Aula 20 – JavaScript Moderno (ES6+): Assincronismo (Parte 2)

Dominando a Arte da Programação Não-Bloqueante

No mundo acelerado do desenvolvimento web, onde cada milissegundo conta para a experiência do usuário, o JavaScript se tornou o maestro de orquestras complexas. Imagine um site que precisa buscar dados de um servidor, carregar imagens e ainda permitir que o usuário interaja sem travamentos. Parece mágica, não é? Na verdade, é o poder do **assincronismo** em ação, uma das características mais cruciais e, por vezes, desafiadoras do JavaScript moderno.

Esta aula é a sua bússola para navegar pelas águas do assincronismo, desvendando como o JavaScript lida com tarefas que levam tempo sem congelar a interface. Você já deve ter se deparado com situações onde um clique não respondia imediatamente, ou uma página demorava a carregar informações. Compreender o assincronismo não é apenas uma habilidade técnica; é a chave para construir aplicações web fluidas, responsivas e que encantam o usuário, um pilar fundamental para a **performance web** e a **acessibilidade**.

Identificar Gargalos

Reconhecer operações síncronas que prejudicam a performance

Aplicar Padrões

Dominar Callbacks, Promises e async/await

Código Eficiente

Escrever código limpo e otimizado para aplicações modernas

Prepare-se para elevar o nível das suas aplicações, garantindo que elas funcionem de forma otimizada, mesmo sob as demandas mais exigentes do cenário atual, onde ferramentas como **Vite** otimizam o desenvolvimento e a performance é medida por **Core Web Vitals**.

Desvendando o Event Loop

Você já parou para pensar como um navegador consegue exibir uma página, tocar uma animação e, ao mesmo tempo, buscar dados de um servidor, tudo sem engasgar? A resposta está no coração do JavaScript: seu modelo de execução assíncrono e o famoso **Event Loop**. Por natureza, o JavaScript é uma linguagem de thread única, o que significa que ele executa uma tarefa por vez. Se uma tarefa demorasse muito, a interface congelaria, e o usuário ficaria frustrado.

É aqui que o **assincronismo** entra em cena, permitindo que operações demoradas (como requisições de rede, leitura de arquivos ou temporizadores) sejam iniciadas e, em vez de bloquear a execução principal, "deleguem" a espera para outro lugar. Pense nisso como um chef de cozinha que recebe um pedido de um prato que leva tempo para cozinhar. Em vez de parar tudo e esperar o prato ficar pronto, ele coloca o prato no forno e continua a preparar outras coisas, como saladas ou sobremesas. Quando o forno apita, ele volta para finalizar o prato principal.

> 💡 **O Segredo da Fluidez**
>
> O Event Loop é o mecanismo que permite ao JavaScript ser **não-bloqueante**, essencial para a fluidez das aplicações modernas.

Essa "delegação" é gerenciada pelo **Event Loop**, um mecanismo fascinante que orquestra a execução de código. Ele monitora a **Call Stack** (onde as funções são executadas) e a **Callback Queue** (onde as tarefas assíncronas "delegadas" esperam para serem executadas). Quando a Call Stack está vazia, o Event Loop pega a primeira tarefa da Callback Queue e a move para a Call Stack. É um ciclo contínuo que garante que o JavaScript seja não bloqueante, essencial para a fluidez das aplicações modernas.

Event Loop em Detalhes: Microtasks vs Macrotasks

Para entender completamente como o JavaScript mantém a interface responsiva, é importante aprofundar um pouco mais no funcionamento do Event Loop e na distinção entre diferentes tipos de tarefas. Nem todas as tarefas assíncronas são tratadas da mesma forma; o Event Loop prioriza algumas sobre outras, garantindo que as interações do usuário e as atualizações visuais sejam processadas de maneira eficiente.

Existem duas categorias principais de filas de tarefas que o Event Loop gerencia: as **Macrotasks** (também conhecidas como tasks) e as **Microtasks**. As Macrotasks incluem eventos como setTimeout(), setInterval(), I/O (entrada/saída) e eventos de UI. As Microtasks, por outro lado, são tarefas de prioridade mais alta e incluem as resoluções de **Promises** e queueMicrotask(). O Event Loop processa todas as Microtasks na fila antes de passar para a próxima Macrotask.

Macrotasks

- setTimeout()
- setInterval()
- I/O operations
- Eventos de UI

Microtasks

- Promise callbacks
- queueMicrotask()
- MutationObserver
- process.nextTick() (Node.js)

> Essa priorização é crucial. Imagine que você está em um café (o navegador) e faz um pedido (uma Macrotask). Enquanto o barista prepara seu café, ele pode rapidamente limpar o balcão ou atender a um pedido rápido de um cliente que já está com o café pronto (Microtasks). Somente depois de todas essas pequenas tarefas urgentes, ele se dedica a fazer o próximo café completo.

No JavaScript, isso significa que uma Promise resolvida (Microtask) será processada antes de um setTimeout (Macrotask) agendado para o mesmo momento, garantindo que a lógica de dados seja atualizada rapidamente. Essa orquestração inteligente é o que permite que aplicações modernas, construídas com ferramentas como **Vite**, mantenham alta performance e responsividade, elementos-chave para uma boa experiência de usuário e para atender aos **Core Web Vitals**.

Callbacks: A Primeira Tentativa

Antes das soluções mais elegantes que temos hoje, os **Callbacks** eram a principal ferramenta para lidar com o assincronismo no JavaScript. Um Callback é simplesmente uma função que é passada como argumento para outra função e é executada depois que a primeira função termina sua operação. Imagine que você pede uma pizza e diz ao entregador: "Quando a pizza chegar, me ligue para eu descer e pegar". A função de "ligar para você" é o Callback, que só será executada após a "entrega da pizza" ser concluída.

01

Função Principal

Inicia a operação assíncrona

02

Operação em Andamento

Tarefa sendo processada em background

03

Callback Executado

Função é chamada quando a operação termina

No contexto do JavaScript, isso funciona bem para uma ou duas operações assíncronas. Por exemplo, ao carregar um script ou fazer uma requisição simples, você pode passar uma função para ser executada quando a operação terminar. Essa abordagem é direta e fácil de entender para casos isolados, sendo a base para muitos eventos e operações de temporização.

```
// Exemplo de Callback simples
function buscarDados(callback) {
  console.log("Iniciando busca de dados...");
  setTimeout(() => {
    const dados = "Dados do servidor";
    console.log("Dados recebidos!");
    callback(dados);
  }, 2000);
}

buscarDados((resultado) => {
  console.log("Resultado:", resultado);
});
```

⚠️ O Problema do Callback Hell

Quando você precisa encadear várias operações assíncronas, o código se torna difícil de ler, manter e depurar, formando uma estrutura indentada que lembra uma pirâmide.

O problema, carinhosamente conhecido como **"Callback Hell"** (ou "Pirâmide da Perdição"), acontece quando você precisa encadear várias operações assíncronas, onde o resultado de uma depende da anterior. Isso compromete a legibilidade e a escalabilidade do projeto, um desafio real para equipes que buscam desenvolver aplicações robustas e acessíveis.

O Pesadelo do Callback Hell

Para ilustrar o "Callback Hell", vamos imaginar um cenário comum em uma aplicação web: você precisa autenticar um usuário, depois buscar o perfil desse usuário, e então carregar as postagens dele. Cada uma dessas operações é assíncrona, pois envolve comunicação com um servidor. Se usarmos apenas Callbacks, o código rapidamente se torna um labirinto, dificultando a compreensão do fluxo lógico e a identificação de possíveis erros.

```
// Exemplo de Callback Hell em ação
autenticarUsuario("aluno", "123",
  (token) => {
    console.log("Autenticado. Token:", token);
    buscarPerfil(token,
      (perfil) => {
        console.log("Perfil:", perfil);
        carregarPostagens(perfil.id,
          (postagens) => {
            console.log("Postagens:", postagens);
            // E se precisarmos de mais uma operação?
          },
          (erroPostagens) => {
            console.error("Erro postagens:", erroPostagens);
          }
        );
      },
      (erroPerfil) => {
        console.error("Erro perfil:", erroPerfil);
      }
    );
  },
  (erroAuth) => {
    console.error("Erro auth:", erroAuth);
  }
);
```

Indentação Profunda

Cada nova operação adiciona um nível de aninhamento

Tratamento de Erros Duplicado

Cada callback precisa de seu próprio bloco de erro

Difícil Manutenção

Modificar ou adicionar operações se torna complexo

Baixa Legibilidade

O fluxo lógico fica obscurecido pela estrutura

Percebe como o código se aninha e se torna cada vez mais difícil de seguir? A cada nova operação assíncrona, adicionamos mais um nível de indentação e mais um bloco de tratamento de erros. É como tentar desvendar um novelo de lã completamente emaranhado; cada puxada pode piorar a situação. Essa complexidade impacta diretamente a produtividade do desenvolvedor e a qualidade do software. Em projetos que utilizam ferramentas modernas como **Vite**, um código com "Callback Hell" pode se tornar um gargalo, minando os benefícios de um ambiente de desenvolvimento rápido.

Promises: A Solução Elegante

Diante dos desafios do "Callback Hell", o JavaScript introduziu as **Promises** (Promessas) no ES6 (ECMAScript 2015) como uma solução mais estruturada e legível para lidar com operações assíncronas. Uma Promise é um objeto que representa a eventual conclusão (ou falha) de uma operação assíncrona e seu valor resultante. Pense em uma Promise como um bilhete de loteria: você compra o bilhete (inicia a operação assíncrona), mas o resultado (ganhar ou perder) só será conhecido no futuro.

Pending

Estado inicial - operação em andamento

Fulfilled

Operação concluída com sucesso

Rejected

Operação falhou com erro

> A grande vantagem das Promises é a capacidade de encadear operações assíncronas de forma sequencial e legível, usando os métodos `.then()` para lidar com o sucesso e `.catch()` para lidar com erros.

Isso elimina a necessidade de aninhamento profundo, transformando a "pirâmide" em uma "escada" de operações. Além disso, uma Promise pode ser retornada por uma função e manipulada em outro lugar, promovendo a modularidade do código e facilitando a reutilização.

Criando e Consumindo Promises

Para criar uma Promise, você instancia o objeto Promise e passa uma função executora como argumento. Essa função executora recebe dois argumentos: `resolve` e `reject`, que são funções que você chama para mudar o estado da Promise para "fulfilled" ou "rejected", respectivamente.

Criando uma Promise

```
function buscarDados() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const sucesso = Math.random() > 0.5;
      if (sucesso) {
        resolve("Dados obtidos!");
      } else {
        reject("Erro na busca");
      }
    }, 2000);
  });
}
```

Consumindo a Promise

```
buscarDados()
  .then((resultado) => {
    console.log(resultado);
    return "Processado";
  })
  .then((novo) => {
    console.log(novo);
  })
  .catch((erro) => {
    console.error(erro);
  })
  .finally(() => {
    console.log("Finalizado");
});
```

1

.then()

Executado quando a Promise é resolvida com sucesso

2

.catch()

Captura erros de qualquer Promise anterior na cadeia

3

.finally()

Sempre executado, independente de sucesso ou falha

✨ Encadeamento Linear

Cada `.then()` retorna uma nova Promise, permitindo que você continue encadeando operações assíncronas de forma sequencial e com tratamento de erros centralizado.

Este encadeamento linear de `.then()` é o que permite transformar o "Callback Hell" em um fluxo mais claro e gerenciável, criando uma sequência lógica de operações que é fácil de ler e manter.

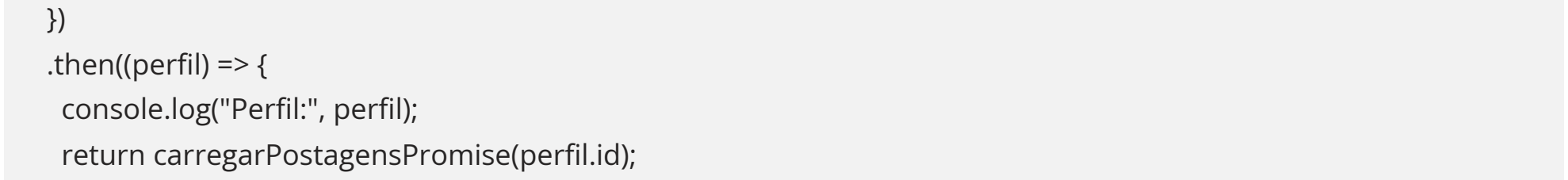
Promises em Ação: Encadeamento Prático

Com as Promises, o cenário de autenticação, busca de perfil e carregamento de postagens, que antes era um "Callback Hell", se transforma em um fluxo muito mais gerenciável. Cada função assíncrona agora retorna uma Promise, e podemos encadeá-las usando `.then()`, criando uma sequência lógica de operações.

```
// Funções que retornam Promises
function autenticarUsuarioPromise(usuario, senha) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (usuario === "aluno" && senha === "123") {
        resolve("token123");
      } else {
        reject("Credenciais inválidas.");
      }
    }, 500);
  });
}

function buscarPerfilPromise(token) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (token === "token123") {
        resolve({ id: 1, nome: "João", email: "joao@example.com" });
      } else {
        reject("Token inválido.");
      }
    }, 700);
  });
}


function carregarPostagensPromise(userId) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (userId === 1) {
        resolve(["Post 1", "Post 2", "Post 3"]);
      } else {
        reject("Usuário não encontrado.");
      }
    }, 1000);
  });
}
```

Encadeamento Linear e Elegante

```
autenticarUsuarioPromise("aluno", "123")
  .then((token) => {
    console.log("Autenticado. Token:", token);
    return buscarPerfilPromise(token);
  })
  .then((perfil) => {
    console.log("Perfil:", perfil);
    return carregarPostagensPromise(perfil.id);
  })
  .then((postagens) => {
    console.log("Postagens:", postagens);
    console.log("Fluxo completo concluído!");
  })
  .catch((erro) => {
    console.error("Erro no fluxo:", erro);
  })
  .finally(() => {
    console.log("Sessão finalizada.");
});
```



Este padrão de encadeamento é muito mais robusto e legível. O `.catch()` no final da cadeia atua como um único ponto de tratamento de erros para qualquer Promise que seja rejeitada em qualquer etapa. Isso simplifica a lógica de tratamento de exceções e melhora a manutenibilidade do código, um aspecto crucial para a acessibilidade e a performance de aplicações complexas.

async/await: Sintaxe Mais Limpa

Embora as Promises tenham resolvido o "Callback Hell", a sintaxe de `.then()` e `.catch()` ainda pode parecer um pouco verbosa para alguns, especialmente quando se lida com muitas operações sequenciais. Para tornar o código assíncrono ainda mais parecido com o código síncrono, o JavaScript introduziu as palavras-chave **async** e **await** no ES2017. Elas são, na verdade, um "açúcar sintático" sobre as Promises, tornando o trabalho com elas incrivelmente mais limpo e intuitivo.

async

Declara uma função como assíncrona. Uma função `async` sempre retorna uma Promise.

```
async function minhaFuncao() {
  return "valor";
}
// Equivalente a:
// return Promise.resolve("valor")
```

await

Pausa a execução até que a Promise seja resolvida ou rejeitada. Só pode ser usado dentro de funções `async`.

```
async function buscar() {
  const dados = await fetch(url);
  console.log(dados);
}
// Espera a Promise resolver
```

> Pense no `await` como uma pausa estratégica. É como se você estivesse lendo um livro e chegasse a uma frase que diz "Espere aqui até que o mensageiro traga a próxima página". Você para, espera o mensageiro, pega a página e continua lendo. O resto do mundo (o Event Loop) continua funcionando, mas sua leitura (a função async) está pausada até a informação chegar.

Legibilidade

Código assíncrono que parece síncrono

Tratamento de Erros

Use try...catch familiar

Produtividade

Escreva e mantenha código mais rápido

async/await: Tratamento de Erros

A beleza do async/await reside não apenas na sua sintaxe mais limpa para encadear operações, mas também na forma como ele simplifica o tratamento de erros. Diferente do `.catch()` das Promises, que é um método encadeado, com async/await podemos usar o bloco `try...catch` que já conhecemos do código síncrono. Isso torna a lógica de tratamento de exceções muito mais familiar e intuitiva.

```
async function buscarDadosComErro() {
  console.log("Iniciando busca...");

  try {
    const dados = await new Promise((resolve, reject) => {
      setTimeout(() => {
        const sucesso = false; // Força falha
        if (sucesso) {
          resolve("Dados obtidos!");
        } else {
          reject("Erro: Falha na conexão.");
        }
      }, 1500);
    });

    console.log(dados);

  } catch (erro) {
    console.error("Erro capturado:", erro);

  } finally {
    console.log("Operação finalizada.");
  }
}

buscarDadosComErro();
```

try

Tenta executar o código assíncrono

1

2

await

Pausa até a Promise resolver

catch

Captura erros se a Promise for rejeitada

3

4

finally

Sempre executado ao final

🎯 Depuração Simplificada

O uso do try...catch com async/await facilita a depuração. Você pode definir pontos de interrupção (breakpoints) e seguir o fluxo do código assíncrono de forma mais linear, como se fosse síncrono.

Quando uma Promise aguardada por `await` é rejeitada, ela se comporta como se tivesse lançado uma exceção síncrona. Essa exceção pode ser capturada por um bloco `try...catch` que envolve a chamada await, permitindo que você trate erros específicos de cada operação ou um erro geral para um bloco de operações assíncronas.

async/await na Prática: Fluxo Completo

Vamos reescrever o exemplo de autenticação, perfil e postagens usando async/await. A clareza do código é notável, tornando-o quase indistinguível de um fluxo síncrono, mas mantendo todas as vantagens do assincronismo.

```
async function realizarFluxoUsuario() {
  console.log("Iniciando fluxo de usuário...");

  try {
    // Primeira operação: autenticar
    const token = await autenticarUsuarioPromise("aluno", "123");
    console.log("Autenticado. Token:", token);

    // Segunda operação: buscar perfil
    const perfil = await buscarPerfilPromise(token);
    console.log("Perfil:", perfil);

    // Terceira operação: carregar postagens
    const postagens = await carregarPostagensPromise(perfil.id);
    console.log("Postagens:", postagens);

    console.log("Fluxo completo concluído!");

  } catch (erro) {
    console.error("Erro no fluxo:", erro);

  } finally {
    console.log("Sessão finalizada.");
  }
}

realizarFluxoUsuario();
```

Autenticação

await pausa até obter o token

Perfil

await pausa até carregar o perfil

Postagens

await pausa até buscar as postagens

> Observe como o código agora se lê de cima para baixo, de forma linear. Cada `await` faz com que a execução da função pause até que a Promise seja resolvida, e então o valor resultante é atribuído à variável. Se qualquer uma das Promises for rejeitada, o controle é imediatamente transferido para o bloco `catch`.

A combinação de async/await com ferramentas modernas de desenvolvimento como **Vite**, que otimizam o processo de build e recarregamento, permite que os desenvolvedores criem experiências de usuário fluidas e responsivas com um código mais limpo e compreensível. É uma virada de jogo para a produtividade e a qualidade do software, impactando positivamente a **performance web** e a **acessibilidade**.

Escolhendo a Ferramenta Certa

Com três abordagens para lidar com o assincronismo, surge a pergunta: qual delas devo usar? A resposta depende do contexto, da complexidade da operação e da legibilidade que você deseja para o seu código. Embora os Callbacks tenham sido fundamentais, as Promises e, mais recentemente, o async/await, se tornaram os padrões preferenciais para o desenvolvimento moderno em JavaScript.

Conceito	Legibilidade	Tratamento de Erros	Uso Recomendado
Callbacks	⭐⭐	Múltiplos blocos	APIs antigas, eventos simples
Promises	⭐⭐⭐⭐	.catch() centralizado	Encadeamento, operações paralelas
async/await	⭐⭐⭐⭐⭐	try...catch familiar	Fluxos sequenciais, código moderno

Callbacks

Ainda encontrados em APIs antigas, mas levam ao "Callback Hell" em encadeamentos complexos

Promises

Estrutura robusta com encadeamento limpo, ideal para operações paralelas com Promise.all()

async/await

Ápice da legibilidade, código assíncrono que parece síncrono, recomendado para novos projetos

💡 Recomendação

Para a maioria dos novos projetos e para refatorar código existente, priorize o uso de **async/await** sempre que possível, pois ele oferece a melhor experiência de desenvolvimento em termos de clareza e manutenção.

Padrões Avançados: Promise.all() e Promise.race()

Dominar o assincronismo vai além de encadear operações sequenciais. Muitas vezes, em aplicações modernas, precisamos lidar com múltiplas operações assíncronas que podem ser executadas em paralelo ou que precisam competir entre si. As Promises oferecem métodos estáticos poderosos para gerenciar esses cenários de forma eficiente e legível.

Promise.all() - Execução Paralela

Imagine que você precisa carregar várias imagens para uma galeria ou buscar dados de diferentes APIs para compor uma única tela. Se essas operações não dependem uma da outra, executá-las sequencialmente seria um desperdício de tempo. É aqui que `Promise.all()` brilha.

```
const carregarImagem = (url) =>
  new Promise(resolve => {
    setTimeout(() => {
      console.log(\`Imagem ${url} carregada\`);
      resolve(\`Dados de ${url}\`);
    }, Math.random() * 2000 + 500);
  });

async function carregarGaleria() {
  console.log("Iniciando carregamento...");

  try {
    const resultados = await Promise.all([
      carregarImagem("foto1.jpg"),
      carregarImagem("foto2.jpg"),
      carregarImagem("foto3.jpg")
    ]);

    console.log("Todas carregadas:", resultados);

  } catch (erro) {
    console.error("Falha:", erro);
  }
}
```

Paralelo

Todas as operações iniciam ao mesmo tempo

Aguarda Todas

Resolve quando todas terminarem

Promise.race() - Competição

Agora, imagine um cenário onde você precisa obter um recurso de múltiplas fontes, mas só precisa do resultado da fonte que responder primeiro. `Promise.race()` é a ferramenta ideal para isso.

```
const fonteLenta = new Promise(resolve =>
  setTimeout(() => resolve("Fonte lenta"), 3000)
);

const fonteRapida = new Promise(resolve =>
  setTimeout(() => resolve("Fonte rápida"), 1000)
);

async function buscarMaisRapido() {
  console.log("Buscando da fonte mais rápida...");

  try {
    const resultado = await Promise.race([
      fonteLenta,
      fonteRapida
    ]);

    console.log("Primeiro resultado:", resultado);
    // Será "Fonte rápida"

  } catch (erro) {
    console.error("Falha:", erro);
  }
}
```

Corrida

Primeira a resolver vence

Resultado Único

Retorna apenas o primeiro

Esses métodos estáticos de Promise são ferramentas poderosas para otimizar o fluxo de trabalho assíncrono, permitindo que você escreva código mais eficiente e responsivo, o que é fundamental para atender aos requisitos de **Core Web Vitals** e proporcionar uma experiência de usuário superior.

Boas Práticas em Assincronismo

Escrever código assíncrono eficiente e robusto vai além de apenas saber usar Promises ou async/await. É sobre aplicar boas práticas que garantam a estabilidade, a performance e a manutenibilidade da sua aplicação. Em um ambiente de desenvolvimento moderno, onde ferramentas como **Vite** aceleram o processo, a qualidade do código assíncrono é um diferencial.

1. **Sempre Trate Erros**

   Use try...catch com async/await ou .catch() com Promises. Uma Promise rejeitada não capturada pode levar a erros não tratados e comportamentos inesperados.

2. **Evite Misturar Padrões**

   Mantenha consistência dentro de um módulo. Se está usando async/await, evite misturar com .then() desnecessariamente.

3. **Minimize Efeitos Colaterais**

   Isole a lógica assíncrona e evite modificar estado global. Torne seu código mais previsível e fácil de testar.

4. **Otimize Operações Paralelas**

   Use Promise.all() para operações independentes. Isso economiza tempo e melhora as métricas de Core Web Vitals.

5. **Pense em Acessibilidade**

   Forneça feedback visual durante operações assíncronas. Use spinners, desabilite botões e comunique atualizações com ARIA live regions.

Impacto na Performance

- **LCP (Largest Contentful Paint):** Operações paralelas reduzem tempo de carregamento
- **FID (First Input Delay):** Código não-bloqueante mantém interface responsiva
- **CLS (Cumulative Layout Shift):** Carregamento assíncrono bem gerenciado evita mudanças de layout

Impacto na Acessibilidade

- **Feedback Visual:** Indicadores de carregamento para todos os usuários
- **ARIA Live Regions:** Comunicar atualizações para leitores de tela
- **Estados de Botões:** Desabilitar durante operações para evitar cliques múltiplos

🎯 Qualidade em 2025

Ao seguir essas boas práticas, você não apenas escreverá código JavaScript mais eficaz, mas também contribuirá para a criação de aplicações web de alta qualidade que são rápidas, confiáveis e inclusivas.

Consolidação e Próximos Passos

Chegamos ao fim da nossa jornada pelo assincronismo moderno em JavaScript. Vimos como o **Event Loop** é o coração que permite ao JavaScript, uma linguagem single-threaded, lidar com operações demoradas sem bloquear a interface. Exploramos os desafios do **Callback Hell** e como as **Promises** surgiram como uma solução mais estruturada, permitindo o encadeamento de operações com .then(), .catch() e .finally(). Finalmente, mergulhamos no **async/await**, o "açúcar sintático" que torna o código assíncrono tão legível quanto o síncrono.

Event Loop Orquestra a execução não-bloqueante	**Promises** Estrutura robusta para assincronismo
async/await Sintaxe limpa e intuitiva	**Boas Práticas** Performance e acessibilidade

Em Prática

- [ ] **Sempre que iniciar uma operação assíncrona, pense em como o sucesso e a falha serão tratados**
- [ ] **Prefira async/await para operações sequenciais pela sua clareza e legibilidade**
- [ ] **Use Promise.all() para operações independentes que podem ser executadas em paralelo**
- [ ] **Monitore a performance de suas operações assíncronas, especialmente requisições de rede**
- [ ] **Forneça feedback visual ao usuário durante operações assíncronas para melhorar UX e A11Y**

Autoavaliação

Questões Objetivas:

1. **Qual é o principal problema que as Promises e o async/await buscam resolver em relação aos Callbacks?**
   - a) A dificuldade de depurar código síncrono.
   - b) O "Callback Hell", que torna o código aninhado e ilegível.
   - c) A incapacidade do JavaScript de executar operações em paralelo.
   - d) O alto consumo de memória causado por funções assíncronas.
2. **Em uma função async, qual palavra-chave é utilizada para pausar a execução até que uma Promise seja resolvida ou rejeitada?**
   - a) yield
   - b) pause
   - c) await
   - d) defer
3. **Qual método estático de Promise é ideal para executar várias Promises em paralelo e esperar que todas sejam resolvidas, ou que a primeira seja rejeitada?**
   - a) Promise.race()
   - b) Promise.any()
   - c) Promise.all()
   - d) Promise.resolve()
4. **O que acontece se uma Promise aguardada por await dentro de um bloco try for rejeitada?**
   - a) A execução da função async continua normalmente.
   - b) A função async retorna uma Promise resolvida com o erro.
   - c) Uma exceção é lançada, e o controle é transferido para o bloco catch (se existir).
   - d) O navegador congela e exibe uma mensagem de erro global.

Questão Discursiva:

Discuta como a escolha entre Callbacks, Promises e async/await pode impactar a manutenibilidade e a legibilidade de um projeto Frontend moderno, considerando as tendências de desenvolvimento e a importância da performance e acessibilidade.

Gabarito

1

Resposta: B

O "Callback Hell" é o principal problema resolvido

2

Resposta: C

await pausa a execução da função async

3

Resposta: C

Promise.all() executa em paralelo

4

Resposta: C

Exceção é lançada para o catch

Próxima Aula: Frameworks Frontend

Aula 21 – Introdução aos Frameworks Frontend

Na próxima aula, daremos um salto para o próximo nível do desenvolvimento web, explorando como frameworks como **React**, **Vue** e **Angular** revolucionam a construção de interfaces complexas e escaláveis, aproveitando muitos dos conceitos de JavaScript moderno que aprendemos.

Você descobrirá como esses frameworks utilizam o assincronismo que estudamos hoje para gerenciar estado, fazer requisições e criar experiências de usuário dinâmicas e responsivas.

01

Conceitos de Componentes

Arquitetura baseada em componentes reutilizáveis

02

Gerenciamento de Estado

Como frameworks lidam com dados assíncronos

03

Ecossistema Moderno

Ferramentas e bibliotecas do ecossistema

Recursos Adicionais

📚 MDN Web Docs

Concorrência e o Event Loop - Para aprofundar nos detalhes técnicos

💻 javascript.info

Promises, async/await - Tutoriais práticos com exemplos interativos

🎯 Google Developers

Core Web Vitals - Como o assincronismo impacta as métricas de performance

⚖️ NOTA IMPORTANTE

As informações regulatórias/legais/técnicas desta aula estão atualizadas até 2025. Consulte sempre fontes oficiais para verificar alterações.

Parabéns por concluir esta aula sobre Assincronismo em JavaScript! Continue praticando e explorando esses conceitos para se tornar um desenvolvedor frontend de excelência.

Revisão Rápida: Conceitos-Chave

Event Loop

O coração do JavaScript que permite execução não-bloqueante através do gerenciamento de Call Stack e Callback Queue

Callback Hell

Problema de aninhamento profundo de callbacks que torna o código difícil de ler e manter

Promises

Objetos que representam operações assíncronas futuras com estados Pending, Fulfilled e Rejected

async/await

Sintaxe moderna que torna código assíncrono legível como código síncrono com try/catch

Comparativo Visual:
Evolução do Assincronismo

1

Era dos Callbacks (Pré-ES6)

Primeira solução para assincronismo, mas levava ao "Callback Hell" em operações complexas. Código difícil de manter e depurar.

2

Promises (ES6/2015)

Introdução de uma estrutura mais robusta com encadeamento .then() e tratamento de erros centralizado com .catch().

3

async/await (ES2017)

Revolução na legibilidade com sintaxe que parece síncrona, mantendo todos os benefícios do assincronismo.

4

Padrões Avançados (Atual)

Promise.all(), Promise.race(), e outras ferramentas para gerenciar operações paralelas e competitivas.

Microtasks vs Macrotasks: Entendendo a Prioridade

Macrotasks (Tasks)

- **setTimeout() e setInterval()**

  Temporizadores agendados

- **I/O Operations**

  Operações de entrada/saída

- **UI Events**

  Eventos de interface do usuário

- **requestAnimationFrame**

  Animações e renderização

Microtasks

- **Promise callbacks**

  .then(), .catch(), .finally()

- **queueMicrotask()**

  API explícita para microtasks

- **MutationObserver**

  Observação de mudanças no DOM

- **process.nextTick()**

  Específico do Node.js

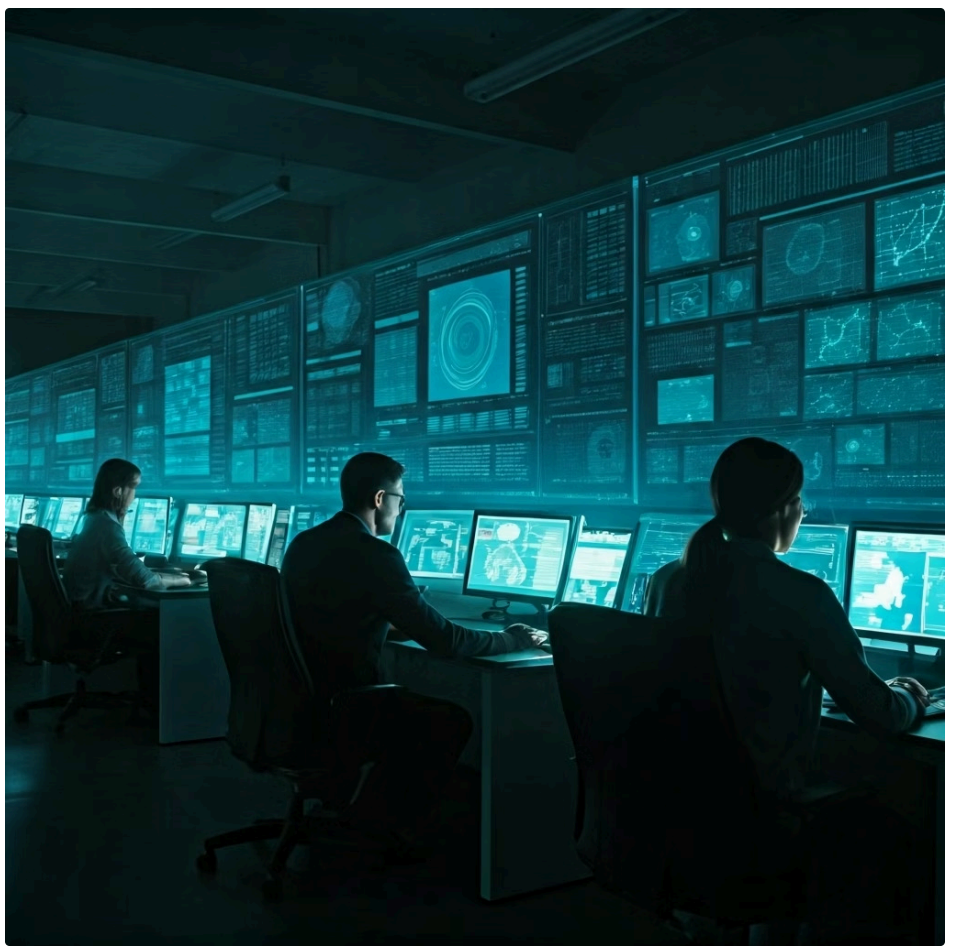
⚡ Regra de Ouro

O Event Loop processa **todas** as Microtasks na fila antes de passar para a próxima Macrotask. Isso garante que Promises sejam resolvidas rapidamente.

```
console.log('1: Síncrono');

setTimeout(() => console.log('2: Macrotask'), 0);

Promise.resolve().then(() => console.log('3: Microtask'));

console.log('4: Síncrono');

// Ordem de execução: 1, 4, 3, 2
// Microtask (3) executa antes da Macrotask (2)
```

Promise.all() vs Promise.race(): Quando Usar?

Promise.all()



Características

- Aguarda **todas** as Promises
- Retorna array com todos os resultados
- Falha se **qualquer** Promise rejeitar
- Ideal para operações independentes

Casos de Uso

- Carregar múltiplas imagens
- Buscar dados de várias APIs
- Validar múltiplos formulários
- Processar lote de operações

Promise.race()

Características

- Aguarda a **primeira** Promise
- Retorna resultado da mais rápida
- Resolve/rejeita com a primeira
- Ideal para competição de fontes

Casos de Uso

- Timeout de operações
- Múltiplas fontes de dados
- Fallback de servidores
- Otimização de latência

```
// Exemplo de timeout com Promise.race()
const operacaoLenta = fetch('https://api.exemplo.com/dados');
const timeout = new Promise((_, reject) =>
 setTimeout(() => reject('Timeout!'), 5000)
);

try {
 const resultado = await Promise.race([operacaoLenta, timeout]);
 console.log('Dados recebidos:', resultado);
} catch (erro) {
 console.error('Operação falhou ou demorou demais:', erro);
}
```

Tratamento de Erros: Estratégias Avançadas

Erro Específico por Operação

```
async function buscarDados() {
try {
const usuario = await buscarUsuario();
} catch (erro) {
console.error('Erro ao buscar usuário:', erro);
// Tratamento específico
}

try {
const posts = await buscarPosts();
} catch (erro) {
console.error('Erro ao buscar posts:', erro);
// Tratamento específico
}
}
```

Erro Global para Fluxo Completo

```
async function fluxoCompleto() {
  try {
    const usuario = await buscarUsuario();
    const posts = await buscarPosts(usuario.id);
    const comentarios = await
buscarComentarios(posts);
    return { usuario, posts, comentarios };
  } catch (erro) {
    console.error('Erro no fluxo:', erro);
    // Tratamento centralizado
    throw erro; // Re-lança se necessário
  }
}
```

Padrão de Retry (Tentativa Novamente)

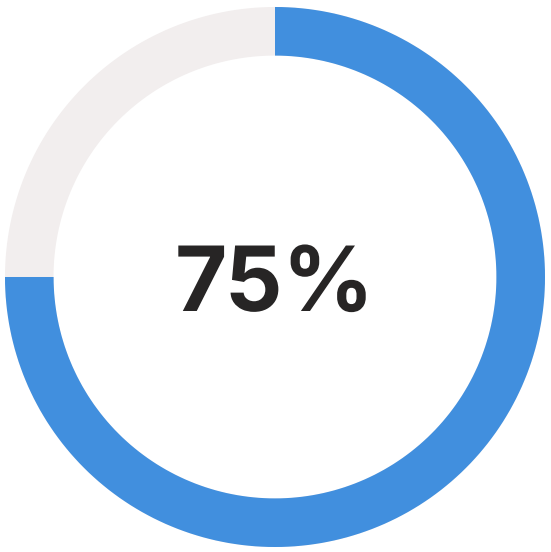
```
async function fetchComRetry(url, tentativas = 3) {
  for (let i = 0; i < tentativas; i++) {
    try {
      const resposta = await fetch(url);
      if (!resposta.ok) throw new Error('Resposta não OK');
      return await resposta.json();
    } catch (erro) {
      console.log(\`Tentativa ${i + 1} falhou\`);
      if (i === tentativas - 1) throw erro;
      await new Promise(resolve => setTimeout(resolve, 1000 * (i + 1)));
    }
  }
}
```

🛡️ Defesa em Profundidade

Combine tratamento de erros específico com tratamento global. Use retry para operações de rede e sempre forneça feedback ao usuário.

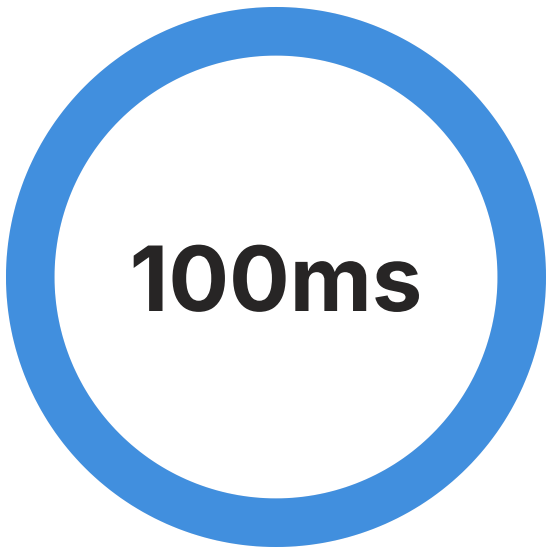
Performance Web: Core Web Vitals

O assincronismo bem implementado impacta diretamente as métricas de performance que o Google usa para ranquear sites. Entender como suas operações assíncronas afetam os Core Web Vitals é essencial para criar aplicações de alta qualidade.



LCP (Largest Contentful Paint)

Tempo até o maior elemento de conteúdo ser renderizado. Use Promise.all() para carregar recursos em paralelo.



FID (First Input Delay)

Tempo até a primeira interação. Código não-bloqueante mantém a interface responsiva.

CLS (Cumulative Layout Shift)

Estabilidade visual. Carregamento assíncrono bem gerenciado evita mudanças de layout.

Otimizações Práticas

Para LCP

- Carregue recursos críticos primeiro
- Use Promise.all() para paralelo
- Implemente lazy loading
- Otimize imagens

Para FID

- Evite operações síncronas longas
- Use Web Workers para tarefas pesadas
- Implemente debounce/throttle
- Priorize interatividade

Para CLS

- Reserve espaço para conteúdo assíncrono
- Use skeleton screens
- Defina dimensões de imagens
- Evite inserções dinâmicas

Acessibilidade (A11Y): Assincronismo Inclusivo

A forma como gerenciamos o assincronismo tem impacto direto na acessibilidade da aplicação. Usuários com deficiências visuais, motoras ou cognitivas dependem de feedback adequado durante operações assíncronas.

Feedback Visual

Use spinners, barras de progresso ou skeleton screens para indicar que uma operação está em andamento. Nunca deixe o usuário sem saber o que está acontecendo.

ARIA Live Regions

Comunique atualizações dinâmicas para leitores de tela usando aria-live="polite" ou "assertive". Essencial para conteúdo carregado assincronamente.

Estados de Botões

Desabilite botões durante operações assíncronas (aria-disabled="true") para evitar cliques múltiplos e fornecer feedback tátil.

Mensagens de Erro

Exiba erros de forma clara e acessível, usando role="alert" para notificações importantes que devem ser anunciadas imediatamente.

Exemplo Prático de Acessibilidade

```
async function carregarDadosAcessivel() {
  const botao = document.getElementById('carregar');
  const status = document.getElementById('status');
  const resultado = document.getElementById('resultado');

  // Desabilita botão e atualiza estado
  botao.disabled = true;
  botao.setAttribute('aria-busy', 'true');
  status.textContent = 'Carregando dados...';
  status.setAttribute('aria-live', 'polite');

  try {
    const dados = await fetch('/api/dados').then(r => r.json());

    // Sucesso: atualiza conteúdo
    resultado.innerHTML = renderizarDados(dados);
    status.textContent = 'Dados carregados com sucesso!';

  } catch (erro) {
    // Erro: notifica usuário
    status.textContent = 'Erro ao carregar dados. Tente novamente.';
    status.setAttribute('role', 'alert');

  } finally {
    // Reabilita botão
    botao.disabled = false;
    botao.setAttribute('aria-busy', 'false');
  }
}
```

♿ Inclusão Digital

Um site que congela ou não fornece feedback adequado durante operações assíncronas pode ser frustrante para todos, mas é especialmente problemático para usuários com deficiências.

Ferramentas Modernas: Vite e Assincronismo

O Vite é uma ferramenta de build moderna que revolucionou o desenvolvimento frontend com seu servidor de desenvolvimento extremamente rápido e otimizações inteligentes. Entender como o Vite lida com código assíncrono pode melhorar significativamente seu fluxo de trabalho.

Hot Module Replacement (HMR)

O Vite usa HMR assíncrono para atualizar módulos sem recarregar a página inteira. Isso é possível graças ao uso inteligente de Promises e importações dinâmicas.

```
// Importação dinâmica com Vite
if (import.meta.hot) {
  import.meta.hot.accept((newModule) => {
    // Atualização assíncrona do módulo
    console.log('Módulo atualizado!');
  });
}
```

Code Splitting Automático

O Vite divide automaticamente seu código em chunks menores usando importações dinâmicas, melhorando o tempo de carregamento inicial.

```
// Lazy loading de componentes
const MeuComponente = async () => {
  const modulo = await import(
    './MeuComponente.js'
  );
  return modulo.default;
};
```

1

Desenvolvimento Rápido

Servidor dev instantâneo com ESM nativo

2

Build Otimizado

Rollup para produção com tree-shaking

3

HMR Eficiente

Atualizações assíncronas sem reload

Otimizações de Performance com Vite

- **Pre-bundling de dependências:** Vite converte dependências CommonJS para ESM de forma assíncrona
- **Lazy loading automático:** Rotas e componentes são carregados sob demanda
- **Caching inteligente:** Usa HTTP caching para módulos que não mudam
- **Minificação assíncrona:** Build paralelo para melhor performance

Padrões de Design: Async Patterns

Além dos conceitos básicos, existem padrões de design específicos para código assíncrono que podem tornar suas aplicações mais robustas e escaláveis.

Retry Pattern

Tenta novamente operações que falharam, com backoff exponencial

Circuit Breaker

Previne chamadas repetidas a serviços que estão falhando

Timeout Pattern

Cancela operações que demoram muito tempo

Queue Pattern

Gerencia múltiplas operações assíncronas em fila

Debounce/Throttle

Controla frequência de execução de operações

Implementação: Debounce

```
function debounce(funcao, delay) {
  let timeoutId;

  return function(...args) {
    clearTimeout(timeoutId);

    timeoutId = setTimeout(() => {
      funcao.apply(this, args);
    }, delay);
  };
}

// Uso: busca com debounce
const buscarComDebounce = debounce(async (termo) => {
  const resultados = await fetch(\`/api/busca?q=${termo}\`);
  console.log(await resultados.json());
}, 300);

// Chama apenas após 300ms de inatividade
inputElement.addEventListener('input', (e) => {
  buscarComDebounce(e.target.value);
});
```

Implementação: Circuit Breaker

```
class CircuitBreaker {
 constructor(funcao, limiteErros = 3, timeout = 60000) {
 this.funcao = funcao;
 this.limiteErros = limiteErros;
 this.timeout = timeout;
 this.erros = 0;
 this.estado = 'FECHADO'; // FECHADO, ABERTO, MEIO_ABERTO
 this.proximaTentativa = Date.now();
 }

 async executar(...args) {
 if (this.estado === 'ABERTO') {
 if (Date.now() < this.proximaTentativa) {
 throw new Error('Circuit breaker está ABERTO');
 }
 this.estado = 'MEIO_ABERTO';
 }

 try {
 const resultado = await this.funcao(...args);
 this.sucesso();
 return resultado;
 } catch (erro) {
 this.falha();
 throw erro;
 }
 }

 sucesso() {
 this.erros = 0;
 this.estado = 'FECHADO';
 }

 falha() {
 this.erros++;
 if (this.erros >= this.limiteErros) {
 this.estado = 'ABERTO';
 this.proximaTentativa = Date.now() + this.timeout;
 }
 }
}
```

Debugging: Depurando Código Assíncrono

Depurar código assíncrono pode ser desafiador, mas com as ferramentas e técnicas certas, você pode identificar e corrigir problemas rapidamente.

1

Use console.log Estrategicamente

```
async function fluxoComplexo() {
  console.log('1: Iniciando fluxo');

  const dados = await buscarDados();
  console.log('2: Dados recebidos:', dados);

  const processado = await processar(dados);
  console.log('3: Dados processados:', processado);

  return processado;
}
```

2

Breakpoints em DevTools

Use breakpoints em funções async para pausar a execução. O Chrome DevTools mostra claramente o estado de Promises e a call stack assíncrona.

3

Async Stack Traces

Navegadores modernos mostram stack traces completas para código assíncrono, facilitando rastrear a origem de erros.

4

Promise Rejection Tracking

```
// Captura Promises rejeitadas não tratadas
window.addEventListener('unhandledrejection', (event) => {
  console.error('Promise rejeitada não tratada:', event.reason);
  event.preventDefault();
});
```

Ferramentas Úteis

Chrome DevTools

- Async stack traces
- Promise inspection
- Network throttling
- Performance profiling

Bibliotecas de Teste

- Jest (async/await support)
- Testing Library (waitFor)
- Cypress (automatic waiting)
- Playwright (async by default)

Testes: Testando Código Assíncrono

Testar código assíncrono requer abordagens específicas para garantir que as operações sejam concluídas antes das asserções. Frameworks modernos como Jest facilitam esse processo.

Testando com async/await

```
// Função a ser testada
async function buscarUsuario(id) {
  const resposta = await fetch(\`/api/usuarios/${id}\`);
  if (!resposta.ok) throw new Error('Usuário não encontrado');
  return await resposta.json();
}

// Teste com Jest
describe('buscarUsuario', () => {
  test('deve retornar dados do usuário', async () => {
    const usuario = await buscarUsuario(1);

    expect(usuario).toHaveProperty('id', 1);
    expect(usuario).toHaveProperty('nome');
  });

  test('deve lançar erro para usuário inexistente', async () => {
    await expect(buscarUsuario(999))
      .rejects
      .toThrow('Usuário não encontrado');
  });
});
```

Testando com Mocks

```
// Mock de fetch
global.fetch = jest.fn(() =>
  Promise.resolve({
    ok: true,
    json: () => Promise.resolve({ id: 1, nome: 'João' })
  })
);

test('deve chamar fetch com URL correta', async () => {
  await buscarUsuario(1);

  expect(fetch).toHaveBeenCalledWith('/api/usuarios/1');
  expect(fetch).toHaveBeenCalledTimes(1);
});
```

Sempre use async/await

Torna os testes mais legíveis e evita problemas com timing

Mock de APIs externas

Use jest.fn() ou bibliotecas como MSW para simular respostas

Teste cenários de erro

Garanta que erros sejam tratados corretamente

Use waitFor para UI

Testing Library fornece waitFor para aguardar mudanças assíncronas

Casos de Uso Reais: Aplicações Práticas

Vamos explorar cenários reais onde o assincronismo é essencial para criar experiências de usuário excepcionais.

E-commerce: Checkout Assíncrono

Validação de cupom, cálculo de frete e processamento de pagamento acontecem de forma assíncrona, mantendo a interface responsiva enquanto múltiplas APIs são consultadas.

Redes Sociais: Feed Infinito

Carregamento progressivo de posts usando Intersection Observer e async/await para buscar mais conteúdo conforme o usuário rola a página.

Chat: Mensagens em Tempo Real

WebSockets combinados com Promises para enviar e receber mensagens, com retry automático em caso de falha de conexão.

Dashboard: Múltiplas Fontes de Dados

Promise.all() para carregar dados de várias APIs em paralelo, exibindo skeleton screens enquanto aguarda, melhorando LCP.

Exemplo: Sistema de Busca com Autocomplete

```
class BuscaInteligente {
  constructor(inputElement, resultadosElement) {
    this.input = inputElement;
    this.resultados = resultadosElement;
    this.abortController = null;

    // Debounce para evitar muitas requisições
    this.buscarComDebounce = this.debounce(
      this.buscar.bind(this),
      300
    );

    this.input.addEventListener('input', (e) => {
      this.buscarComDebounce(e.target.value);
    });
  }

  debounce(func, delay) {
    let timeoutId;
    return (...args) => {
      clearTimeout(timeoutId);
      timeoutId = setTimeout(() => func(...args), delay);
    };
  }

  async buscar(termo) {
    if (!termo.trim()) {
      this.resultados.innerHTML = '';
      return;
    }

    // Cancela requisição anterior
    if (this.abortController) {
      this.abortController.abort();
    }

    this.abortController = new AbortController();
    this.resultados.innerHTML = '

Buscando...

';  try {  const resposta = await fetch(  \`/api/busca?q=${encodeURIComponent(termo)}\`,  { signal: this.abortController.signal }  );  const dados = await resposta.json();  this.exibirResultados(dados);  } catch (erro) {  if (erro.name !== 'AbortError') {  this.resultados.innerHTML = '

Erro na busca

';  }  }  }  exibirResultados(dados) {  this.resultados.innerHTML = dados  .map(item => \`

${item.titulo}

\`) .join(''); } }
```

Armadilhas Comuns: Erros a Evitar

Mesmo desenvolvedores experientes podem cair em armadilhas comuns ao trabalhar com código assíncrono. Conhecer esses problemas ajuda a evitá-los.

❌ Esquecer await

```
// ERRADO: Promise não aguardada
async function buscar() {
  const dados = buscarDados(); // Falta await!
  console.log(dados); // Promise, não os dados
}

// CORRETO
async function buscar() {
  const dados = await buscarDados();
  console.log(dados); // Dados reais
}
```

❌ Usar await em loop

```
// ERRADO: Sequencial, lento
async function buscarTodos(ids) {
  const resultados = [];
  for (const id of ids) {
    resultados.push(await buscar(id)); // Um por
vez
  }
  return resultados;
}

// CORRETO: Paralelo, rápido
async function buscarTodos(ids) {
  return await Promise.all(
    ids.map(id => buscar(id))
  );
}
```

❌ Não tratar erros

```
// ERRADO: Erro não tratado
async function processar() {
  const dados = await buscarDados();
  return processar(dados);
}

// CORRETO: Com tratamento
async function processar() {
  try {
    const dados = await buscarDados();
    return processar(dados);
  } catch (erro) {
    console.error('Erro:', erro);
    throw erro; // ou retornar valor padrão
  }
}
```

❌ Misturar .then() e await

```
// ERRADO: Mistura confusa
async function buscar() {
  const dados = await buscarDados()
    .then(d => processar(d))
    .catch(e => console.error(e));
  return dados;
}

// CORRETO: Consistente
async function buscar() {
  try {
    const dados = await buscarDados();
    return await processar(dados);
  } catch (erro) {
    console.error(erro);
  }
}
```

⚠️ Atenção Especial

A armadilha mais comum é usar `await` dentro de loops quando as operações poderiam ser paralelas. Sempre pergunte: "Essas operações dependem uma da outra?" Se não, use `Promise.all()`.

Cancelamento: AbortController

Uma funcionalidade importante mas frequentemente esquecida é a capacidade de cancelar operações assíncronas. O AbortController é a API padrão para isso.

Por que Cancelar?

Cenários Comuns

- Usuário navega para outra página
- Nova busca antes da anterior terminar
- Timeout de operações longas
- Componente desmontado no React

Benefícios

- Economiza banda e recursos
- Evita race conditions
- Melhora performance
- Previne memory leaks

Uso Básico

```
const controller = new AbortController();
const signal = controller.signal;

// Inicia requisição cancelável
fetch('/api/dados', { signal })
  .then(resposta => resposta.json())
  .then(dados => console.log(dados))
  .catch(erro => {
    if (erro.name === 'AbortError') {
      console.log('Requisição cancelada');
    } else {
      console.error('Erro:', erro);
    }
  });

// Cancela após 5 segundos
setTimeout(() => controller.abort(), 5000);
```

Padrão com React

```
function MeuComponente() {
  const [dados, setDados] = useState(null);

  useEffect(() => {
    const controller = new AbortController();

    async function buscar() {
      try {
        const resposta = await fetch('/api/dados', {
          signal: controller.signal
        });
        const dados = await resposta.json();
        setDados(dados);
      } catch (erro) {
        if (erro.name !== 'AbortError') {
          console.error('Erro:', erro);
        }
      }
    }

    buscar();

    // Cleanup: cancela ao desmontar
    return () => controller.abort();
  }, []);

  return
```

```
{dados ? dados.titulo : 'Carregando...'}
```

```
; }
```

1

Crie o Controller

new AbortController()

2

Passe o Signal

{ signal: controller.signal }

3

Cancele Quando Necessário

controller.abort()

4

Trate AbortError

if (erro.name === 'AbortError')

Web Workers: Paralelismo Real

Embora o JavaScript seja single-threaded, os Web Workers permitem executar código em threads separadas, ideal para operações pesadas que não devem bloquear a UI.

Quando Usar Web Workers

- Processamento de imagens
- Cálculos matemáticos complexos
- Parsing de grandes arquivos
- Criptografia
- Compressão/descompressão

Limitações

- Sem acesso ao DOM
- Comunicação via mensagens
- Overhead de criação
- Não compartilha memória

Exemplo: Processamento Pesado

```
// worker.js
self.addEventListener('message', (e) => {
  const { dados, operacao } = e.data;

  // Operação pesada
  const resultado = processarDados(dados, operacao);

  // Envia resultado de volta
  self.postMessage({ resultado });
});

function processarDados(dados, operacao) {
  // Simulação de processamento pesado
  let resultado = 0;
  for (let i = 0; i < 1000000000; i++) {
    resultado += Math.sqrt(i);
  }
  return resultado;
}
```

```
// main.js
async function processarComWorker(dados) {
  return new Promise((resolve, reject) => {
    const worker = new Worker('worker.js');

    worker.onmessage = (e) => {
      resolve(e.data.resultado);
      worker.terminate(); // Limpa o worker
    };

    worker.onerror = (erro) => {
      reject(erro);
      worker.terminate();
    };

    worker.postMessage({ dados, operacao: 'calcular' });
  });
}

// Uso
async function executar() {
  console.log('Iniciando processamento...');
  const resultado = await processarComWorker([1, 2, 3]);
  console.log('Resultado:', resultado);
}
```

> 🚀 **Performance Boost**
>
> Web Workers são especialmente úteis para manter a UI responsiva durante operações pesadas, melhorando significativamente o FID (First Input Delay) dos Core Web Vitals.

Async Iterators: for await...of

Para situações onde você precisa iterar sobre dados assíncronos, como streams ou paginação, o JavaScript oferece async iterators com a sintaxe for await...of.

Conceito

Um async iterator é um objeto que implementa o protocolo de iteração assíncrona, permitindo que você itere sobre valores que são resolvidos de forma assíncrona.

```
// Criando um async iterator
async function* gerarNumeros() {
  for (let i = 1; i <= 5; i++) {
    await new Promise(resolve => setTimeout(resolve, 1000));
    yield i;
  }
}

// Consumindo com for await...of
async function consumir() {
  for await (const numero of gerarNumeros()) {
    console.log(numero); // 1, 2, 3, 4, 5 (um por segundo)
  }
}

consumir();
```

Caso de Uso: Paginação

```
async function* buscarPaginado(url) {
  let pagina = 1;
  let temMais = true;

  while (temMais) {
    const resposta = await fetch(\`${url}?pagina=${pagina}\`);
    const dados = await resposta.json();

    yield dados.itens;

    temMais = dados.temProxima;
    pagina++;
  }
}

// Uso
async function carregarTodos() {
  const todosItens = [];

  for await (const itensPagina of buscarPaginado('/api/produtos')) {
    todosItens.push(...itensPagina);
    console.log(\`Carregados ${todosItens.length} itens até agora\`);
  }

  return todosItens;
}
```

Fonte de Dados

API com paginação

Async Generator

Busca página por página

for await...of

Itera sobre resultados

Dados Completos

Todos os itens carregados

Streams API

```
// Lendo um stream de forma assíncrona
async function lerStream(url) {
 const resposta = await fetch(url);
 const reader = resposta.body.getReader();
 const decoder = new TextDecoder();

 try {
 while (true) {
 const { done, value } = await reader.read();

 if (done) break;

 const texto = decoder.decode(value, { stream: true });
 console.log('Chunk recebido:', texto);
 }
 } finally {
 reader.releaseLock();
 }
}
```

Observables: Além de Promises

Embora Promises sejam excelentes para operações assíncronas únicas, Observables (da biblioteca RxJS) são mais adequados para streams de dados contínuos e eventos múltiplos.

Característica	Promise	Observable
Valores	Um único valor	Múltiplos valores ao longo do tempo
Cancelamento	Não (precisa AbortController)	Sim (unsubscribe)
Lazy	Não (executa imediatamente)	Sim (só executa ao subscrever)
Operadores	Limitados (.then, .catch)	Muitos (map, filter, debounce, etc)

Exemplo com RxJS

```
import { fromEvent, debounceTime, map, switchMap } from 'rxjs';
import { ajax } from 'rxjs/ajax';

// Busca com debounce usando Observables
const input = document.getElementById('busca');

const busca$ = fromEvent(input, 'input').pipe(
  map(event => event.target.value),
  debounceTime(300), // Aguarda 300ms de inatividade
  switchMap(termo =>
    ajax.getJSON(\`/api/busca?q=${termo}\`)
  )
);

// Subscreve para receber resultados
const subscription = busca$.subscribe({
  next: resultados => console.log('Resultados:', resultados),
  error: erro => console.error('Erro:', erro)
});

// Cancela quando necessário
// subscription.unsubscribe();
```

Reatividade

Reage automaticamente a mudanças de dados

Operadores Poderosos

Transforme, filtre e combine streams facilmente

Cancelamento Fácil

unsubscribe() para parar de receber valores

Múltiplos Valores

Ideal para eventos contínuos e streams

🔄 Quando Usar Observables

Use Observables para eventos de UI, WebSockets, Server-Sent Events, ou qualquer situação onde você precisa lidar com múltiplos valores ao longo do tempo. Para operações únicas, Promises são mais simples.

Checklist Final: Assincronismo de Qualidade

Antes de considerar seu código assíncrono pronto para produção, verifique se você atendeu a todos esses critérios essenciais.

1

✅ Tratamento de Erros Completo

- Todo await está dentro de try...catch
- Toda Promise tem .catch()
- Erros são logados e comunicados ao usuário
- Fallbacks estão implementados

2

✅ Performance Otimizada

- Operações independentes usam Promise.all()
- Debounce/throttle em eventos frequentes
- Lazy loading implementado onde apropriado
- Web Workers para operações pesadas

3

✅ Acessibilidade Garantida

- Feedback visual durante operações (spinners)
- ARIA live regions para atualizações dinâmicas
- Botões desabilitados durante requisições
- Mensagens de erro acessíveis

4

✅ Cancelamento Implementado

- AbortController para requisições canceláveis
- Cleanup em useEffect (React)
- Prevenção de race conditions
- Memory leaks evitados

5

✅ Código Limpo e Testável

- Padrão consistente (async/await preferido)
- Funções pequenas e focadas
- Testes unitários para cenários assíncronos
- Documentação de comportamento assíncrono

6

✅ Monitoramento e Debugging

- Logs estratégicos para rastreamento
- Error tracking configurado (Sentry, etc)
- Performance monitoring (Core Web Vitals)
- Alertas para falhas críticas

> "Código assíncrono de qualidade não é apenas sobre fazer funcionar, mas sobre fazer funcionar bem, de forma confiável, acessível e performática para todos os usuários."

Parabéns! 🎉

Você Dominou o Assincronismo em JavaScript

Você completou uma jornada profunda pelo mundo do assincronismo em JavaScript, desde os fundamentos do Event Loop até padrões avançados como Web Workers e Observables. Este conhecimento é fundamental para construir aplicações web modernas, performáticas e acessíveis.

35	15+	100%
Cards Completos	**Exemplos Práticos**	**Preparado**
De conteúdo aprofundado	Com código real e aplicável	Para desafios reais

Continue Sua Jornada

Na **Aula 21 – Introdução aos Frameworks Frontend**, você aplicará todo esse conhecimento de assincronismo em frameworks modernos como React, Vue e Angular, descobrindo como eles utilizam esses conceitos para criar interfaces dinâmicas e escaláveis.

Desenvolvido com ❤️ para estudantes que buscam excelência em desenvolvimento web. Continue praticando e explorando!