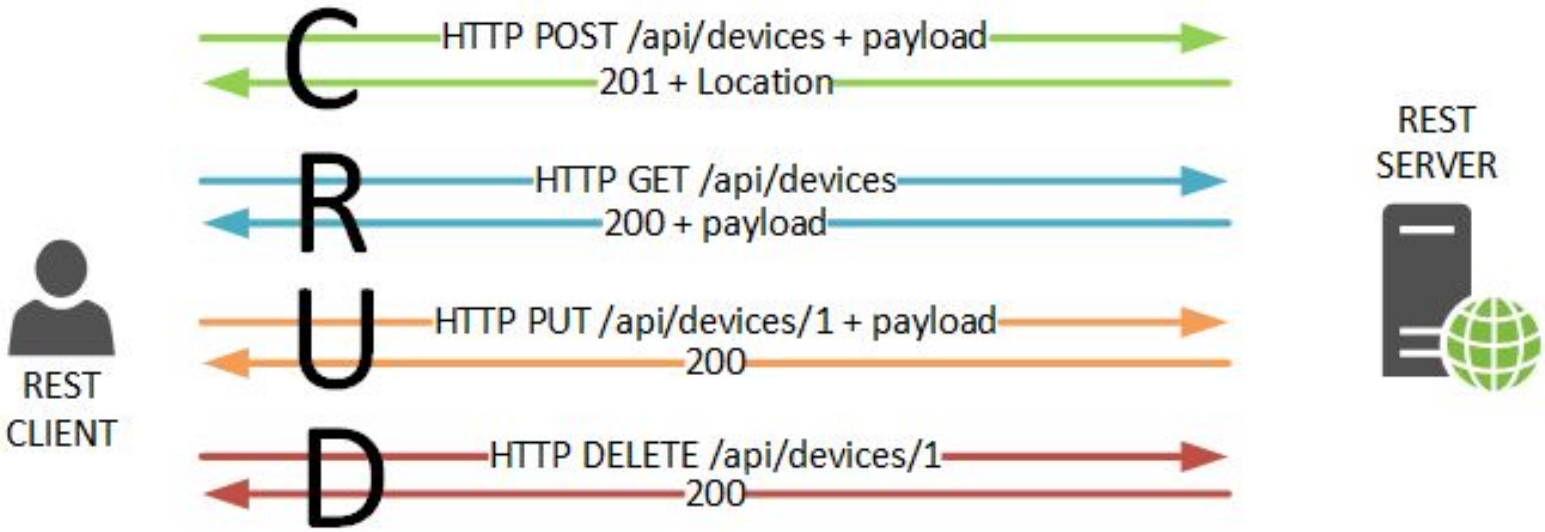



# How RESTful APIs use HTTP methods to perform database operations

| REST operations | CRUD operations |
| --- | --- |
| **GET:** Used to retrieve data. | **Read:** Retrieving data. |
| **POST:** Used to create new data. | **Create:** Adding new data. |
| **PUT:** Used to update existing data. | **Update:** Modifying existing data. |
| **DELETE:** Used to delete data. | **Delete:** Removing data. |

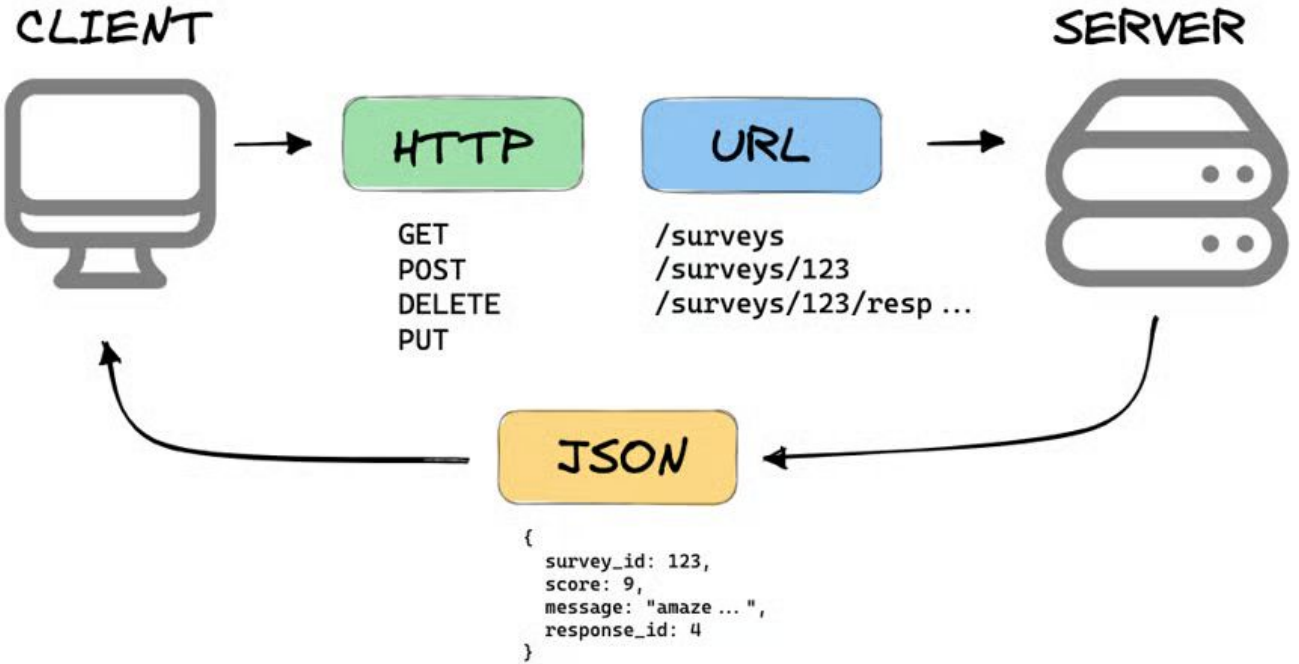

Base URL
Resource
Resource
Request Params
https://api.example.com/users/{userId}/orders/{orderId}?sort=asc&filter=active
Path Variable
Path Variable


HTTP Method
Content type
Base URL
API call
Payload
HTTP POST https://<mgmt>/web_api/login
Headers Content-Type: application/json
Body
{
"user" : "Jim",
"password" : "MyPwd",
"domain" : "Nordics"
}


C
HTTP POST /api/devices + payload
201 + Location
R
HTTP GET /api/devices
200 + payload
U
HTTP PUT /api/devices/1 + payload
200
D
HTTP DELETE /api/devices/1
200
REST CLIENT
REST SERVER

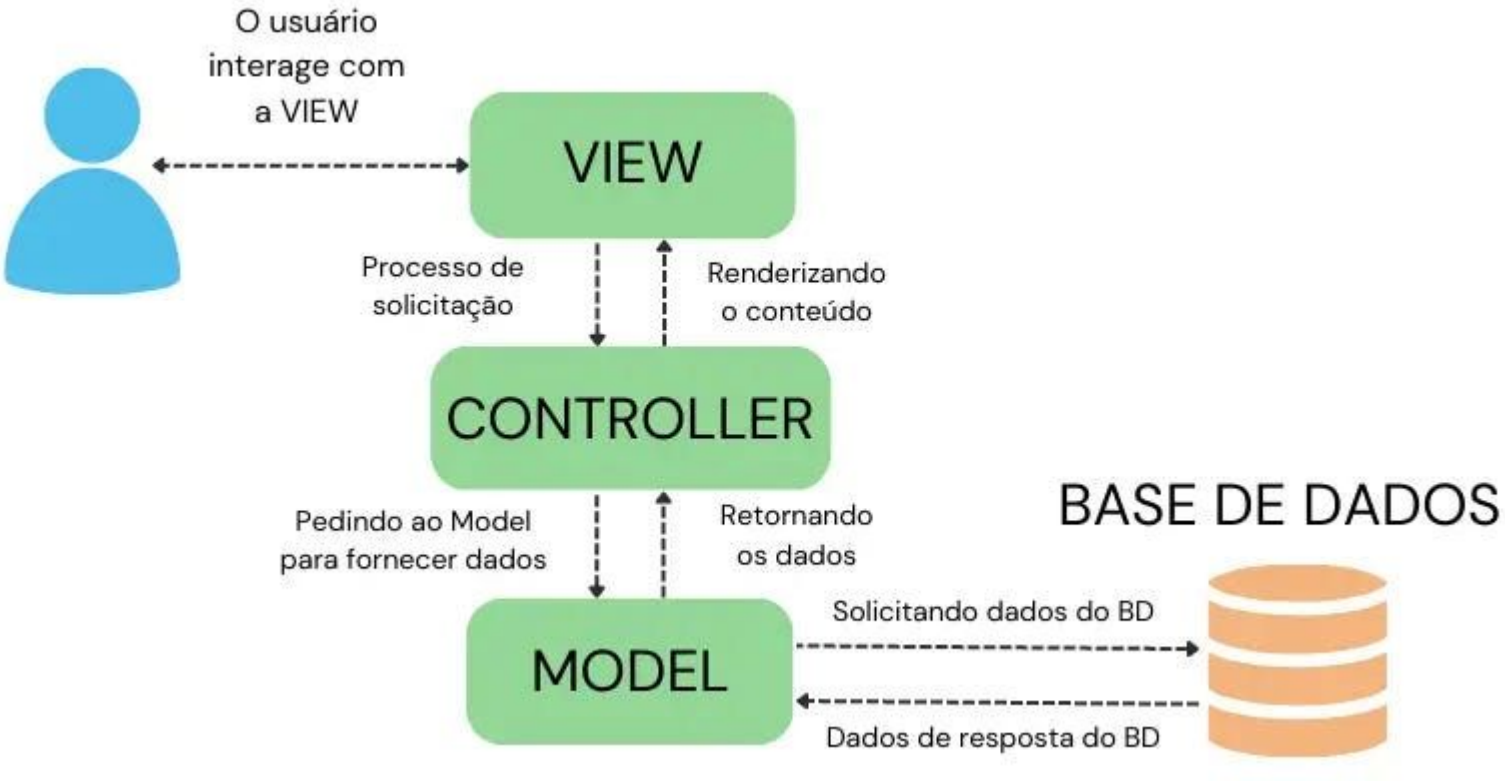
| Task | Method | Path |
| --- | --- | --- |
| Create a new task | POST | /tasks |
| Delete an existing task | DELETE | /tasks/{id} |
| Get a specific task | GET | /tasks/{id} |
| Search for tasks | GET | /tasks |
| Update an existing task | PUT | /tasks/{id} |

# WHAT IS A REST API?

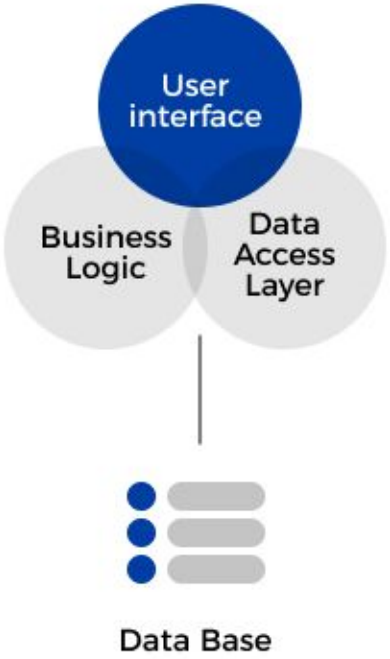



mannhowie.com

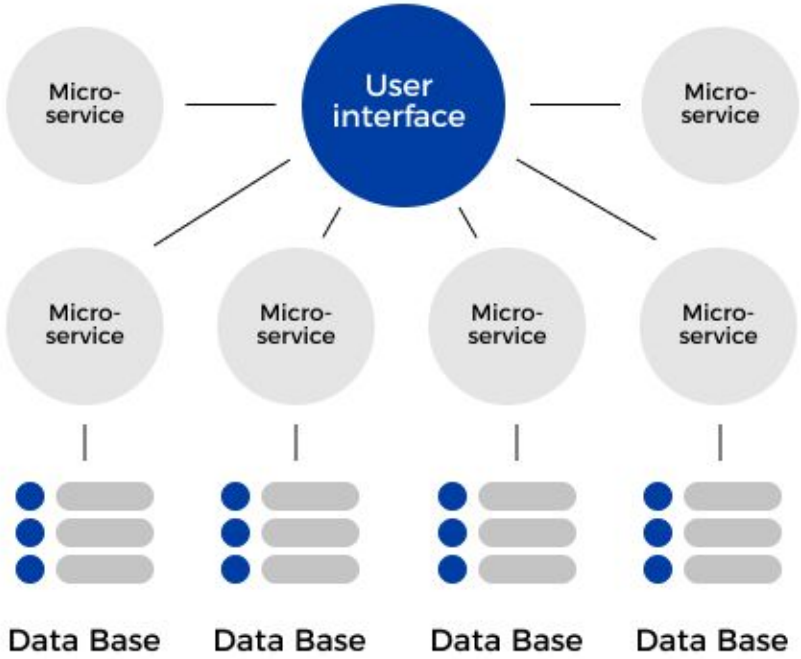
# MVC


O usuário
interage com
a VIEW
VIEW
Processo de
solicitação
Renderizando
o conteúdo
CONTROLLER
Pedindo ao Model
para fornecer dados
Retornando
os dados
MODEL
BASE DE DADOS
Solicitando dados do BD
Dados de resposta do BD

## MONOLITHIC ARCHITECTURE




## MICROSERVICE ARCHITECTURE



# Aplicação a ser construída

- Sistema de clínica médica
  - Resources:
    - usuários (secretários), pacientes, médicos, consultas
  - Endpoints:
    - findOne, findAll, create, update, delete
  - Telas:
    - Login, home, cadastro/alteração de paciente, cadastro/alteração de consulta

# Roteiro da criação da API

- Criação da conta no Github/Firebase
- Criação do codespace/projeto
- Inicialização da app com express e ts
- Configurar package.json
- Configurar compilador do ts
- Criação do arquivo principal
- Rodar app
- Expor porta
- Testar com apidog

# Estrutura de diretórios da API

```
├── src
│   ├── controllers
│   ├── routes
│   ├── services
│   └── index.ts
├── .env
├── tsconfig.json
└── package.json
```

# Comandos iniciais

npm init -y

npm install express

npm install --save-dev typescript @types/express @types/node ts-node nodemon

npx tsc --init

# tsconfig

```
{
  "compilerOptions": {
    "target": "es6", // Versão do JavaScript de saída
    "module": "commonjs", // Sistema de módulos
    "rootDir": "./src", // Onde nosso código .ts estará
    "outDir": "./dist", // Onde o JavaScript compilado será salvo
    "esModuleInterop": true, // Permite compatibilidade entre módulos CommonJS e ES
    "forceConsistentCasingInFileNames": true,
    "strict": true, // Habilita todas as checagens de tipo estritas
    "skipLibCheck": true
  }
}
```

# Comandos iniciais

criar diretório src

altera no package.json:

```
"scripts": {
  "build": "tsc",
  "start": "node dist/index.js",
  "dev": "nodemon src/index.ts"
},
```

# Início da API

https://gist.github.com/douglasmeneses/c84b8a72408eacbebc523bb44caba36b

# Criando os demais endpoints e tipando

https://gist.github.com/douglasmeneses/11642e291ec25cc028f28b9938c4c21b

# Integrando com Banco de Dados

1) Criar novo projeto com template PostgreSQL
2) Configura novo projeto TS e Express
   a) Testar o tsx ao invés do nodemon
   b) "dev": "tsx watch src/index.ts"
3) Configurar o Prisma
   a) npm install prisma  e npx prisma init
   b) na extensão de SQL:
      i) criar banco (CREATE DATABASE clinica)
      ii) criar nova conexão no banco clinica
      iii) criar um .env com

```
DATABASE_URL="postgresql://user:mypassword@localhost:5432/clinica?sslmode=disable"
```

4) atualizar o schema.prisma

```
model User {
  id    Int     @id @default(autoincrement())
  nome  String
  idade Int

@@map("users")
}
```

5) Atualizar index.ts

https://gist.github.com/douglasmeneses/8fece7181174c420cb293d72253a8b12

6) npx prisma db push

7) Criar diretórios (routes, controllers, services, db)

8) Resultado final:
https://gist.github.com/douglasmeneses/85ece19d38cc48afebd84c1f2678669b

9) Replicar para resource médico e paciente

# Finalizando a API

1) npm install dotenv @prisma/client
2) npx prisma migrate dev --name init  [no lugar do db push]
3) 
```
npm install swagger-ui-express swagger-jsdoc @types/swagger-ui-express @types/swagger-jsdoc
```
4) validações com zod

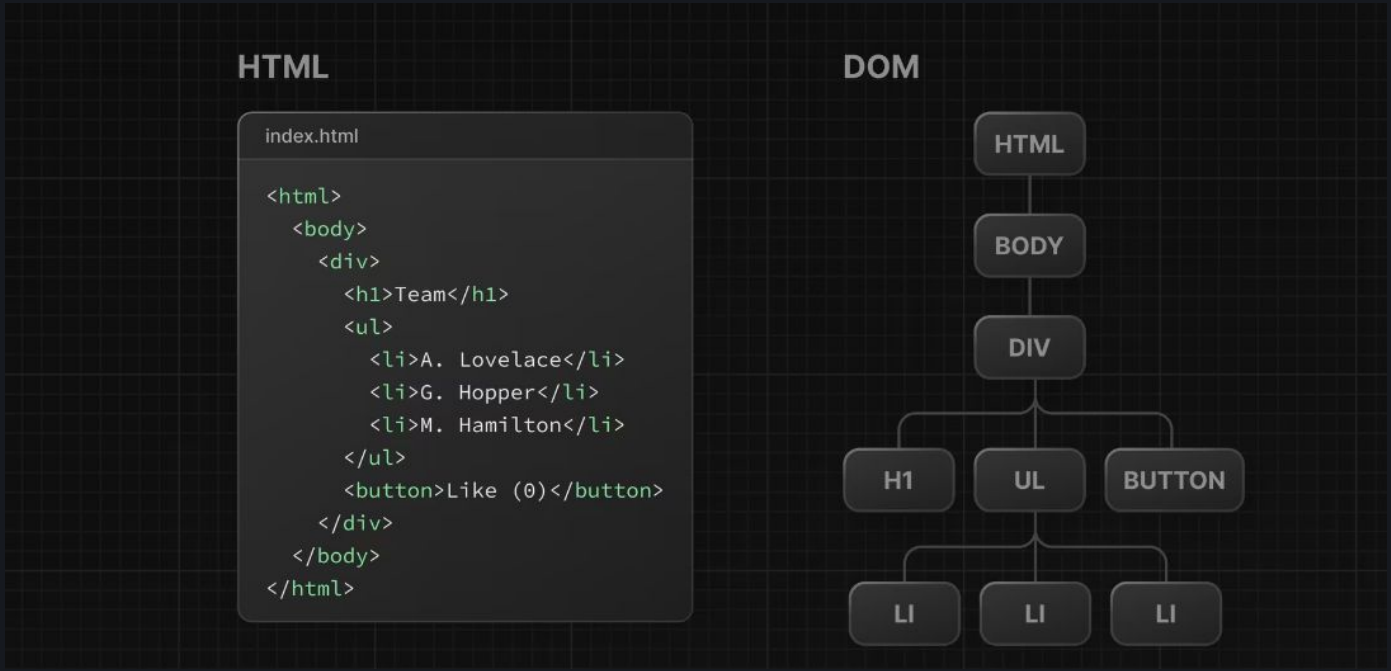
https://github.com/douglasmeneses/clinica-api

# Frontend

# Javascript no Client-side

# Browsers e HTML

Quando um usuário visita uma página web, o servidor retorna um arquivo HTML para o browser. Ele então lê o HTML e constrói o DOM (Document Object Model)



# DOM(Document Object Model)

* É uma representação em forma de objeto dos elementos HTML

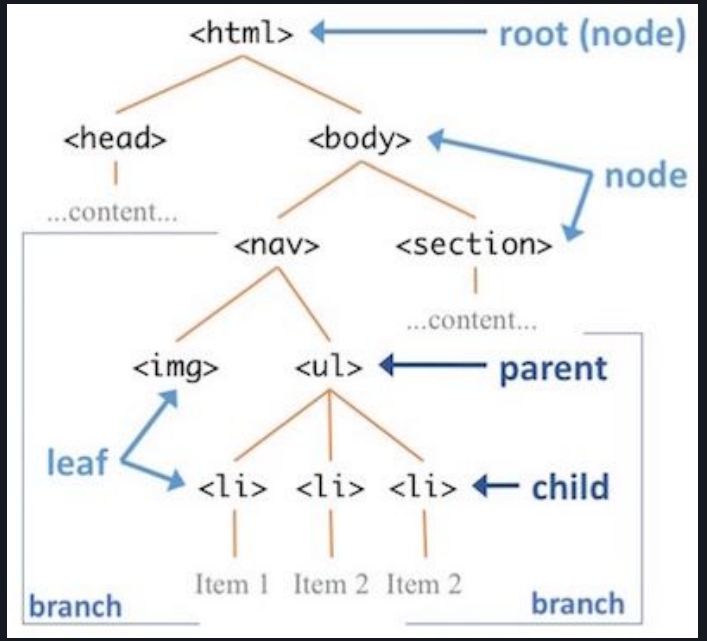
* Funciona como uma ponte entre o código escrito e a interface visualizada pelo usuário

* Em outras palavras: O HTML DOM é um padrão para recuperação, alteração, adição, ou remoção de elementos HTML e suas propriedades.

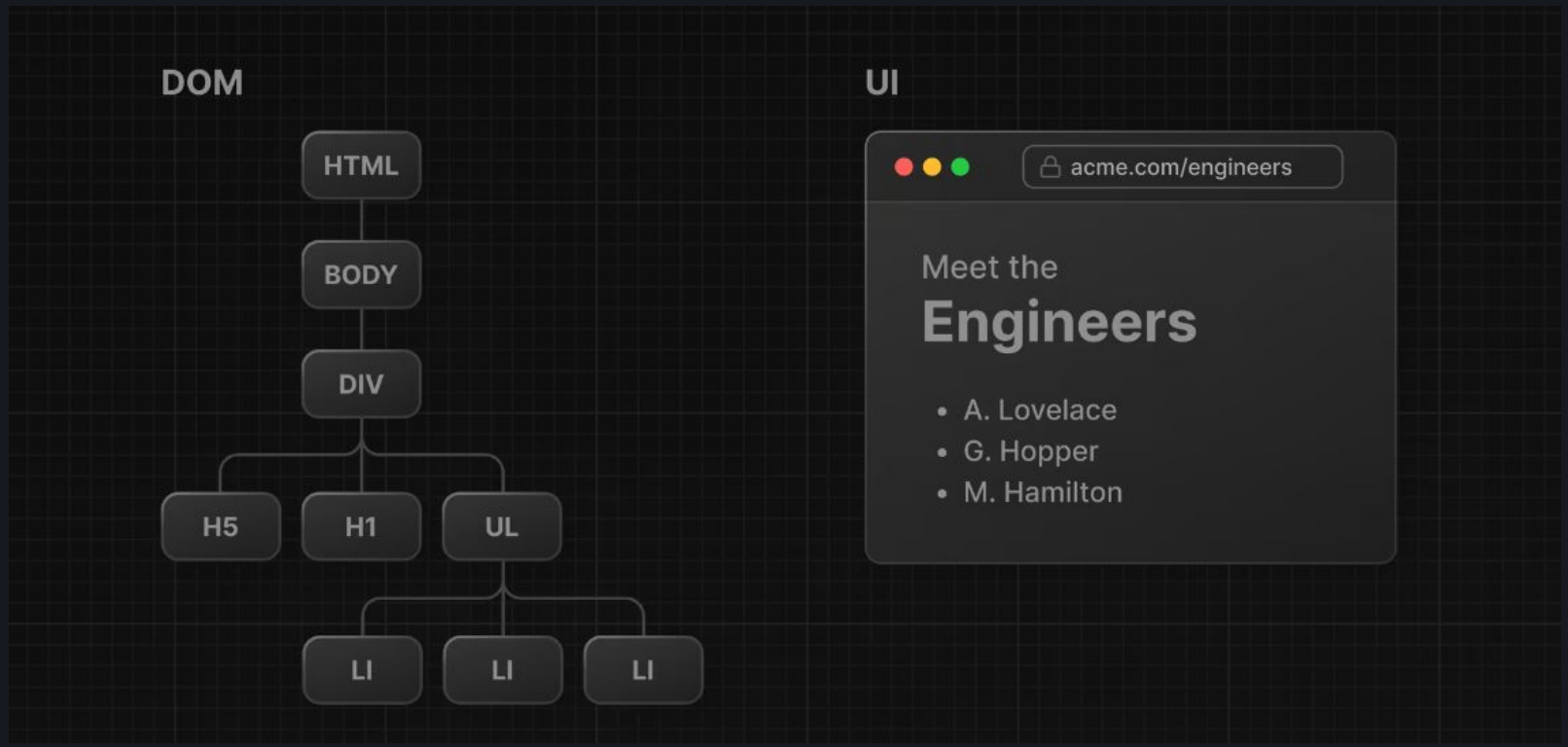
# DOM(Document Object Model)

Tem uma estrutura de árvore (hierarquia) com relacionamento de pai-filho. Ou seja:

* O documento inteiro é um nó
* Cada elemento HTML é um nó
* Os textos dos elementos HTML são nós
* Cada atributo do HTML é um atributo de um nó



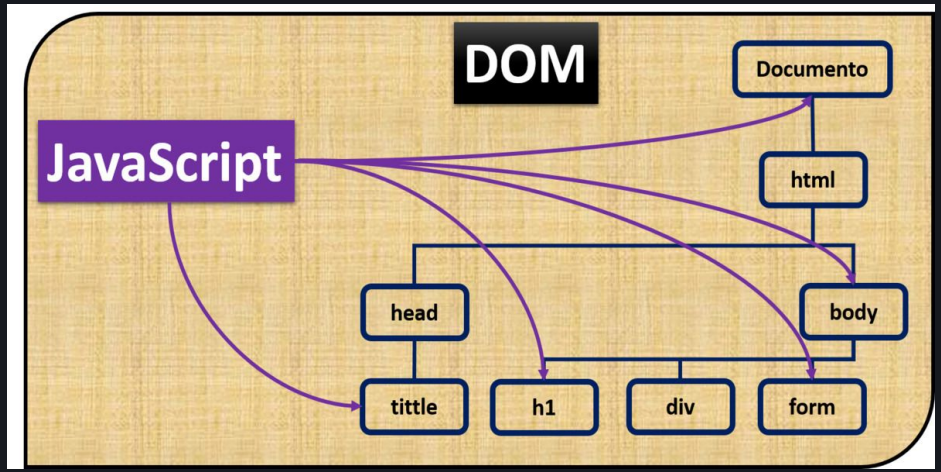
# Browser e DOM



# Javascript e DOM

Você pode usar métodos do DOM e JavaScript para escutar eventos do usuário e manipular o DOM selecionando, adicionando, atualizando e excluindo elementos específicos na interface do usuário.

A manipulação do DOM permite não apenas direcionar elementos específicos, mas também alterar seu estilo e conteúdo.



# Escrevendo JS dentro do html

```
<script> ou <script type=”text/javascript”>
//código js
</script>

ou

<script src="myscripts.js"></script>
```

# Manipulando o DOM → document.

```
<html>
 <head></head>
 <body>
   <script type="text/javascript">
     var d = new Date();
     var hora = d.getHours();
     if (hora < 12) document.write("<b> Bom dia! </b>");
     else if (hora < 18) document.write("<b> Boa tarde! </b>");
     else document.write("<b> Boa noite! </b>");
   </script>
 </body>
</html>
```

# Eventos

- Eventos são ações - em geral do usuário - que podem ser detectadas em javascript
- A partir da implementação de eventos, podemos criar páginas dinâmicas
- Cada elemento em HTML têm um próprio conjunto de eventos que podem ser capturados
- A especificação dos eventos que serão “escutados” é definido nas tags HTML

# Principais eventos

## Common HTML Events

Here is a list of some common HTML events:

| Event | Description |
| --- | --- |
| onchange | An HTML element has been changed |
| onclick | The user clicks an HTML element |
| onmouseover | The user moves the mouse over an HTML element |
| onmouseout | The user moves the mouse away from an HTML element |
| onkeydown | The user pushes a keyboard key |
| onload | The browser has finished loading the page |

# Exemplo

```
//Elementos do Body do HTML
<h1 onmouseover="mouseOverDisciplina()">Disciplina Programação Web</h1>
<h2 id="aula">Eventos</h2>
<input type="text" id="texto"/>
<button id="botao">Clique</button>


<script>
function mouseOverDisciplina() {
        alert("Mouse Over!");
    }
</script>
```

# DevTools

Se você observar os elementos do DOM nas ferramentas de desenvolvedor do seu navegador, notará que o DOM inclui o elemento <h1>. O DOM da página é diferente do código-fonte – ou, em outras palavras, do arquivo HTML original que você criou.

## DOM

Developer Tools Elements

```
<html>
  <head></head>
  <body>
    <div id="app">
      <h1>Develop. Preview. Ship. 🚀</h1>
    </div>
    <script type="text/javascript">...</script>
  </body>
</html>
```

## Source Code (HTML)

index.html

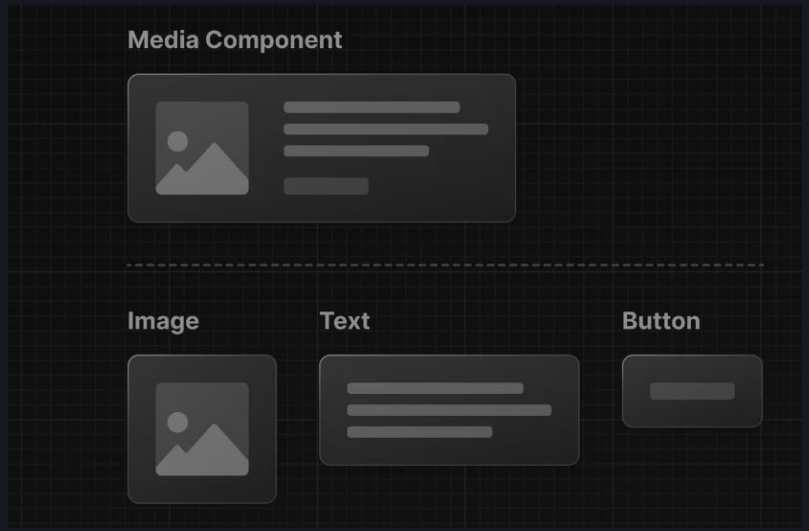
```
<html>
  <head></head>
  <body>
    <div id="app"></div>
    <script type="text/javascript">...</script>
  </body>
</html>
```

# React

- Referência: https://nextjs.org/learn/react-foundations
    - React is a JavaScript library for building interactive user interfaces (UI).
- Programação declarativa (me dê isto) vs. imperativa (faça isso)
    - Programação imperativa é como dar a um chef instruções passo a passo de como fazer uma pizza. Programação declarativa é como pedir uma pizza sem se preocupar com as etapas necessárias para prepará-la.
    - Como desenvolvedor, você pode dizer ao React o que deseja que aconteça na interface do usuário, e o React descobrirá os passos de como atualizar o DOM em seu lugar.
- Lib JS/TS criada pelo Facebook
- Arquivos .jsx/tsx
    - Descrevem UI num modo semelhante à sintaxe do HTML
    - Babel: transforma JSX em JS

# React - conceitos principais

- Componentes
  - funções que retornam elementos de UI (que podem ser outros componentes ou tags html)
  - Representam cada elemento de UI da tela
  - São reutilizáveis




```
1  function Header() {
2    return <h1>Develop. Preview. Ship.</h1>;
3  }
4
5  function HomePage() {
6    return (
7      <div>
8        <Header />
9        <Header />
10     </div>
11   );
12 }
```

# React - conceitos principais

- Propriedades → props
  - argumentos enviados do componente pai para o filho
  - em forma de objeto

```
1  function Header(props) {
2    console.log(props); // { title: "React" }
3    return <h1>Develop. Preview. Ship.</h1>;
4  }
```

```
function Header({ title }) {
  return <h1>{title ? title : 'Default title'}</h1>;
}
```

```
function HomePage() {
  return (
    <div>
      <Header title="React" />
      <Header title="A new title" />
    </div>
  );
}
```

# React - conceitos principais

- const [variavel, setVariavel] = React.useState(valorInicial)

Estado → useState()

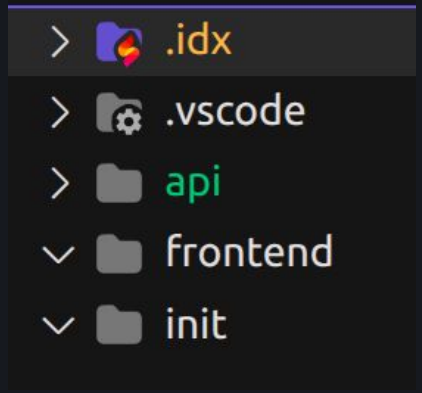
- um dos hooks (funções especiais) do React
- é qualquer informação em sua interface que mude com o tempo, geralmente devido à interação com o usuário
- É armazenado dentro de cada componente

```
function HomePage() {
  // ...
  const [likes, setLikes] = React.useState(0);

  function handleClick() {
    setLikes(likes + 1);
  }

  return (
    <div>
      {/* ... */}
      <button onClick={handleClick}>Likes ({likes})</button>
    </div>
  );
}
```

# Criando o projeto




```
npx create-vite@latest . --template react-ts
```

```
npm install tailwindcss postcss autoprefixer
adiciona @import "tailwindcss"; no index.css
```

```
npm install @mui/material @emotion/react @emotion/styled @mui/icons-material
```

```
criar /src e Login.tsx
```

```
ajustar conteúdo do App.tsx, main.tsx e index.css
```

https://gist.github.com/douglasmeneses/e3e365d878cf1564b04c28f3cf3ed8f3