KORU


DESENVOLVE
programa de formação
e inclusão de talentos
em tecnologia
Grupo
Boticário
KORU

PROMISES E FETCH

O QUE VAMOS APRENDER HOJE?

- **Revisão:** Código Síncrono vs. Assíncrono
- **Callbacks**: A forma "clássica" (e seus problemas)
- **Promises**: A salvação! ✨ (Estados, `.then`, `.catch`, `.finally`)
- **Múltiplas Promises**: `Promise.all` e `Promise.race`
- **API `fetch`**: Conversando com servidores (GET, POST)
- **Tratando Respostas**: O objeto `Response` (`.ok`, `.status`, `.json()`, `.text()`)
- **`async/await`**: Deixando tudo mais legível
- **Tratamento de Erros Moderno**: `try...catch`

SÍNCRONO VS ASSÍNCRONO

SÍNCRONO VS. ASSÍNCRONO

Síncrono (Sync):

Uma tarefa depois da outra.

Se uma tarefa demorar (ex: ler um arquivo grande), o programa trava esperando.

Pense: Uma ligação telefônica 📞 (você espera a outra pessoa atender e conversar).

Assíncrono (Async):

Inicia uma tarefa e não espera ela terminar para começar a próxima.

Quando a tarefa demorada termina, ela "avisa" (callback, promise).

Ideal para I/O (Input/Output): Rede, Disco, Timers.

Pense: Enviar uma mensagem de texto 📱 (você envia e continua sua vida, recebe a resposta depois).

JavaScript no Browser e Node.js é fortemente assíncrono!

CALLBACKS: A FORMA CLÁSSICA DE ASSINCRONICIDADE

Uma **callback** é uma **função passada como argumento para outra função**, para ser executada depois que alguma operação assíncrona for concluída.

Problema: O que acontece se precisarmos de várias operações assíncronas em sequência? 🤔 A solução - nada elegante - chamamos de *callback hell*.

```
console.log("Início");

setTimeout(function minhaCallback() {
  console.log("Executou depois de 2 segundos!");
}, 2000); // 2000 milissegundos = 2 segundos

console.log("Fim (será logado ANTES da callback)");
```


PROMISES

PROMISES

Promise

É um **objeto** que representa a eventual conclusão (ou falha) de uma operação assíncrona e seu valor resultante

Uma "promessa" de que algo será entregue no futuro (ou uma explicação do porquê não foi).

Vantagens sobre Callbacks:

- Melhor legibilidade (encadeamento).
- Melhor controle de fluxo.
- Tratamento de erros centralizado.
- Componibilidade (combinar várias promises).

PROMISES

Estados de uma Promise

Uma Promise sempre começa em um estado e transita para outro:

Pending (Pendente):

Estado inicial. A operação ainda não foi concluída.

Esperando a pizza chegar... 🍕⏳

Fulfilled (Realizada / Resolvida):

A operação foi concluída com sucesso. A Promise tem um valor.

A pizza chegou! 🍕✅

Rejected (Rejeitada):

A operação falhou. A Promise tem um motivo (erro).

O entregador não achou o endereço... 😭❌

PROMISES

Consumindo Promises (.then)

Usamos o método .then() para registrar o que fazer quando a Promise for Fulfilled (Realizada).

```
const myPromise = new Promise((resolve, reject) => {
  // Simulando uma operação assíncrona
  setTimeout(() => {
    const success = true; // Tente mudar para false!
    if (success) {
      resolve("Dados recebidos com sucesso!"); // Valor da Promise
    } else {
      reject("Ocorreu um erro!"); // Motivo da rejeição
    }
  }, 1500);
});

console.log("Promise iniciada (pending)...");

myPromise.then((result) => {
  // Esta função SÓ executa se a promise for RESOLVIDA
  console.log("Promise resolvida:", result);
});

// O .then() também retorna uma Promise
```

PROMISES

Lidando com Erros (.catch)

Usamos o método .catch() para registrar o que fazer quando a Promise for Rejected (Rejeitada).

```
myPromise
  .then((resultado) => {
    console.log("Promise resolvida:", resultado);
    // return algo; // O valor retornado aqui vai para o próximo .then()
  })
  .catch((erro) => {
    // Esta função SÓ executa se a promise for REJEITADA
    // (ou se ocorrer um erro dentro de um .then() anterior)
    console.error("Promise rejeitada:", erro);
  });

console.log("Callback do .then/.catch registrado.");
```

PROMISES

Quando a promise termina:

Usamos o método **.finally()** para registrar uma função que será executada independentemente se a Promise foi resolvida ou rejeitada.

Útil para "limpeza" (ex: esconder um loading spinner, fechar uma conexão).

.finally() não recebe argumentos e não interfere no valor/erro da Promise.

```
myPromise
  .then((resultado) => {
    console.log("Sucesso:", resultado);
  })
  .catch((erro) => {
    console.error("Erro:", erro);
  })
  .finally(() => {
    // Executa SEMPRE (após .then ou .catch)
    console.log("Operação finalizada (com sucesso ou erro).");
    // Ex: document.getElementById('loading').style.display = 'none';
  });
```

PROMISES

Encadeando Promises

Como **.then()** (e .**catch()**) retornam também Promises, podemos encadeá-los, criando um fluxo assíncrono muito mais legível.

Se um .then() retorna um valor, a próxima Promise na cadeia é resolvida com esse valor.

Se um erro ocorre em qualquer ponto, a execução "pula" para o próximo .catch() na cadeia.

```
fetch('https://api.example.com/users/1') // Retorna Promise
.then(response => response.json())      // Retorna Promise
.then(userData => {                     // Recebe UserData
  console.log("Usuário:", userData.name);
  // Podemos retornar outra promise aqui!
  return fetch(`https://api.example.com/posts?userId=${userData.id}`); // Promise
})
.then(response => response.json())      // Retorna Promise
.then(posts => {                        // Recebe Array
  console.log(`Posts do usuário: ${posts.length}`);
})
.catch(error => {                       // Pega QUALQUER erro na cadeia
  console.error("Algo deu errado:", error);
});
```

PROMISES PARALELAS

Promise.all()

Recebe um array de Promises e retorna uma única Promise.

Essa Promise só será *fulfilled* se todas as Promises no array forem fulfilled. O resultado será um array com os valores resolvidos, na mesma ordem do array original.

Essa Promise será *rejected* assim que a primeira Promise no array for rejected. O motivo será o erro da primeira Promise rejeitada.

Quando usar? Quando você precisa que várias tarefas assíncronas independentes terminem antes de continuar.

```
const p1 = delay(1000).then(() => 'Resultado 1');
// Termina antes, mas all espera p1
const p2 = delay(500).then(() => 'Resultado 2');
const p3 = delay(1500).then(() => 'Resultado 3');

Promise.all([p1, p2, p3])
  .then(results => { // results = ['Resultado 1', 'Resultado 2', 'Resultado 3']
    console.log('Todas terminaram:', results);
  })
  .catch(err => {
    console.error('Alguma deu erro:', err);
  });
```

PROMISES PARALELAS

Promise.race()

Recebe um array de Promises e
Retorna uma única Promise.

Essa Promise será resolvida (seja fulfilled ou rejected) assim que a primeira Promise no array for resolvida.

O resultado será o da primeira Promise a resolver.

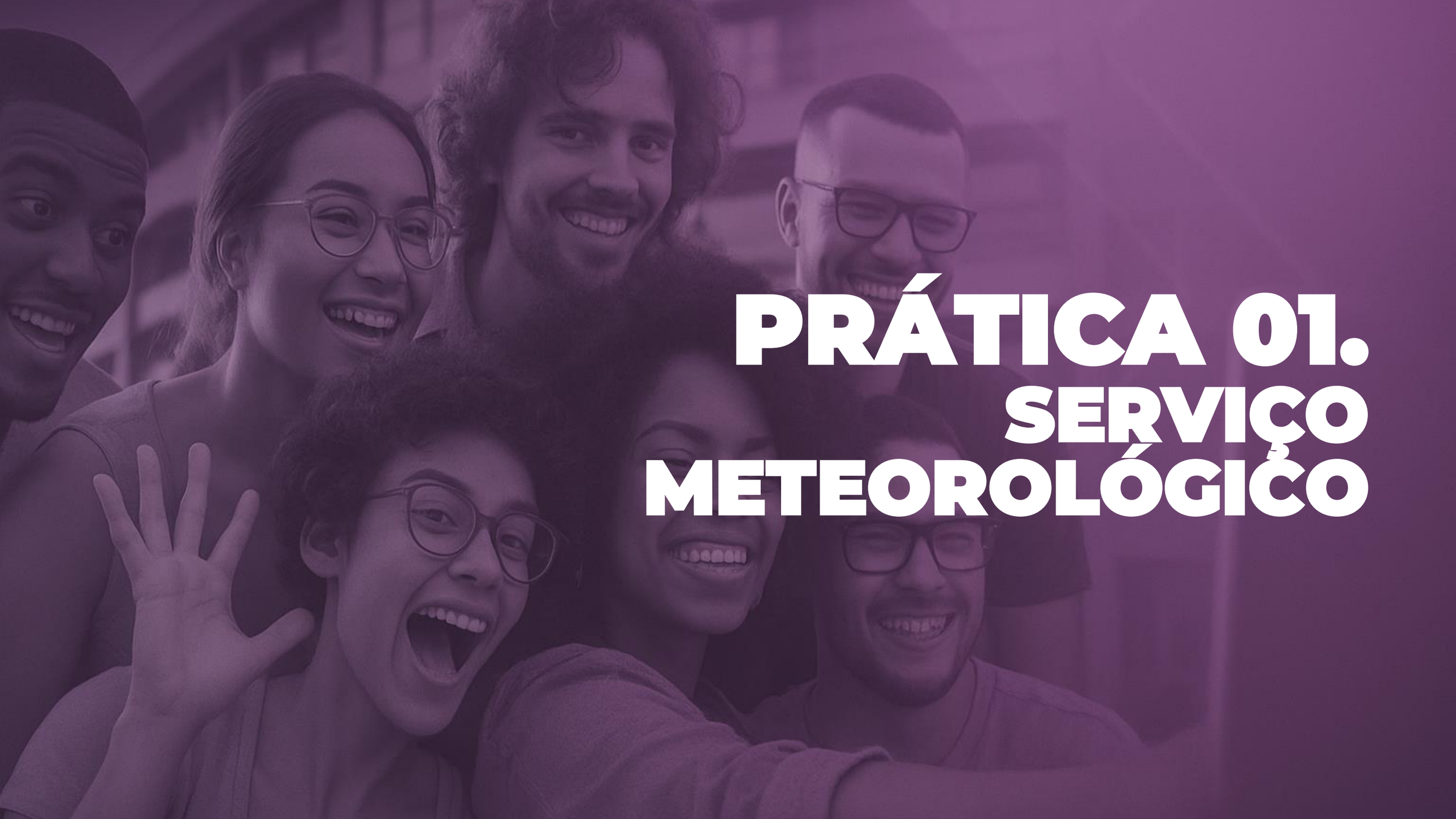
Promise.any()

Igual à Promise.race(), mas apenas resolve com a primeira que for bem-sucedida (fulfilled)

Só rejeita se todas falharem.

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(reject, 100, 'Erro 1');
});

const promise2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 200, 'Sucesso 2');
});

const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 300, 'Sucesso 3');
});

Promise.any([promise1, promise2, promise3])
  .then((value) => {
    console.log('Primeira promessa resolvida:', value);
  })
  .catch((error) => {
    console.log('Todas as promessas falharam:', error);
  });
```

```
const pA = delay(2000).then(() => 'A');
const pB = delay(500).then(() => 'B'); // Esta ganha a corrida
const pC = delay(1000).then(() => 'C');

Promise.race([pA, pB, pC])
  .then((result) => {
    // result = 'B'
    console.log('Primeira a terminar:', result);
  })
  .catch((err) => {
    // Se a primeira a terminar for rejeitada
    console.error('A primeira deu erro:', err);
  });
```



PRÁTICA 01.
SERVIÇO METEOROLÓGICO

PRÁTICA

Objetivo

Temos 3 serviços hipotéticos de meteorologia que busca a temperatura de uma determinada cidade.

Implemente uma função chamada **getTemperature()**
Faça isso da melhor forma em termos de **performance** e **confiabilidade** do sistema.

FETCH API CONVERSANDO COM O MUNDO

FETCH API

Interface moderna para fazer requisições de rede assíncronas.

Disponível nativamente no Navegador e no Node (v 18+).

Retorna uma Promise que resolve para o **objeto Response**.

Recebe a url e opções como argumentos

Retorna uma Promise que resolve em um **objeto Response**.

O Objeto Response é uma API Web

```
// url
const url = 'https://developer.mozilla.org';

// Sintaxe básica (GET)
fetch(url);

// Sintaxe com options object
fetch(url, {
  method: 'POST', // ou 'GET', 'PUT', 'DELETE', etc.
  headers: {
    'Content-Type': 'application/json',
    Authorization: 'Bearer token123',
  },
  body: JSON.stringify({ key: 'value' }), // Para POST/PUT
});
```

(https://developer.mozilla.org/en-US/docs/Web/API/Response)

FETCH API - MÉTODOS HTTP

Métodos HTTP

Quando fazemos uma requisição para um servidor (com fetch), precisamos dizer a ele qual tipo de ação queremos realizar.

Isso é feito usando Métodos HTTP (ou Verbos HTTP).

Principais Métodos (Comuns em APIs REST):

GET: Solicita dados de um recurso específico. Não deve ter "efeitos colaterais" no servidor (apenas leitura). (Ex: buscar um post, listar usuários).

POST: Envia dados para criar um novo recurso no servidor. (Ex: criar um novo post, cadastrar um usuário).

PUT: Envia dados para atualizar/substituir um recurso existente. (Ex: editar um post existente).

DELETE: Solicita a remoção de um recurso específico. (Ex: deletar um post).

FETCH API - O OBJETO RESPONSE

Objeto Response

A Promise retornada por fetch resolve para este objeto que contém informações sobre a resposta da requisição.

Propriedades Importantes:

response.status: Código de status HTTP (ex: 200, 404, 500).

response.ok: Booleano (true para status 2xx, false caso contrário).

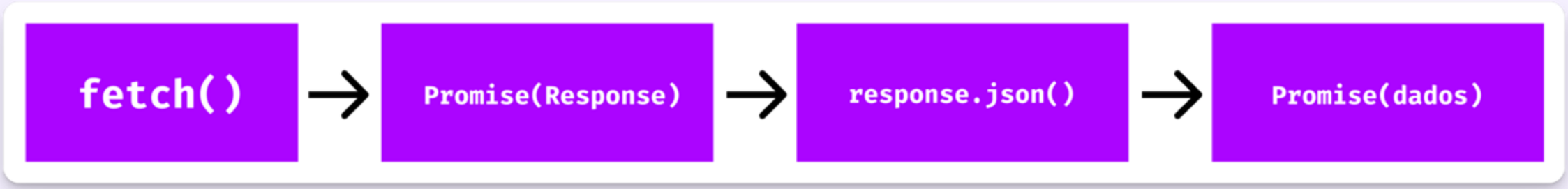
response.headers: Cabeçalhos da resposta.

Métodos para Obter o Corpo da Resposta (Retornam Promises):

response.json(): Processa a resposta como JSON.

response.text(): Processa a resposta como texto puro.

FETCH API - O OBJETO RESPONSE



PRÁTICA 02.
ENDEREÇO PELO CEP

PRÁTICA

Objetivo

Vamos usar o serviço da viaCEP para buscar um endereço baseado no CEP. Crie uma função que receba como argumento o cep e retorne um objeto com logradouro, cidade e estado.

ASYNC / AWAIT

ASYNC AWAIT

Introduzidos no ES2017.

Torna o código mais fácil de ler e entender, especialmente com múltiplas operações assíncronas.

await pausa a execução da função async até que a Promise à sua direita seja resolvida (fulfilled ou rejected).

Quando a Promise é fulfilled, await retorna o valor resolvido.

Quando a Promise é rejected, await lança um erro.

Async / Await nada mais é do que um "syntatic sugar".

PRÁTICA 03.
ENDEREÇO PELO CEP COM ASYNC AWAIT

PRÁTICA

Objetivo

Reescreva a solução da prática anterior (endereço através do CEP), mas usando apenas a sintaxe do async/await.


TRY...CATCH

TRY...CATCH

Como **await** lança um erro quando a Promise é rejeitada, podemos usar a estrutura try...catch.

try { ... código que pode dar erro ... }

catch(error) { ... código para lidar com o erro ... }

Benefício: Permite centralizar o tratamento de erros de múltiplas operações await em um único bloco catch.

PRÁTICA 03. ENDEREÇO PELO CEP COM ASYNC AWAIT + TRY...CATCH PARA TRATAMENTO DE ERRO

RESUMO

RESUMO

- **Javascript é síncrono** por padrão, mas precisa lidar com **tarefas assíncronas** (rede, tempo, I/O).
- **Callbacks foram a solução inicial**, mas levam ao "Callback Hell".
- **Promises** são objetos que representam um resultado futuro e facilitam o encadeamento (**.then()**) e tratamento de erros (**.catch(), .finally()**).
- **Promise.all(), Promise.race()** e **Promise.any()** para múltiplos Promises.
- A **API fetch** é a forma moderna de fazer requisições **HTTP**.
- **async/await** é uma sintaxe mais limpa para trabalhar com Promises.
- Use **try...catch** com async/await para um tratamento de erro elegante.

KORU


DESENVOLVE
programa de formação
e inclusão de talentos
em tecnologia
Grupo
Boticário
KORU