Web programming

Tópicos de Informática para Automação

Mário Antunes

November 24, 2025

Universidade de Aveiro

Table of Contents i

JavaScript: Visão Geral Detalhada i

O JavaScript (JS) é frequentemente mal compreendido como um “brinquedo de scripting”, mas é uma linguagem sofisticada e de alto nível.

1. **Tipagem Dinâmica e Tipagem Fraca**
   - As variáveis não estão vinculadas a um *tipo de dados* específico.
   - *Porque é que isto importa:* Pode atribuir um Número a uma variável e, mais tarde, atribuir uma String à mesma variável. Isto oferece flexibilidade, mas aumenta o risco de erros em tempo de execução (ex: tentar multiplicar uma string).

JavaScript: Visão Geral Detalhada ii

```
let x = 42;
x = "olá";
console.log(x)
```

2. Orientação a Objetos baseada em Protótipos

- *Como funciona:* Ao contrário das linguagens baseadas em Classes (Java/C++), onde os objetos são instanciados a partir de "plantas" (classes), os objetos JS herdam diretamente de outros objetos (protótipos).
- *Implicação:* A eficiência de memória envolve clonar estruturas existentes em vez de definir hierarquias rígidas.

JavaScript: Visão Geral Detalhada iii

```
let person  = {
  eats: true,
  hasLegs: 2,
  walks(){ console.log('I can walk')}
}
//definir outro objeto
let man = {
  hasBreast: false,
  hasBeard : true,
}
//definir o protótipo de man para o objeto person
man.__proto__ = person;
//definir um terceiro objeto
let samuel = {
```

```
    age: 23
}
//definir o protótipo de samuel para man
samuel.__proto__ = man;
//aceder ao método walk a partir de samuel
console.log(samuel.walks())
//aceder a hasBeard a partir de samuel
console.log(samuel.hasBeard)
```

3. Execução Single-Threaded

- *A Restrição:* O JS tem uma **única Call Stack** (Pilha de Chamadas). Só consegue fazer *uma coisa de cada vez*.

- *O Risco:* Se executar um ciclo matemático pesado (ex: calcular Pi até mil milhões de dígitos), o separador do browser congela totalmente (bloqueio da UI) porque a thread está ocupada.

Programação Sequencial (Procedimental) i

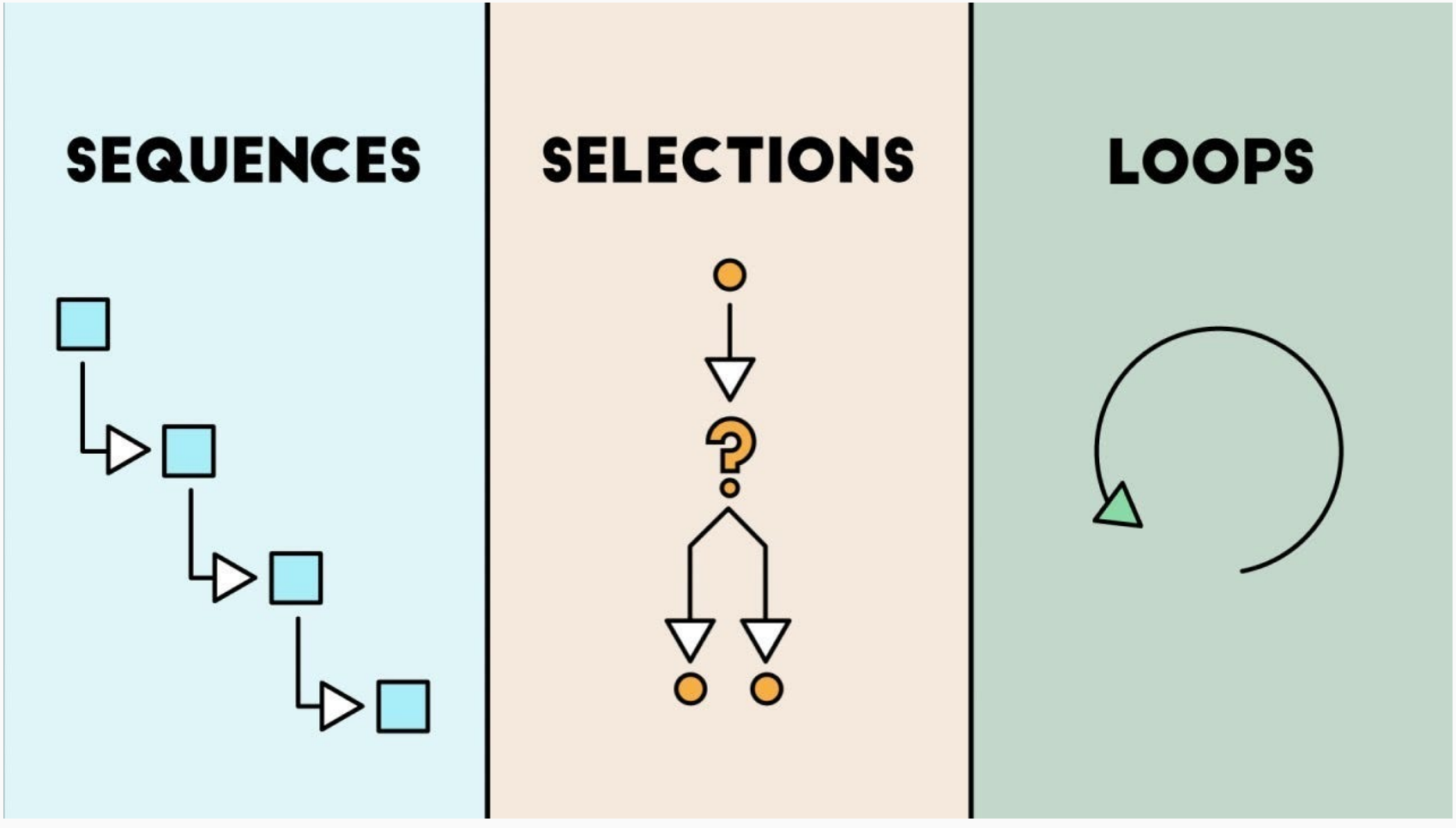
Este é o modelo utilizado em C básico, Fortran ou scripts simples de Python.

A Lógica:

1. O programa inicia.
2. A Linha 1 executa.
3. A Linha 2 executa.
4. **A Linha 3 pede entrada de dados (`scanf`, `input()`).**
5. O programa **PARA** (bloqueia) e espera pelo utilizador. Nada mais acontece até o utilizador carregar no Enter.

Programação Sequencial (Procedimental) ii

Porque é que isto falha na UI: Numa interface web, não podemos **"parar"** o motor de renderização para esperar por um clique do rato. Se o fizéssemos, os botões não animariam e os gifs não seriam reproduzidos.



Programação Orientada a Eventos i

As interfaces modernas (Web, Windows, macOS) utilizam uma arquitetura **Orientada a Eventos** (Event-Driven).

A Lógica:

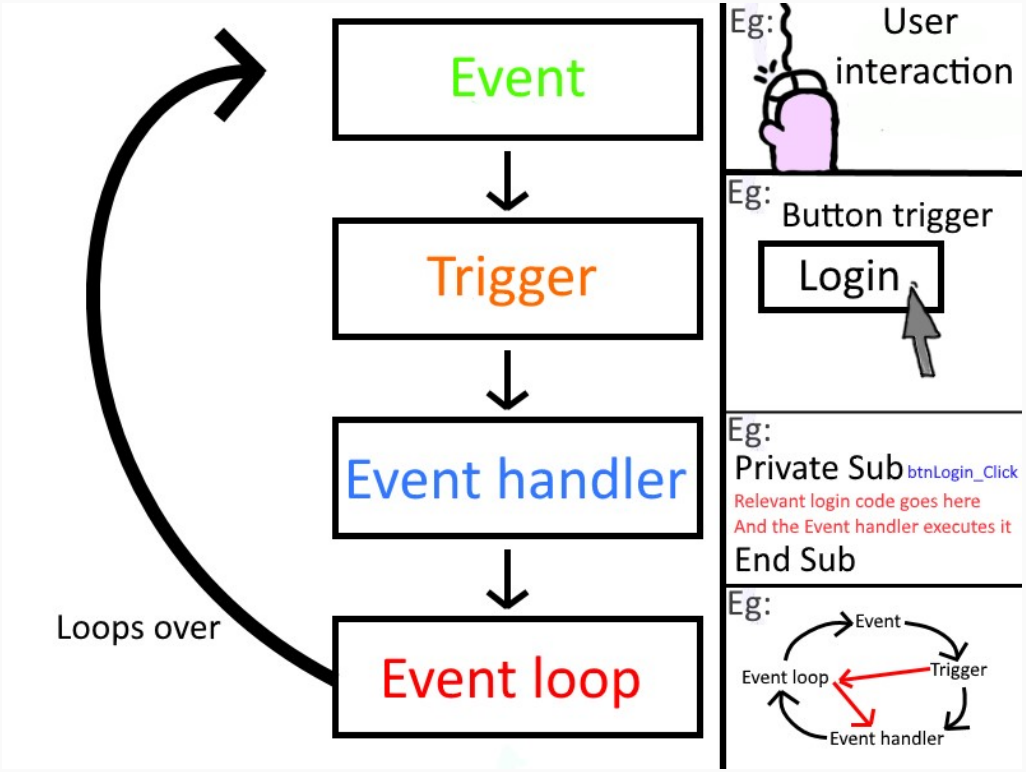
1. O programa inicia (Inicialização).
2. Define "Handlers" (funções à espera de gatilhos específicos).
3. Entra no **Event Loop** (Ciclo de Eventos).
4. O programa fica num **Estado de Escuta** (Listening State).

O "Princípio de Hollywood":

- *Não nos ligue, nós ligamos-lhe.*

Programação Orientada a Eventos ii

- O código não pergunta “O utilizador clicou?”. Em vez disso, o browser interrompe o código dizendo “Acabou de ocorrer um clique, execute a Função de Clique.”



A Cadeia de Eventos i

Como é que uma ação física se torna execução de código?

1. **Nível de Hardware:** O utilizador move o rato. O hardware do rato envia um sinal elétrico (interrupção) ao CPU.
2. **Nível do SO:** O Sistema Operativo (Windows/Linux) interpreta este sinal como uma mudança de coordenadas e pinta o cursor a mover-se.
3. **Nível do Browser:** A janela do browser vê que o cursor está sobre um botão HTML específico e que o botão do rato foi pressionado.

A Cadeia de Eventos ii

4. **O Evento:** O browser cria um Objeto JavaScript `Event` contendo detalhes (coordenadas X/Y, qual o botão, timestamps).
5. **O Listener:** O browser verifica: *Este elemento HTML tem um listener anexado?*
6. **Execução:** Se sim, a função JS registada é empurrada para a pilha de execução (stack).

O Document Object Model (DOM)

O Conceito: Quando escreve um ficheiro HTML, é apenas uma string de texto. O browser analisa (faz o parse) desta string para uma estrutura em memória chamada DOM.

- **HTML:** `<div id="app"></div>` (Texto no disco rígido)
- **DOM:** `HTMLDivElement` (Objeto na RAM)

Porque é que o JS usa o DOM: O JavaScript não pode editar o ficheiro de texto no servidor. Ele edita o **Objeto na RAM**. O motor de renderização do browser vigia constantemente o DOM; quando o JS atualiza o objeto DOM, o browser "repinta" o ecrã.

Estratégias de Execução e Carregamento i

O HTML é analisado sequencialmente (de cima para baixo). Quando o parser vê uma tag `<script>`, pausa a análise do HTML para descarregar e executar o script. Isto cria problemas:

1. **O Truque "Bottom of Body"**
   - *Técnica:* Colocar o `<script>` logo antes de `</body>`.
   - *Raciocínio:* Garante que todos os elementos HTML existem no DOM antes que o script tente encontrá-los.
2. **O Atributo `defer` (Padrão Moderno)**

```
<script src="app.js" defer></script>
```

Estratégias de Execução e Carregamento ii

- *Comportamento:* O script é descarregado em segundo plano (paralelo) enquanto o HTML é analisado.
- *Execução:* O browser garante que o script só correrá **depois** de o HTML estar totalmente analisado, mas **antes** do evento `DOMContentLoaded`.
- *Benefício:* Tempos de carregamento de página mais rápidos e acesso seguro ao DOM.

O Event Loop (Detalhe Técnico) i

Como é que o JS single-threaded lida com tarefas assíncronas (como obter dados) sem congelar?

1. **Call Stack:** Executa código síncrono (LIFO - Last In, First Out).
2. **Web APIs:** Quando chama `setTimeout` ou `fetch`, o "trabalho" é delegado às threads C++ do Browser (não à thread JS).
3. **Callback Queue (Fila):** Quando a Web API termina, coloca a sua função de callback numa Fila.
4. **O Loop:** O Event Loop verifica: *"A Stack está vazia?"*
    - Se **NÃO**: Espera.
    - Se **SIM**: Move o primeiro item da Fila para a Stack.

O Event Loop (Detalhe Técnico) ii

É por isto que `setTimeout(fn, 0)` *não corre imediatamente — espera que a stack fique limpa.*

1. Manipulação de Eventos do Rato i

Usamos `addEventListener`. Esta é a fase de registo da programação Orientada a Eventos.

```
const box = document.querySelector('#box');
// O objeto 'event' é passado automaticamente pelo brow
function handleMove(event) {
    // Atualizar texto com coordenadas do rato
    box.textContent = `X: ${event.clientX}, Y: ${event.
    // Estilo dinâmico baseado em lógica
    if (event.clientX > 500) {
        box.style.backgroundColor = 'red';
    } else {
        box.style.backgroundColor = 'blue';
```

```
    }
}
// Subscrever o evento 'mousemove'
box.addEventListener('mousemove', handleMove);
```

2. Conteúdo Dinâmico (Biblioteca de Fotos) i

Podemos criar a interface programaticamente. É assim que o React/Vue funcionam "debaixo do capô" (Abordagem Imperativa).

```
const urls = ['img1.jpg', 'img2.jpg'];
const container = document.getElementById('gallery');

urls.forEach(url ⇒ {
    // 1. Create Element: Cria um objeto órfão em memór
    const img = document.createElement('img');
    // 2. Configure Object: Define propriedades
    img.src = url;
    img.className = 'thumbnail';
```

2. Conteúdo Dinâmico (Biblioteca de Fotos) ii

```
        // 3. Attach Event: Torna-o interativo imediatament
        img.addEventListener('click', () ⇒ {
            console.log("Clicou em " + url);
        });
        // 4. Mount: Insere na árvore DOM viva.
        container.appendChild(img);
    });
```

3. Dados Assíncronos (Fetch API) i

Obter dados de uma API leva tempo (latência). Usamos **Promises** (`async/await`) para evitar bloqueios.

```
async function getData() {
    try {
        // 'await' cede a thread até que a Promise seja
        // A UI permanece responsiva durante esta pausa
        const response = await fetch('[https://api.data
        // O parsing do JSON também é assíncrono (gere
        const data = await response.json();
        console.log(data); // Corre apenas após a rede
    } catch (error) {
        // Lida com falhas de rede (404, 500, Offline)
```

3. Dados Assíncronos (Fetch API) ii

```
            console.error("Fetch falhou:", error);
        }
    }
```

4. Comunicação em Tempo Real (WebSockets) i

HTTP vs. WebSockets:

- **HTTP:** Cliente pede, Servidor responde, Ligação fecha. (Stateless).
- **WebSocket:** Cliente realiza um "Handshake", a Ligação atualiza para socket TCP, a Ligação mantém-se aberta.

```
const socket = new WebSocket('ws://localhost:8080');
// Evento: Ligação Estabelecida
socket.onopen = () ⇒ {
    console.log("Ligado ao Servidor de Chat");
    socket.send("Utilizador entrou");
};
```

4. Comunicação em Tempo Real (WebSockets) ii

```
// Evento: Servidor enviou dados para nós
socket.onmessage = (event) ⇒ {
    // Isto dispara sempre que o servidor envia dados.
    const message = JSON.parse(event.data);
    displayMessage(message);
};
```

O Desafio das Linguagens Interpretadas i

Ao contrário de C, C++ ou Rust, o JavaScript é uma linguagem **Interpretada** (ou compilada JIT).

Linguagens Compiladas (C/C++):

- O compilador analisa todo o código **antes** da execução.
- Erros de sintaxe e incompatibilidade de tipos são apanhados em **Tempo de Compilação**.
- *Resultado:* Não pode executar o programa até que estes erros sejam corrigidos.

Linguagens Interpretadas (JavaScript):

O Desafio das Linguagens Interpretadas ii

- O browser lê e executa o código linha-a-linha (ou bloco-a-bloco) em **Tempo de Execução (Runtime)**.
- *Resultado:* A aplicação pode carregar perfeitamente e correr durante minutos.
- **O Crash:** O erro ocorre apenas quando o fluxo de execução atinge a linha específica com bug (ex: quando um utilizador clica num botão específico).

Consequência: "Funciona na minha máquina" é comum. Pode não encontrar o erro porque não ativou o caminho de execução específico que contém o bug.

A Lacuna de Ambiente: Editor vs. Browser i

A depuração de Aplicações Web introduz uma desconexão entre onde **escreve** o código e onde **executa** o código.

1. **A Mudança de Contexto (Context Switch):**
   - Escreve código num **IDE** (VS Code), que tem análise estática e linting.
   - Executa código no **Browser** (Chrome/Firefox).
   - Quando ocorre um erro, ele aparece na Consola do Browser, não imediatamente no seu editor de texto.
2. **O Problema da "Caixa Negra":**

A Lacuna de Ambiente: Editor vs. Browser ii

- O browser executa frequentemente código “minificado” ou “agrupado” (bundled) (para poupar largura de banda).
- Um erro na linha 1 do `bundle.js` é inútil para o programador.
- *Solução:* Confiamos em **Source Maps**, que dizem ao browser como mapear o código em execução de volta aos seus ficheiros originais.

Estratégias de Depuração

1. **Depuração "Printf" (`console.log`)**
   - O método mais antigo. Imprime variáveis na consola do browser para inspecionar o estado.
   - *Prós:* Rápido, simples.
   - *Contras:* Atravanca o código, requer limpeza, não pausa a execução.
2. **A palavra-chave `debugger;`**
   - Colocar a instrução `debugger;` no seu código força o browser a **pausar a execução** (breakpoint) nessa linha.
   - Pode então percorrer o código linha-a-linha.
3. **Browser DevTools (O separador Sources)**
   - Browsers modernos (Chrome/Firefox) têm debuggers integrados que rivalizam com IDEs de desktop.

O Problema “Estado vs. Vista” i

Em apps complexas (ex: Facebook, Spotify), manter a UI (Vista) sincronizada com os dados (Estado) usando Vanilla JS é propenso a erros.

As Frameworks resolvem isto através de:

1. **Programação Declarativa:** Define *o que* a UI deve parecer para um determinado estado, e não *como* atualizá-la.
2. **Componentização:** Dividir a UI em pedaços reutilizáveis e isolados.

React: A Biblioteca i

Desenvolvido pelo Facebook (Meta). O React é tecnicamente uma **Biblioteca**, não uma Framework, focada apenas na camada de Vista (View).

Conceitos Chave:

1. **Virtual DOM:** O React mantém uma cópia leve do DOM em memória. Quando o estado muda, calcula a "diferença" (diff) e atualiza apenas as partes alteradas do DOM real.
2. **JSX (JavaScript XML):** Extensão de sintaxe que permite escrever HTML dentro de JS.

React: A Biblioteca ii

3. **Fluxo de Dados Unidirecional:** Os dados fluem para baixo (Pai -> Filho).

Exemplo React i

Note a natureza **Declarativa**. Não chamamos `appendChild`. Retornamos a estrutura que queremos.

```
import React, { useState } from 'react';
function ImageGallery() {
  // State Hook: Quando 'images' muda, a UI auto-atuali
  const [images, setImages] = useState([
    { id: 1, url: 'img1.jpg' }
  ]);
  return (
    <div id="gallery">
      {/* Ciclo dentro de JSX */}
      {images.map(img ⇒ (
```

Exemplo React ii

```
          <img key={img.id} src={img.url}
          className="thumbnail" />
        ))}
      </div>
    );
  }
```

Angular: A Framework i

Desenvolvido pela Google. O Angular é uma **Framework** completa. Inclui routing, clientes HTTP e gestão de formulários "out of the box".

Conceitos Chave:

1. **TypeScript:** Obrigatório. Adiciona tipagem estática (Interfaces, Classes) ao JS para segurança.
2. **Injeção de Dependência (DI):** Sistema integrado para gerir serviços e estado.
3. **Ligação de Dados Bidirecional (Two-Way Data Binding):** Alterações na UI atualizam o Estado; Alterações no Estado atualizam a UI (automaticamente).

Angular: A Framework ii

4. **Real DOM:** O Angular opera diretamente no DOM mas usa um mecanismo sofisticado de Deteção de Mudanças (Zones).

Exemplo Angular i

O Angular separa a Lógica (Typescript) da Vista (Template HTML).

Lógica do Componente (`gallery.component.ts`)

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-gallery',
  templateUrl: './gallery.component.html'
})
export class GalleryComponent {
  // Array Tipado
  images: Array<{url: string}> = [{ url: 'img1.jpg' }];
}
```

Exemplo Angular ii

Template (`gallery.component.html`)

```
<div id="gallery">
  <img *ngFor="let img of images"
       [src]="img.url"
       class="thumbnail">
</div>
```

Comparação Resumida i

Característica	**Vanilla JS**	**React**	**Angular**
Paradigma	Imperativo	Declarativo	Declarativo
Linguagem	JavaScript	JS + JSX	TypeScript
DOM	Acesso Direto	Virtual DOM	Real DOM + Zones
Escala	Scripts pequenos	Apps Médias/Grandes	Apps Empresariais
Curva de Aprendizagem	Baixa	Média	Alta

Node.js & NPM i

Node.js não é uma linguagem; é um **Ambiente de Execução (Runtime)**. Pega no Motor V8 do Chrome e adiciona bindings C++ para Sistema de Ficheiros (FS) e Redes, permitindo que o JS corra em servidores.

NPM (Node Package Manager):

- Gere dependências (bibliotecas).
- `package.json`: O manifesto do projeto. Lista quais as bibliotecas necessárias (`dependencies`) e como correr o projeto (`scripts`).

Servidor Express Simples i

Express é a framework padrão para Node. Simplifica o routing.

```
// Importar biblioteca express
const express = require('express');
const cors = require('cors'); // Middleware para Segura
const app = express();
// Ativar CORS: Permite que o nosso JS baseado no brows
// vá buscar dados a este servidor. Sem isto, o browser
app.use(cors());
// Definir uma Rota (Endpoint)
app.get('/api/hello', (req, res) ⇒ {
    // Enviar resposta JSON
```

Servidor Express Simples ii

```
    res.json({
        msg: "Olá Mundo",
        serverTime: Date.now()
    });
});
app.listen(3000, () ⇒ console.log("A correr na porta 3
```

Python para Serviços Web i

Enquanto o Node.js partilha uma linguagem com o frontend, o **Python** é dominante em Ciência de Dados e IA.

Características FastAPI:

1. **Assíncrono:** Usa `async def` do Python (padrão ASGI), tornando-o muito mais rápido que Flask/Django.
2. **Dicas de Tipo (Type Hints):** Valida dados automaticamente.
3. **Swagger UI:** Gera um website de documentação (`/docs`) para a sua API automaticamente.

Exemplo FastAPI i

```python
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

# Configuração CORS
# Permitir explicitamente o contentor/origem do fronten
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"], # Em prod, substituir * pelo d
    allow_methods=["*"],
)
```

Exemplo FastAPI ii

```python
@app.get("/api/items")
async def read_items():
    # Dicionário Python é convertido automaticamente pa
    return [
        {"name": "Item 1", "price": 10.5},
        {"name": "Item 2", "price": 20.0}
    ]
```

A Arquitetura i

Temos duas aplicações separadas:

1. **Frontend:** HTML/JS estático servido pelo Nginx (ou uma app React/Angular compilada).
2. **Backend:** API Python/Node a processar dados.

Precisamos de as correr juntas e garantir que conseguem comunicar.

Configuração do Docker Compose i

`docker-compose.yml` orquestra aplicações multi-contentor.

```yaml
services:
  # --- O BACKEND ---
  backend-api:
    build: ./backend_folder          # Construir imagem a
    container_name: py_api
    ports:
      - "8000:8000"                  # Expor porta 8000 pa
    volumes:
      - ./backend_folder:/app        # Hot-reload de alter
```

Configuração do Docker Compose ii

```
  # --- O FRONTEND ---
  frontend-web:
    image: nginx:alpine              # Usar Nginx pré-cons
    container_name: my_website
    ports:
      - "8080:80"                    # Browser acede a loc
    volumes:
      # Injetar o nosso HTML/JS (ou build React) no Ngi
      - ./frontend_folder:/usr/share/nginx/html
    depends_on:
      - backend-api                  # Esperar que a API i
```

Conceito Crítico de Redes:

Configuração do Docker Compose iii

- **Browser para Backend:** Quando o seu JavaScript corre no *browser*, está a correr na *Máquina do Utilizador*. Portanto, o URL do `fetch` JS deve apontar para `http://localhost:8000` (a porta exposta pelo Docker para a máquina anfitriã), não para o nome interno do contentor.

Recursos Adicionais i

JavaScript & A Web

- MDN Web Docs (Mozilla) - A bíblia do desenvolvimento web.
- JavaScript.info - Mergulho profundo na linguagem moderna.
- What the heck is the event loop anyway? (Philip Roberts) - Visualização essencial do runtime de JS.

Frameworks

- React Documentation - Documentação oficial (reescrita recentemente).
- Angular University - Tutoriais abrangentes para Angular.

Recursos Adicionais ii

Backend & DevOps

- Node.js Best Practices - Padrões de arquitetura.
- FastAPI User Guide - Documentação excelente com exemplos interativos.
- Docker Curriculum - Um guia prático para iniciantes.