# Aula 28 – Otimização de Front-End

Imagine a seguinte cena: você está navegando na internet, ansioso para ver o resultado de uma busca ou acessar um serviço online. Clica no link e... nada. A página demora a carregar, elementos aparecem aos poucos, imagens pipocam na tela. A frustração é quase instantânea, não é mesmo? Em um mundo onde a velocidade da informação é rei, um site lento não é apenas um incômodo; é um obstáculo que afasta usuários, prejudica negócios e compromete a experiência digital.

No universo do desenvolvimento de aplicações web, a otimização de front-end não é um luxo, mas uma necessidade estratégica. Ela é a arte e a ciência de garantir que a parte visível e interativa de uma aplicação – aquilo que o usuário vê e com o que interage – seja carregada e executada da forma mais rápida e eficiente possível. Para estudantes universitários e profissionais que buscam aprimorar suas habilidades, dominar essas técnicas significa construir aplicações mais robustas, competitivas e, acima de tudo, amigáveis ao usuário.

Nesta aula, nosso objetivo é desvendar os segredos por trás de uma experiência web fluida e responsiva. Você aprenderá a identificar os gargalos de performance e a aplicar estratégias eficazes para minimizá-los. Abordaremos desde a redução do "peso" dos arquivos que compõem sua aplicação até a forma inteligente de carregar recursos, garantindo que seu front-end seja não apenas bonito, mas também incrivelmente rápido. Prepare-se para transformar a lentidão em agilidade e a frustração em satisfação para seus futuros usuários.

# O Inimigo Silencioso: A Lentidão no Front-End

## O Impacto da Performance

Em um cenário digital cada vez mais dinâmico, a paciência do usuário é um recurso escasso. Estudos mostram que a cada segundo adicional no tempo de carregamento de uma página, a taxa de rejeição (bounce rate) aumenta exponencialmente, e a satisfação do usuário despenca.

Para empresas, isso se traduz em perda de vendas, menor engajamento e uma reputação digital comprometida. Pense em um e-commerce: se o cliente não consegue ver os produtos rapidamente, ele simplesmente vai para o concorrente.

## A Raiz do Problema

O problema reside no fato de que, à medida que as aplicações web se tornam mais ricas em funcionalidades e design, elas também se tornam mais "pesadas". Isso significa mais código JavaScript para interatividade, mais folhas de estilo CSS para o visual e mais imagens e vídeos para o conteúdo.

Todos esses recursos precisam ser baixados, processados e renderizados pelo navegador do usuário, e cada etapa pode ser um gargalo.

> **Nossa missão:** combater esse "peso" excessivo e otimizar cada etapa do processo de carregamento. É como preparar um carro de corrida: não basta ter um motor potente; é preciso reduzir o peso, otimizar a aerodinâmica e garantir que cada componente trabalhe em perfeita sintonia para alcançar a máxima velocidade.

# Minificação: Tirando o Excesso de Bagagem

### O que é Minificação?

Processo de remover caracteres desnecessários do código-fonte sem alterar sua funcionalidade

### O que é Removido?

Espaços em branco, quebras de linha, comentários e nomes de variáveis encurtados

### Ferramentas Principais

UglifyJS e Terser para JavaScript, CSSNano para CSS

Você já se perguntou o que acontece com o código que escrevemos antes de ele ser entregue ao navegador do usuário? Muitas vezes, ele passa por um processo de "emagrecimento" chamado minificação. Este é o primeiro passo fundamental para reduzir o tamanho dos arquivos e, consequentemente, o tempo de carregamento da sua aplicação.

A minificação consiste em remover todos os caracteres desnecessários do código-fonte sem alterar sua funcionalidade. Isso inclui espaços em branco, quebras de linha, comentários e, em alguns casos, até mesmo encurtar nomes de variáveis e funções. Para nós, desenvolvedores, esses elementos são cruciais para a legibilidade e manutenção do código. No entanto, para o navegador, eles são apenas bytes extras que precisam ser baixados e que não contribuem para a execução lógica.

> Pense na minificação como arrumar uma mala de viagem: você tira tudo o que é supérfluo – embalagens grandes, papéis desnecessários – para que caiba mais coisas essenciais e a mala fique mais leve.

```
// Código JavaScript original (exemplo)
function calcularTotal(precoUnitario, quantidade) {
 // Esta função calcula o valor total de um produto
 const total = precoUnitario * quantidade;
 return total;
}

// Código JavaScript minificado (exemplo)
function calcularTotal(e,t){const n=e*t;return n}
```

# Compressão: Empacotando para a Viagem Rápida

Depois de minificar seu código, ele já está mais leve, mas ainda pode ser reduzido para a transmissão pela rede. É aqui que entra a compressão, uma técnica que atua como um "empacotador a vácuo" para seus arquivos. Ela pega o código minificado e o comprime ainda mais, transformando-o em um pacote menor que viaja mais rápido pela internet.

### Gzip

Algoritmo de compressão tradicional e amplamente suportado. Identifica padrões repetitivos e os substitui por referências mais curtas.

### Brotli

Algoritmo mais moderno que oferece taxas de compressão superiores ao Gzip, especialmente para texto e código.

Os algoritmos de compressão mais comuns no contexto web são Gzip e Brotli. Ambos funcionam identificando padrões repetitivos nos dados e substituindo-os por referências mais curtas. Imagine que você tem um livro com muitas palavras repetidas. Em vez de escrever a palavra "otimização" cem vezes, você poderia escrever "otimização" uma vez e, nas outras 99, usar um símbolo que remeta à primeira ocorrência. É uma simplificação, mas ilustra o princípio.

**Vantagem Principal:** A compressão reduz significativamente o volume de dados transferidos, acelerando o tempo de carregamento e economizando largura de banda. A maioria dos servidores web modernos já vem configurada para aplicar Gzip ou Brotli automaticamente.

# Minificação e Compressão em Ação: Otimizando Todos os Assets

## HTML

A minificação remove espaços em branco entre tags, quebras de linha e comentários. Embora o ganho não seja tão drástico quanto em JS ou CSS, ele ainda contribui para um carregamento mais rápido.

## CSS

Além da remoção de espaços e comentários, inclui otimização de seletores, fusão de regras duplicadas e conversão de valores (ex: rgb(255, 0, 0) para #F00).

## JavaScript

A minificação é ainda mais crítica. Pode renomear variáveis para nomes curtos (a, b, c) e realizar "tree shaking", removendo código não utilizado.

A beleza da minificação e compressão é que elas podem ser aplicadas a praticamente todos os tipos de arquivos de texto que compõem seu front-end: HTML, CSS e JavaScript. Cada um tem suas particularidades, mas o objetivo final é sempre o mesmo: entregar o mínimo de bytes possível ao navegador do usuário.

A integração dessas técnicas geralmente ocorre durante o processo de build da aplicação. Ferramentas como Webpack, Rollup ou Vite, com seus respectivos plugins, automatizam a minificação e a compressão de todos os assets, transformando seu código-fonte legível em uma versão otimizada pronta para produção. Isso significa que você, como desenvolvedor, pode continuar escrevendo código limpo e bem comentado, enquanto o sistema se encarrega de prepará-lo para a máxima performance.

# Otimização de Imagens: A Carga Mais Pesada da Web

### O Vilão Comum

Se há um vilão comum na lentidão das páginas web, ele frequentemente se esconde nas imagens. Fotos de alta resolução, ícones detalhados e gráficos complexos são essenciais para a estética e a comunicação visual, mas podem facilmente se tornar a maior parte do "peso" de uma página.

### O Desafio do Equilíbrio

O desafio aqui é equilibrar qualidade visual com performance. Não podemos simplesmente reduzir a qualidade de todas as imagens a ponto de ficarem pixelizadas, pois isso prejudicaria a experiência do usuário.

### A Solução Inteligente

A solução reside em uma abordagem inteligente, onde cada imagem é tratada de forma a ser a mais leve possível, sem sacrificar a clareza ou o impacto visual.

> Pense na otimização de imagens como escolher o veículo certo para transportar uma carga. Você não usaria um caminhão enorme para levar uma única caixa pequena, nem tentaria carregar um elefante em um carro de passeio.

Da mesma forma, precisamos escolher o formato de imagem adequado, o tamanho correto e a técnica de carregamento mais eficiente para cada contexto. Ignorar a otimização de imagens é como tentar correr uma maratona com pesos amarrados aos tornozelos: você pode até chegar lá, mas será muito mais lento e exaustivo.

# Formatos de Imagem Modernos e Eficientes

A escolha do formato de imagem é um dos pilares da otimização. Por muito tempo, JPEG e PNG foram os padrões, mas a evolução da web trouxe opções mais eficientes que oferecem melhor compressão e qualidade superior. Conhecer e aplicar esses formatos modernos é crucial para reduzir o tamanho dos arquivos sem comprometer a fidelidade visual.

### WebP

Desenvolvido pelo Google, oferece compressão superior (com e sem perdas) em comparação com JPEG e PNG. Suporta transparência e animação, tornando-o extremamente versátil.

### AVIF

Baseado no codec de vídeo AV1, é ainda mais recente e promete compressão ainda maior que o WebP, com excelente qualidade. Ideal para imagens de alta resolução.

> **Estratégia Ideal:** Use formatos modernos sempre que possível, oferecendo JPEG ou PNG como fallback para navegadores mais antigos. O elemento HTML <picture> é perfeito para isso, permitindo que você especifique múltiplas fontes de imagem.

A estratégia ideal é usar esses formatos modernos sempre que possível, oferecendo JPEG ou PNG como fallback para navegadores mais antigos que ainda não os suportam. O elemento HTML <picture> é perfeito para isso, permitindo que você especifique múltiplas fontes de imagem, e o navegador escolhe a mais adequada e eficiente. Ao adotar WebP e AVIF, você não apenas acelera o carregamento da página, mas também oferece uma experiência visual mais rica com menos dados.

# Lazy Loading: Carregando o Essencial Primeiro

Nem todo conteúdo de uma página web precisa ser carregado no momento em que a página é acessada. Pense em uma longa página de blog ou um feed de notícias: o usuário só vê a parte superior inicialmente. Carregar todas as imagens e vídeos que estão "abaixo da dobra" (fora da área visível da tela) é um desperdício de recursos e atrasa o carregamento do conteúdo que realmente importa. É aqui que o *lazy loading* entra em cena.

01

### Carregamento Inicial

A página carrega apenas o conteúdo visível na viewport (área visível da tela)

02

### Detecção de Rolagem

O navegador monitora quando o usuário rola a página e novos elementos entram na área visível

03

### Carregamento Sob Demanda

Recursos adicionais são carregados apenas quando necessários, conforme o usuário interage

Lazy loading, ou carregamento preguiçoso, é uma técnica que adia o carregamento de recursos (como imagens e vídeos) até que eles sejam realmente necessários, ou seja, quando o usuário rola a página e eles se tornam visíveis na viewport. Isso significa que a página carrega mais rápido inicialmente, pois o navegador só precisa baixar o que está visível. O restante é carregado sob demanda, conforme o usuário interage com a página.

> Imagine que você está em um restaurante e o garçom traz apenas o cardápio e a água. Ele não traz todos os pratos do menu de uma vez, certo? Ele espera você fazer seu pedido para então preparar e servir o que você realmente quer.

O lazy loading funciona de forma similar: o navegador "pede" os recursos apenas quando o usuário "pede" para vê-los. A implementação é relativamente simples, muitas vezes com o atributo loading="lazy" em tags <img> e <iframe>, ou através de bibliotecas JavaScript que utilizam a Intersection Observer API para detectar quando um elemento entra na área visível.

# Code Splitting: Dividindo para Conquistar

À medida que as aplicações JavaScript se tornam mais complexas, o tamanho do bundle de código pode crescer exponencialmente. Um único arquivo JavaScript grande pode bloquear o carregamento da página, atrasar a interatividade e consumir muita memória. Para combater isso, utilizamos uma técnica poderosa chamada *Code Splitting*, ou divisão de código.

### Bundle Único

Todo o código em um arquivo grande

### Code Splitting

Divisão em pedaços menores

### Carregamento Otimizado

Apenas o necessário é carregado

Code Splitting é o processo de dividir o bundle de JavaScript em pedaços menores, que podem ser carregados sob demanda ou em paralelo. Em vez de enviar todo o código da aplicação de uma vez, você envia apenas o que é estritamente necessário para a visualização inicial da página. O restante do código é carregado de forma assíncrona, apenas quando o usuário precisa dele – por exemplo, ao navegar para uma nova rota, abrir um modal ou interagir com um componente específico.

> **Analogia:** Pense na construção de uma casa. Seria ineficiente trazer todos os materiais de uma vez para o canteiro de obras. Em vez disso, os materiais são entregues em etapas, conforme cada fase da construção avança.

O Code Splitting aplica essa mesma lógica ao seu código, garantindo que apenas os "materiais" essenciais para a "fundação" da sua aplicação sejam carregados primeiro, e o restante venha conforme a "construção" avança. Isso melhora drasticamente o tempo de carregamento inicial e a responsividade da aplicação.

# Carregamento Assíncrono de Módulos e Componentes

## Atributos async e defer

- **async:** Script é baixado em paralelo e executado assim que disponível, sem bloquear a renderização
- **defer:** Script é baixado em paralelo, mas execução é adiada até o HTML ser completamente analisado

## Importação Dinâmica

Frameworks modernos usam a sintaxe import() para carregar módulos sob demanda:

- Route-based splitting (por rota)
- Component-based splitting (por componente)

A implementação do Code Splitting está intrinsecamente ligada ao conceito de carregamento assíncrono. Em vez de ter um único script que o navegador precisa baixar e executar antes de renderizar qualquer coisa, podemos instruir o navegador a carregar partes do nosso código em segundo plano, sem bloquear a renderização da página.

Os atributos async e defer nas tags <script> são os mecanismos mais básicos para isso. Um script com async é baixado em paralelo com a análise do HTML e executado assim que estiver disponível, sem bloquear a renderização. Já um script com defer também é baixado em paralelo, mas sua execução é adiada até que o HTML seja completamente analisado, antes do evento DOMContentLoaded. Ambos são cruciais para garantir que o JavaScript não atrase a exibição do conteúdo principal.

No contexto de frameworks e bibliotecas modernas, o carregamento assíncrono é frequentemente gerenciado por ferramentas de build como Webpack ou Vite, em conjunto com a sintaxe de importação dinâmica do JavaScript (import()). Isso permite que você defina "pontos de divisão" no seu código, onde módulos ou componentes específicos serão carregados apenas quando forem solicitados. Por exemplo, você pode carregar o código de uma página de administração apenas quando o usuário navegar para ela, ou o código de um componente de chat apenas quando o usuário clicar para abri-lo. Essa estratégia, conhecida como *route-based splitting* ou *component-based splitting*, otimiza o Time To Interactive (TTI), tornando a aplicação responsiva muito mais rapidamente.

# Priorizando o Essencial: Critical CSS e JavaScript

Mesmo com minificação, compressão e code splitting, ainda existe um conjunto de estilos e scripts que são absolutamente essenciais para a primeira renderização da página – o que chamamos de "acima da dobra" (above the fold). Se esses recursos não estiverem disponíveis rapidamente, o usuário verá uma página sem estilo ou sem funcionalidade básica, resultando em um "flash de conteúdo não estilizado" (FOUC) ou uma experiência quebrada.

### Critical CSS

Identifica e extrai apenas os estilos necessários para renderizar o conteúdo visível. Esses estilos são "inlined" (embutidos) diretamente no <head> do HTML.

### Critical JavaScript

Código JavaScript fundamental para a interatividade inicial. Pode ser inlined ou carregado com async/defer para evitar bloqueios.

A técnica de Critical CSS envolve identificar e extrair apenas os estilos necessários para renderizar o conteúdo visível da página. Esses estilos são então "inlined" (embutidos diretamente) na tag <style> dentro do <head> do HTML. Isso garante que o navegador tenha todos os estilos críticos imediatamente, sem precisar esperar por um arquivo CSS externo ser baixado. O restante do CSS pode ser carregado de forma assíncrona ou deferida.

Da mesma forma, o Critical JavaScript refere-se ao código JavaScript que é fundamental para a interatividade inicial da página. Assim como o CSS, ele pode ser inlined ou carregado com async/defer para evitar bloqueios. A ideia é que o navegador possa renderizar a parte mais importante da página o mais rápido possível, oferecendo uma experiência inicial fluida.

> É como montar um kit de emergência: você coloca os itens mais vitais (água, comida, primeiros socorros) em um lugar de fácil acesso, enquanto o restante da bagagem pode ser acessado depois.

Essa priorização é fundamental para otimizar métricas como o Largest Contentful Paint (LCP).

# Ferramentas e Automação para Otimização

Realizar todas as otimizações de front-end manualmente seria uma tarefa hercúlea e propensa a erros. Felizmente, o ecossistema de desenvolvimento web oferece uma vasta gama de ferramentas que automatizam esses processos, integrando-os perfeitamente ao fluxo de trabalho de desenvolvimento.

### Webpack

Bundler poderoso e altamente configurável. Oferece plugins para minificação (Terser), otimização de CSS (CSSNano), code splitting e muito mais.

### Vite

Build tool moderna e extremamente rápida. Usa ESM nativo durante o desenvolvimento e Rollup para produção, com otimizações automáticas.

### Rollup

Focado em criar bundles otimizados para bibliotecas. Excelente tree shaking e suporte a múltiplos formatos de saída.

### Imagemin / Sharp

Ferramentas especializadas para otimização de imagens. Comprimem e convertem imagens automaticamente para formatos modernos como WebP.

As **ferramentas de build** são o coração dessa automação. Webpack, Rollup e Vite são os mais populares, cada um com suas particularidades, mas todos com o objetivo de empacotar, transformar e otimizar seu código para produção. Eles funcionam como orquestradores, aplicando minificação (com plugins como Terser para JS e CSSNano para CSS), compressão, tree shaking e code splitting de forma configurável.

Além das ferramentas de build, existem plugins e bibliotecas específicas para otimização de imagens (como imagemin ou sharp), que podem ser integrados ao seu pipeline de build para comprimir e converter imagens automaticamente para formatos modernos como WebP. A integração contínua e entrega contínua (CI/CD) também desempenham um papel crucial, permitindo que essas otimizações sejam aplicadas e testadas automaticamente a cada nova versão do código, garantindo que a performance seja sempre uma prioridade. Ao dominar essas ferramentas, você não apenas acelera suas aplicações, mas também otimiza seu próprio tempo de desenvolvimento.

# Métricas de Performance e Core Web Vitals

Como saber se nossas otimizações estão realmente funcionando? A resposta está nas métricas de performance. O Google, em particular, tem impulsionado a adoção das **Core Web Vitals (CWV)**, um conjunto de métricas que medem a experiência do usuário em termos de carregamento, interatividade e estabilidade visual. Elas são cruciais não apenas para a satisfação do usuário, mas também para o SEO (Search Engine Optimization).

### Largest Contentful Paint (LCP)

Mede o tempo que leva para o maior elemento de conteúdo visível na viewport ser renderizado. Um LCP rápido indica que o conteúdo principal da página está visível rapidamente.

### First Input Delay (FID)

Mede o tempo desde a primeira interação do usuário (clique, toque) até o navegador ser capaz de responder a essa interação. Um FID baixo significa que a página é responsiva rapidamente.

### Cumulative Layout Shift (CLS)

Mede a estabilidade visual da página. Um CLS baixo significa que os elementos da página não se movem inesperadamente enquanto o usuário tenta interagir com eles.

As três Core Web Vitals principais são: **Largest Contentful Paint (LCP)**, **First Input Delay (FID)** e **Cumulative Layout Shift (CLS)**.

| Conceito | Âmbito/Aplicação | Base/Origem | Exemplo |
|---|---|---|---|
| Largest Contentful Paint (LCP) | Tempo de carregamento do maior elemento visível | Percepção de carregamento do conteúdo principal | Imagem de destaque ou título principal de um artigo aparecendo rapidamente |
| First Input Delay (FID) | Responsividade à primeira interação do usuário | Interatividade e fluidez da interface | Tempo entre o clique em um botão e a resposta da aplicação |
| Cumulative Layout Shift (CLS) | Estabilidade visual da página | Movimento inesperado de elementos na tela | Um banner que carrega atrasado e empurra o conteúdo para baixo |

**Ferramentas Essenciais:** Google Lighthouse e PageSpeed Insights são indispensáveis para auditar a performance de sua aplicação e obter pontuações para essas métricas.

Ferramentas como Google Lighthouse e PageSpeed Insights são indispensáveis para auditar a performance de sua aplicação e obter pontuações para essas métricas. Cada técnica de otimização que discutimos nesta aula contribui diretamente para a melhoria de uma ou mais dessas CWV. Por exemplo, minificação e compressão impactam o LCP, enquanto code splitting e carregamento assíncrono melhoram o FID. Monitorar essas métricas é essencial para garantir que suas otimizações estejam gerando resultados tangíveis e uma experiência de usuário de alta qualidade.

# Desafios e Considerações Avançadas

A otimização de front-end é um campo em constante evolução, e embora as técnicas básicas sejam poderosas, o mundo real apresenta desafios e considerações mais complexas. Não existe uma solução única para todos os problemas, e muitas vezes é preciso equilibrar a performance com a complexidade do desenvolvimento e a experiência do desenvolvedor.

### Gestão de Cache

O cache do navegador é poderoso para acelerar recursos repetidos, mas a otimização de assets pode gerar novos nomes de arquivos (com hashes) a cada deploy. Isso exige estratégias de cache inteligentes para garantir que os usuários sempre recebam a versão mais recente.

### Novas Tecnologias

HTTP/3 promete melhorias significativas na forma como os dados são transportados pela rede, reduzindo a latência. Conceitos como Early Hints e Signed Exchanges também estão ganhando força.

Um dos principais desafios é a gestão de cache. Embora o cache do navegador seja uma ferramenta poderosa para acelerar o carregamento de recursos repetidos, a otimização de assets (como a minificação e compressão) pode gerar novos nomes de arquivos (com hashes) a cada deploy. Isso exige estratégias de cache inteligentes para garantir que os usuários sempre recebam a versão mais recente do código, sem invalidar o cache de forma ineficiente.

Além disso, novas tecnologias e padrões estão sempre surgindo. O **HTTP/3**, por exemplo, promete melhorias significativas na forma como os dados são transportados pela rede, reduzindo a latência. Conceitos como **Early Hints** e **Signed Exchanges** também estão ganhando força, oferecendo formas de pré-carregar recursos e garantir a autenticidade do conteúdo de forma ainda mais eficiente. Manter-se atualizado com essas tendências é crucial para qualquer especialista em arquitetura de aplicações web. A otimização é uma jornada contínua, não um destino final, e exige um olhar atento para as inovações e para as necessidades em constante mudança dos usuários e da própria web.

# Consolidação e Próximos Passos

Chegamos ao fim de nossa jornada pela otimização de front-end. Vimos que construir aplicações web rápidas e eficientes não é apenas uma questão técnica, mas uma estratégia fundamental para garantir uma excelente experiência ao usuário, aumentar o engajamento e impulsionar o sucesso de qualquer projeto digital. Desde a minificação e compressão de cada byte de código até a otimização inteligente de imagens e a divisão estratégica do JavaScript, cada técnica contribui para um front-end mais ágil e responsivo.

### Configure Build Tools

Sempre configure suas ferramentas de build para minificar e comprimir seus assets (JS, CSS, HTML)

### Formatos Modernos

Priorize formatos de imagem modernos como WebP e AVIF, usando o elemento <picture>

### Lazy Loading

Implemente lazy loading para imagens e vídeos que não estão visíveis inicialmente

### Code Splitting

Utilize code splitting e carregamento assíncrono para JavaScript, dividindo seu bundle em chunks menores

### Monitore Métricas

Monitore constantemente as Core Web Vitals de sua aplicação para identificar e corrigir gargalos de performance

**Próxima Aula:** Na próxima aula, aprofundaremos ainda mais nossa discussão sobre performance, explorando as **Content Delivery Networks (CDNs)**. Veremos como essas redes distribuídas podem levar seus assets ainda mais perto dos usuários, reduzindo a latência e acelerando o carregamento global de suas aplicações.

## Recursos Adicionais:

- **Web.dev (Google):** Guia completo sobre Core Web Vitals e otimização de performance.
- **MDN Web Docs:** Documentação detalhada sobre atributos async, defer e elemento <picture>.
- **Artigos sobre Webpack/Vite:** Para aprofundar na configuração de ferramentas de build para otimização.

## Autoavaliação

1. Qual das seguintes técnicas tem como principal objetivo remover caracteres desnecessários (espaços, comentários) do código-fonte sem alterar sua funcionalidade?
   a) Compressão Gzip
   b) Lazy Loading
   c) Minificação
   d) Code Splitting
2. Para otimizar o carregamento de imagens e garantir que o navegador escolha o formato mais eficiente (como WebP ou AVIF) com um fallback para formatos mais antigos, qual elemento HTML é mais adequado?
   a) <img> com atributo src
   b) <picture>
   c) <canvas>
   d) <svg>
3. A métrica Core Web Vital que mede o tempo desde a primeira interação do usuário com a página até o navegador ser capaz de responder a essa interação é conhecida como:
   a) Largest Contentful Paint (LCP)
   b) Cumulative Layout Shift (CLS)
   c) First Input Delay (FID)
   d) Time To First Byte (TTFB)
4. Qual das seguintes afirmações sobre Code Splitting está **incorreta**?
   a) Ele divide o bundle de JavaScript em pedaços menores.
   b) Seu objetivo é carregar todo o código da aplicação de uma vez para garantir a interatividade imediata.
   c) Pode ser implementado com importações dinâmicas (import()).
   d) Ajuda a melhorar o tempo de carregamento inicial da página.
5. Explique a importância da otimização de Critical CSS e como ela contribui para a melhoria da experiência do usuário.

## Gabarito:

1. c)
2. b)
3. c)
4. b)