Hiago de Oliveira Mendes e Lucas Sales Salvo Petruci

Uma Análise Comparativa entre *Client-Side Rendering* e *Server-Side Rendering* em Aplicações Web

Campos dos Goytacazes-RJ

Setembro de 2025

Hiago de Oliveira Mendes e Lucas Sales Salvo Petruci

Uma Análise Comparativa entre *Client-Side Rendering* e *Server-Side Rendering* em Aplicações Web

Trabalho de Conclusão apresentado ao curso Bacharelado em Sistemas de Informação do Instituto Federal de Educação, Ciência e Tecnologia Fluminense, como parte dos requisitos para a obtenção do título de Bacharel em Sistemas de Informação.

Instituto Federal de Educação, Ciência e Tecnologia Fluminense

Orientador: Prof. D.Sc. Ronaldo Amaral Santos

Campos dos Goytacazes-RJ

Setembro de 2025

Hiago de Oliveira Mendes e Lucas Sales Salvo Petruci

Uma Análise Comparativa entre *Client-Side Rendering* e *Server-Side Rendering* em Aplicações Web

Trabalho de Conclusão apresentado ao curso Bacharelado em Sistemas de Informação do Instituto Federal de Educação, Ciência e Tecnologia Fluminense, como parte dos requisitos para a obtenção do título de Bacharel em Sistemas de Informação.

Campos dos Goytacazes-RJ, 31 de Setembro de 2025.

Prof. D.Sc. Ronaldo Amaral Santos (orientador)
Instituto Federal Fluminense (IFF)

Prof. D.Sc. banca 1
Instituto Federal Fluminense (IFF)

Prof. D.Sc. banca 2
Instituto Federal Fluminense (IFF)

Campos dos Goytacazes-RJ
Setembro de 2025

Agradecimentos

agradecimentos

Resumo

Resumo aqui **Palavras-chaves:** SPA X MPA.

Abstract

abstract here

Keywords: Spa, Mpa.

Lista de Figuras

Lista de quadros

Lista de codigos

Siglas

CSR	Client-Side Rendering
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
QUIC	Quick UDP Internet Connections
SEO	Search Engine Optimization
SPA	Single Page Application
SSL	Secure Sockets Layer
SSR	Server-Side Rendering
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol

Sumário

1 Introdução

1.1 Problema e contexto

O crescimento acelerado da web e o aumento da complexidade das aplicações modernas impuseram novos desafios ao desenvolvimento e à entrega de conteúdos na internet. Com o crescimento exponencial da web, estima-se que aproximadamente 252 mil novos sites sejam desenvolvidos diariamente, demonstrando não apenas a rapidez com que aplicações são criadas, mas também a necessidade crescente de estratégias eficientes para otimização de desempenho e escalabilidade (SITEEFY, 2021). A escolha da abordagem de renderização tornou-se um fator determinante para a experiência do usuário e a escalabilidade dos sistemas. Inicialmente, os sites eram compostos por páginas estáticas, cujo conteúdo era carregado diretamente do servidor. Com a evolução das tecnologias frontend, novas abordagens surgiram, destacando-se *Client-Side Rendering (CSR)* e *Server-Side Rendering (SSR)*. Cada uma dessas técnicas possui características específicas que influenciam diretamente o desempenho e a experiência do usuário.

A performance em websites é um fator determinante para o sucesso de qualquer aplicação web. O desempenho, frequentemente medido pelo tempo de carregamento das páginas, desempenha um papel fundamental na experiência do usuário e na taxa de conversão de visitantes (WAGNER, 2016). Uma página que carrega rapidamente proporciona uma navegação mais fluida, reduzindo a taxa de rejeição e aumentando a retenção de usuários. Além disso, o desempenho da página não se limita a impactar a experiência do usuário, mas também interfere diretamente no *Search Engine Optimization (SEO)*, tornando-se um critério essencial de indexação e ranqueamento em plataformas como o Google (GOOGLE, 2010).

Um exemplo notável de desafios enfrentados na escolha da estratégia de renderização ocorreu no *Twitter*. Em 2010, a empresa lançou uma nova versão de sua plataforma, conhecida como New Twitter, que utilizava extensivamente a renderização no lado do cliente (CSR) para aprimorar a interatividade e a experiência do usuário. No entanto, essa abordagem resultou em problemas significativos de desempenho, especialmente para usuários com conexões de internet mais lentas ou dispositivos menos potentes. Além disso, a dependência intensa de JavaScript dificultou a indexação de conteúdo pelos mecanismos de busca, impactando negativamente a otimização para motores de busca (SEO) (TRADEOFFS. . . , 2015). Reconhecendo essas limitações, o Twitter decidiu retornar à renderização no lado do servidor (SSR) em 2012, visando melhorar o desempenho e a acessibilidade de sua plataforma.

A arquitetura de frontend desempenha papel fundamental ao definir o fluxo de desenvolvimento e a escolha entre CSR e SSR, sendo indispensável a adoção de um sistema modular e eficiente, capaz de ser mantido e escalado de forma sustentável (GODBOLT, 2016). Na abordagem CSR, a renderização ocorre diretamente no navegador do usuário, reduzindo a carga no servidor, mas exigindo mais processamento no cliente; já na SSR, o conteúdo é gerado no servidor antes de ser enviado ao cliente, o que proporciona carregamento mais rápido e melhor desempenho em dispositivos menos potentes. A decisão entre essas estratégias está diretamente ligada à performance da aplicação e deve considerar fatores como tempo de carregamento, complexidade da página e número de requisições HTTP (WAGNER, 2016), já que diferentes abordagens afetam não apenas a experiência do usuário, mas também os custos operacionais e a infraestrutura necessária para suportar a aplicação.

1.2 Justificativa

Nos últimos anos, observou-se um crescimento expressivo na adoção de abordagens de renderização tanto no lado do cliente (CSR) quanto no lado do servidor (SSR) em aplicações web, sobretudo quando comparadas a modelos tradicionais que utilizam apenas páginas estáticas ou *templates* processados integralmente no servidor. Esse avanço deve-se, em grande parte, à busca contínua por melhor desempenho, experiências de usuário mais rápidas e dinâmicas, além da popularização de *frameworks* e bibliotecas que simplificam a implementação dessas abordagens (EMADAMERHO-ATORI, 2024).

Contudo, essas técnicas são frequentemente empregadas de maneira inadequada em muitos projetos, seja pela falta de entendimento de suas vantagens e limitações, seja por uma análise superficial das necessidades do produto. Um exemplo ilustrativo dessa realidade pode ser visto na experiência do *Airbnb*, que optou por uma abordagem de SSR com o intuito de melhorar o desempenho em dispositivos com recursos limitados e, sobretudo, otimizar a indexação de seu vasto catálogo de acomodações em mecanismos de busca (NEARY, 2017). Por outro lado, a equipe do *Instagram* enfrentou desafios ao equilibrar o carregamento dinâmico de conteúdo no cliente com a necessidade de garantir uma experiência fluida aos usuários, levando-os a adotar soluções híbridas que envolvem tanto CSR quanto SSR em diferentes partes da aplicação (CONNER, 2019).

Paralelamente a esses casos, identifica-se uma carência de estudos de caso reais que analisem de forma aprofundada o impacto da adoção de CSR e SSR, principalmente no contexto nacional. Enquanto algumas publicações se concentram em apenas uma dessas abordagens, outras fornecem exemplos excessivamente simplificados, limitando a compreensão dos desafios técnicos e de negócios ao combinar essas estratégias em sistemas complexos.

Diante desse cenário, o presente trabalho busca contribuir na análise detalhada sobre a implementação de CSR e SSR, avaliando de forma introdutória seus efeitos no desempenho, na experiência do usuário, segurança, otimização do SEO e na escalabilidade de aplicações web modernas. Por meio de um estudo de caso abrangente, espera-se fornecer subsídios que possam orientar equipes de desenvolvimento e gestores na seleção e aplicação dessas técnicas, auxiliando na construção de sistemas mais robustos e eficientes, em sintonia com as demandas atuais do mercado.

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo geral deste trabalho de conclusão de curso é apresentar uma análise comparativa detalhada sobre a implementação de casos com CSR e SSR, avaliando seus efeitos no desempenho, na experiência do usuário, segurança, otimização do SEO e na escalabilidade de aplicações web modernas.

1.3.2 Objetivos Específicos

- Apresentar estratégias de escolhas entre CSR e SSR, analisando métricas de desempenho, tempo de resposta, experiência do usuário e carga no servidor.
- Identificar as principais limitações e desafios enfrentados na escolha entre CSR e SSR, considerando otimização de SEO, escalabilidade e requisitos de infraestrutura.
- Apresentar recomendações práticas para desenvolvedores e gestores, auxiliando na tomada de decisão sobre qual abordagem utilizar com base nos objetivos do projeto e nas demandas do mercado.

1.4 Metodologia

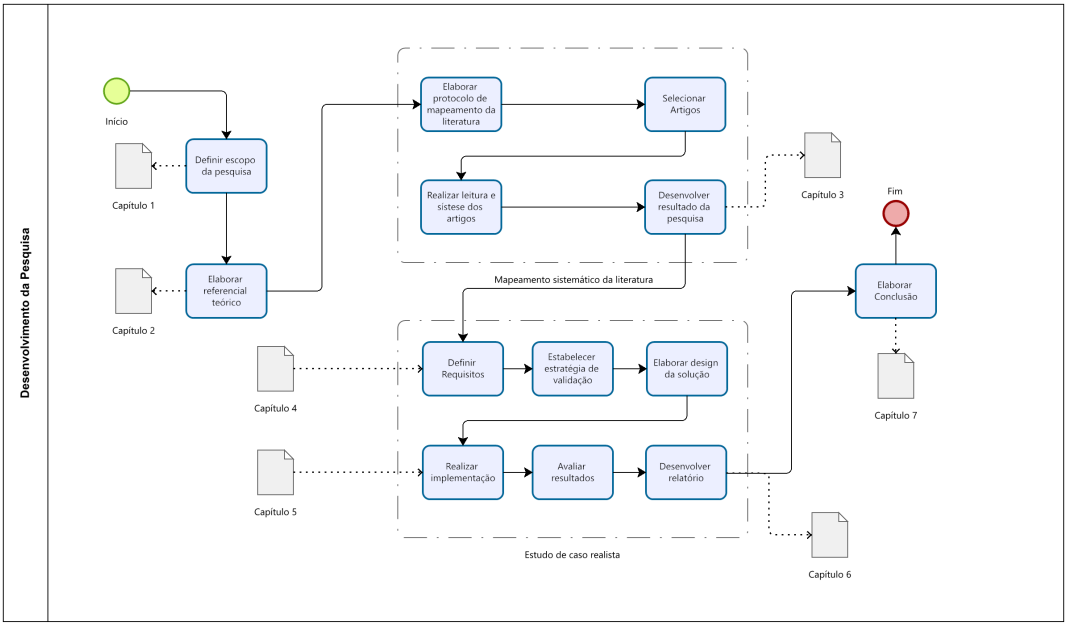
Pode-se observar na Figura 1 as etapas de execução desta pesquisa. Inicialmente, o escopo é definido e o primeiro capítulo é elaborado, onde são apresentados o contexto do estudo, as justificativas e os objetivos a serem atingidos. Em seguida, constrói-se a fundamentação teórica, buscando fornecer uma base sólida para o desenvolvimento do estudo de caso por meio da discussão dos principais conceitos e tecnologias envolvidos.

Posteriormente, realiza-se um mapeamento sistemático da literatura, cujo objetivo é identificar trabalhos similares, bem como lacunas no conhecimento, possibilitando a definição mais clara do escopo do estudo de caso. Além disso, essa fase permite levantar desafios, práticas e padrões comuns no uso de CSR e SSR.

Por fim, desenvolve-se um estudo de caso realista, no qual se descrevem requisitos, métodos e a arquitetura do sistema, bem como trechos de código ilustrativos da aplicação das técnicas. Nessa etapa, também são realizados testes de desempenho para avaliar o impacto das abordagens CSR e SSR em termos de tempo de resposta, carga no servidor, experiência do usuário e otimização para SEO.

Para concluir, elabora-se um relatório final que reúne análises quantitativas e qualitativas dos resultados, além de uma discussão sobre possíveis trabalhos futuros, desafios e benefícios do emprego de CSR e SSR em aplicações web modernas.

Figura 1 – Etapas de desenvolvimento da pesquisa



Fonte: os autores

1.5 Estrutura do Trabalho

Este trabalho está dividido em sete capítulos. O Capítulo 1 expõe o contexto do estudo, as justificativas desta pesquisa e os objetivos a serem atingidos. O Capítulo 2 apresenta conceitos fundamentais sobre CSR e SSR, abordando suas principais características, vantagens e desafios. O Capítulo 3 expõe o protocolo e o resultado do mapeamento da literatura, analisando estudos relacionados e identificando lacunas no conhecimento sobre a adoção dessas abordagens. Da mesma forma, o Capítulo 4 descreve os requisitos, métodos e organização do estudo de caso. Em seguida, o Capítulo 5 apresenta o estudo de caso desenvolvido, incluindo o *design* do sistema e trechos de código chave da implementação. Posteriormente, o Capítulo 6 apresenta os resultados obtidos com a execução dos testes de desempenho, analisando métricas como tempo de resposta, consumo de recursos, impacto no SEO e experiência do usuário. Por fim, o Capítulo 7 apresenta as conclusões

obtidas com o desenvolvimento deste trabalho, destacando os principais achados, desafios e recomendações para a escolha entre CSR e SSR em aplicações web modernas.

2 Fundamentação Teórica

Este capítulo apresenta os conceitos de *Client-Side Rendering (CSR)* e *Server-Side Rendering (SSR)*, abordando os princípios fundamentais do desenvolvimento web relacionados à renderização de conteúdo. Também são discutidos aspectos como SEO, desempenho, infraestrutura de serviços e impacto na experiência do usuário, estabelecendo a base teórica para o estudo de caso desenvolvido neste trabalho.

2.1 *Client-Side Rendering* (CSR)

A ***Client-Side Rendering (CSR)*** é uma técnica em que a geração da interface e do conteúdo final ocorre diretamente no navegador do usuário, utilizando JavaScript[1]. Nessa abordagem, o servidor envia um arquivo *HyperText Markup Language (HTML)* mínimo, contendo apenas a estrutura básica da página e referências a arquivos de estilo e scripts.(EMADAMERHO-ATORI, 2024)

Segundo Emadamerho-Atori (2024), o processo de renderização no cliente segue as seguintes etapas:

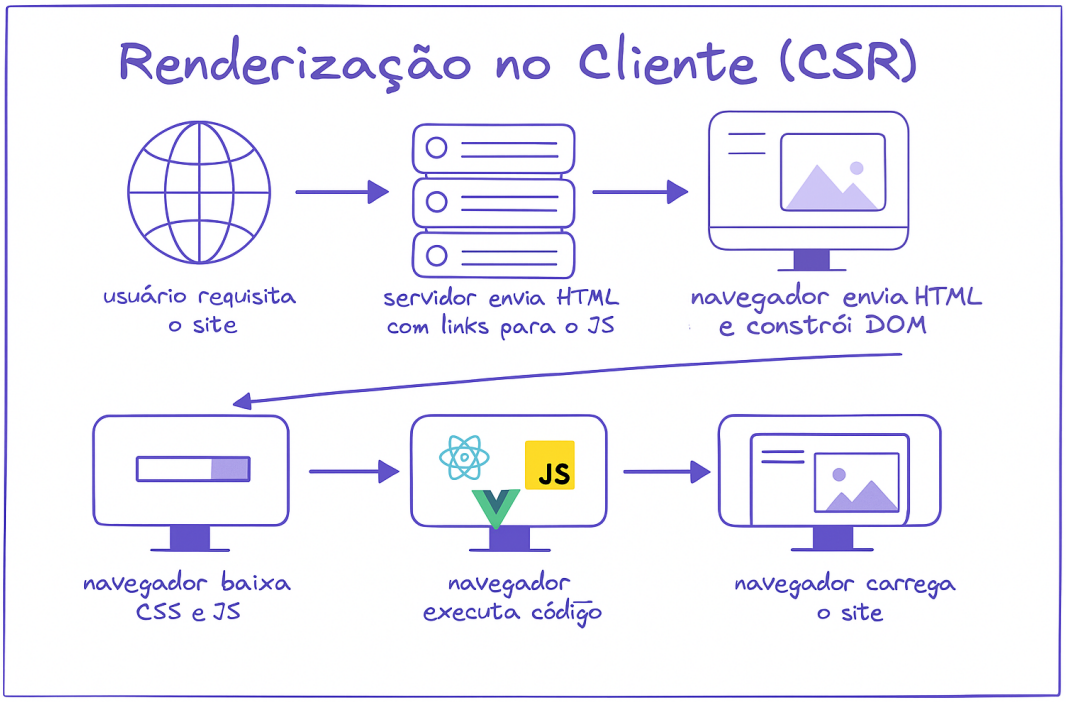
1. O servidor envia uma página HTML em branco contendo apenas links para os arquivos *Cascading Style Sheets (CSS)* e JavaScript.
2. O navegador interpreta o HTML e constrói a árvore do *Document Object Model (DOM)*
3. Os arquivos de estilo (CSS) e script (JavaScript) são baixados pelo navegador.
4. A aplicação é renderizada dinamicamente pelo JavaScript, incluindo elementos visuais como texto, imagens e botões.

[1] JavaScript é uma linguagem de programação interpretada, amplamente utilizada para adicionar interatividade e dinamismo às páginas web. Ela permite implementar funcionalidades como atualizações em tempo real, animações, mapas interativos e validações de formulários, enriquecendo a experiência do usuário. Juntamente com HTML e CSS, JavaScript compõe as três principais tecnologias da World Wide Web. Além de ser executada no navegador (CSR), a linguagem também pode ser utilizada no *Server-Side Rendering (SSR)* por meio de ambientes como Node.js(um ambiente de execução JavaScript de código aberto e multiplataforma que permite executar código JavaScript no lado do servidor, utilizando uma arquitetura orientada a eventos e não bloqueante, ideal para aplicações escaláveis e em tempo real.(NODE..., 2025)), possibilitando o desenvolvimento de aplicações completas com uma única linguagem.(JAVASCRIPT, 2025)

5. O conteúdo da página é atualizado de forma interativa conforme o usuário interage com a aplicação.

Esse modelo é comumente utilizado em aplicações *Single Page Application (SPA)*, nas quais o carregamento inicial é seguido por atualizações dinâmicas sem recarregamento da página. Frameworks como React[2], Vue.js[3], Angular[4] e Svelte[5] são amplamente utilizados para implementar CSR.

Figura 2 – Etapas do método de renderização no lado do cliente



Fonte: (EMADAMERHO-ATORI, 2024) (adaptado)

A Figura 2 ilustra visualmente o fluxo completo da renderização no lado do cliente (CSR). O processo é iniciado quando o usuário acessa o site em questão. Em resposta, o servidor envia o arquivo HTML básico, contendo apenas links para os arquivos de estilo CSS e scripts JavaScript responsáveis por carregar e renderizar o conteúdo da aplicação.

Na sequência, o navegador interpreta esse HTML e constrói a estrutura da página por meio da árvore DOM. No entanto, o conteúdo principal ainda não está visível.

[2] React é uma biblioteca JavaScript para construção de interfaces de usuário, desenvolvida pelo Facebook. (REACT, 2025)

[3] Vue.js é um framework JavaScript progressivo utilizado para a criação de interfaces web interativas, focado na camada de visualização. (VUE..., 2025)

[4] Angular é um framework para desenvolvimento de aplicações web, mantido pelo Google, que utiliza TypeScript como linguagem principal. (ANGULAR, 2025)

[5] Svelte é um framework JavaScript que realiza a compilação de componentes no momento do build, gerando código otimizado sem a necessidade de um virtual DOM. (SVELTE, 2025)

O navegador então precisa baixar os arquivos de estilo (CSS) e os scripts JavaScript referenciados no documento inicial.

Com os scripts carregados, o navegador executa o código JavaScript, que normalmente utiliza bibliotecas ou frameworks como React ou Vue para gerar dinamicamente o conteúdo da aplicação. Somente após essa etapa o conteúdo completo do site é finalmente exibido ao usuário, quando o navegador conclui o processo de renderização e o site é carregado completamente.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>CryptoWebsite</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <style>*,*:before,*:after{margin:0;padding:0;box-sizing:border-box;
    font-family:Inter,sans-serif}html{font-size:62.5%}</style>
  <link rel="stylesheet" href="styles.9d4c7581c7242.css">
</head>
<body>
  <app-root></app-root>
  <script src="runtime.6170988ad52a05db.js" type="module"></script>
  <script src="polyfills.574970d5ec4bdb97.js" type="module"></script>
  <script src="main.202d37bb6740400e.js" type="module"></script>
</body>
</html>
```

Esse padrão é típico de aplicações SPA, onde todo o conteúdo é inserido dinamicamente a partir da execução dos arquivos JavaScript. O elemento `<app-root>` funciona como ponto de entrada da aplicação, sendo substituído no navegador pelos componentes definidos no framework Angular. (EMADAMERHO-ATORI, 2024)

2.2 *Server-Side Rendering* (SSR)

A ***Server-Side Rendering (SSR)*** é uma abordagem em que a geração do conteúdo e da interface ocorre integralmente no servidor antes de ser enviada ao navegador do cliente. Ou seja, o servidor processa a lógica da aplicação, obtém dados necessários (por exemplo, em bancos de dados ou *APIs*) e retorna ao cliente um arquivo *HTML* já renderizado. Dessa forma, o navegador exibe imediatamente a página completa, sem precisar executar *scripts* para montar o conteúdo inicial (EMADAMERHO-ATORI, 2024).

Segundo Emadamerho-Atori (2024), o processo típico de renderização no lado do servidor pode ser descrito em quatro etapas principais:

1. O servidor recebe uma requisição para uma página e recupera os dados necessários para compor seu conteúdo (por exemplo, produtos de uma base de dados ou artigos de um blog).

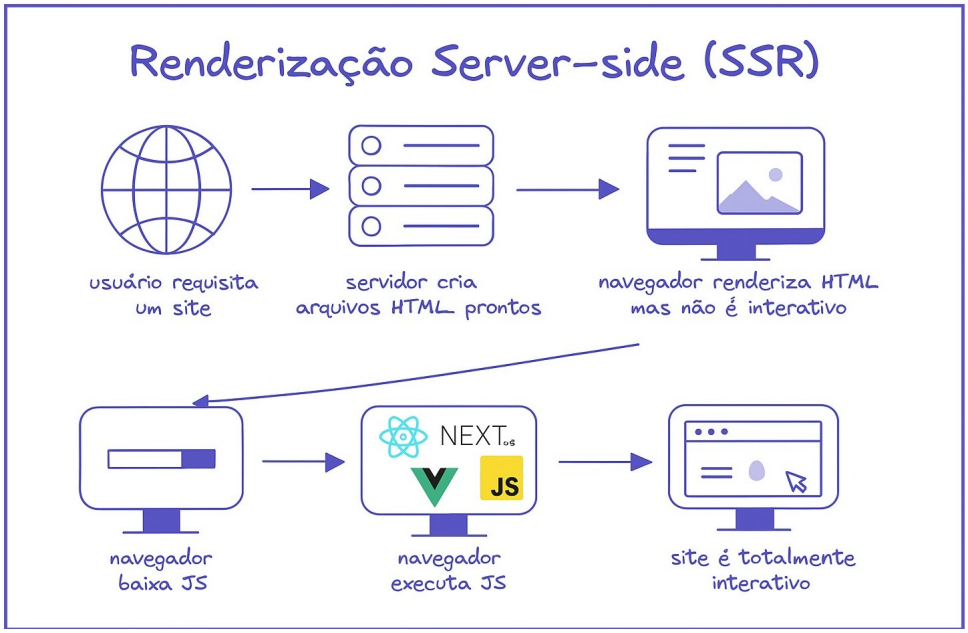
2. O servidor insere esses dados em um *template* HTML, gerando a estrutura final da página.

3. Em seguida, o servidor aplica estilos e finaliza a renderização, resultando em um documento HTML completamente montado.

4. Por fim, esse documento HTML é enviado ao navegador do usuário, exibindo a página prontamente, sem a necessidade de executar *JavaScript* durante o carregamento inicial.

Nesse modelo, a fase de hydration[6] ocorre após o carregamento inicial da página. costuma ocorrer após a entrega do conteúdo estático. Significa que, assim que o arquivo HTML é carregado e mostrado ao usuário, o *JavaScript* do lado do cliente assume o controle para tratar as interações e atualizações dinâmicas subsequentes. Dessa forma, o SSR beneficia tanto o primeiro acesso (tornando o conteúdo visível rapidamente) quanto o SEO, por exibir ao rastreador dos mecanismos de busca um código HTML completo. (EMADAMERHO-ATORI, 2024).

Figura 3 – Etapas do método de renderização no lado do servidor



Fonte: (EMADAMERHO-ATORI, 2024) (adaptado)

[6] Hydration é uma etapa essencial no SSR, em que o JavaScript torna interativo o conteúdo HTML previamente renderizado no servidor.

A Figura 3 ilustra o fluxo de uma aplicação SSR. Ao receber a requisição, o servidor gera a página completa em HTML e a envia ao cliente. Essa estratégia costuma ser vantajosa em cenários onde o carregamento inicial rápido e a indexação por motores de busca são prioridades, como em sites de e-commerce e páginas de *landing*, permitindo que o usuário visualize o conteúdo de forma imediata.

Meta-frameworks[7] como *Next.js*[8] , *Nuxt.js*[9] , *SvelteKit*[10] , *Angular Universal*[11] , *Remix*[12] , *Astro*[13] e *Qwik*[14] são amplamente utilizados para construir aplicações com suporte a SSR, simplificando a configuração e fornecendo recursos prontos para lidar com roteamento, recuperação de dados e *hidration*.

No Código 2.1, pode-se observar que o arquivo HTML já contém todo o *markup* necessário para exibir o conteúdo da página. Assim que o navegador recebe esse arquivo, o usuário já visualiza o cabeçalho, o texto e o layout definidos. Posteriormente, o *JavaScript* baixado (por exemplo, `main.js`) pode entrar em ação para lidar com eventos, rotas adicionais e atualizações dinâmicas, caso o desenvolvedor deseje funcionalidades mais interativas.

Por fim, aplicações SSR tendem a apresentar melhor performance em termos de

[7] Os meta-frameworks são sistemas de desenvolvimento web que operam em um nível superior a frameworks tradicionais, como React, Vue ou Svelte. Seu principal objetivo é agregar e organizar múltiplas funcionalidades comuns no desenvolvimento de aplicações, oferecendo uma estrutura mais completa e opinativa. (HOLMES, 2022)

[8] Next.js é um framework de código aberto para React que facilita a criação de aplicações web com renderização no lado do servidor (SSR) e geração de sites estáticos. Fornece recursos como roteamento baseado em arquivos, pré-renderização e suporte a APIs. (NEXT. . . , 2024)

[9] Nuxt.js é um framework baseado em Vue.js que simplifica a criação de aplicações universais, oferecendo renderização no lado do servidor, geração de sites estáticos e uma arquitetura modular que facilita a escalabilidade. (NUXT. . . , 2024)

[10] SvelteKit é um framework para Svelte que oferece uma experiência de desenvolvimento simplificada, permitindo a criação de aplicações com renderização no lado do servidor e no cliente, além de otimizações de desempenho. (SVELTEKIT, 2024)

[11] Angular Universal é a solução oficial de renderização no lado do servidor para aplicações Angular, permitindo a renderização de páginas no servidor para melhorar o desempenho e a indexação por mecanismos de busca. (ANGULAR. . . , 2024)

[12] Remix é um framework full-stack para React que foca na experiência do desenvolvedor e no desempenho, oferecendo renderização no lado do servidor e um modelo de dados baseado em carregadores e ações. (REMIX, 2024)

[13] Astro é um framework moderno que permite a construção de sites rápidos, carregando apenas o JavaScript necessário e permitindo o uso de componentes de diferentes frameworks como React, Svelte e Vue. (ASTRO, 2024)

[14] Qwik é um framework JavaScript que introduz o conceito de aplicações web "resumíveis", visando tempos de carregamento instantâneos e desempenho aprimorado, utilizando renderização no lado do servidor e aprimoramentos progressivos. (QWIK, 2024)

time-to-first-byte[15] e de *indexabilidade*[16] por motores de busca, ao mesmo tempo em que podem demandar maior carga de processamento no servidor. A escolha por SSR ou não, portanto, depende do perfil da aplicação e das prioridades do projeto, considerando fatores como volume de tráfego, necessidade de interatividade e requisitos de otimização de conteúdo.

Código 2.1 – Exemplo de HTML mínimo em aplicação Next.js com SSR

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="utf-8">
5    <title>My SSR App</title>
6    <meta name="viewport" content="width=device-width, initial-scale=1">
7    <style>
8      /* Exemplo simples de estilo inline */
9      body {
10       margin: 0;
11       font-family: Arial, sans-serif;
12       background: #f6f6f6;
13     }
14     h1 { color: #333; }
15   </style>
16 </head>
17 <body>
18   <!-- Conte do j  processado e inserido no servidor -->
19   <div id="__next">
20     <header>
21       <h1>Ol , mundo!</h1>
22     </header>
23     <main>
24       <p>Este conte do foi renderizado no servidor usando Next.js.</p>
25     </main>
26   </div>
27   <!-- Scripts do Next.js para intera  o no cliente -->
28   <script src="/_next/static/chunks/main.js" defer></script>
29 </body>
30 </html>
```

[15] O *time-to-first-byte* (TTFB) é uma métrica que mede o tempo decorrido entre o envio de uma solicitação HTTP pelo cliente e o recebimento do primeiro byte da resposta do servidor. Um TTFB menor indica maior rapidez na resposta do servidor, impactando diretamente na velocidade de carregamento da página e na experiência do usuário. (SMITH, 2025)

[16] A *indexabilidade* refere-se à capacidade dos motores de busca de rastrear e indexar o conteúdo de uma página web. Aplicações SSR, ao fornecerem conteúdo totalmente renderizado no servidor, facilitam a indexação eficiente pelos motores de busca, melhorando a visibilidade nos resultados de pesquisa. (GüNAçAR, 2025)

2.3 Fundamentos de Desenvolvimento Web

Para entender como as abordagens SSR e CSR se inserem no cenário de desenvolvimento web, é fundamental revisar protocolos, modelos de arquitetura e ferramentas. Os fundamentos de desenvolvimento web englobam os princípios, tecnologias e práticas essenciais para a criação e manutenção de aplicações acessíveis via internet.

O desenvolvimento web baseia-se na arquitetura cliente-servidor, onde o cliente (geralmente um navegador) solicita recursos ao servidor, que processa essas requisições e retorna os dados necessários. Essa interação é mediada por protocolos como o *Hypertext Transfer Protocol (HTTP)* , que define as regras de comunicação entre cliente e servidor.

As tecnologias fundamentais incluem *HyperText Markup Language (HTML)* para estruturação do conteúdo, *Cascading Style Sheets (CSS)* para estilização e JavaScript para interatividade. Essas linguagens permitem a construção de interfaces dinâmicas e responsivas. Além disso, o desenvolvimento web envolve práticas como controle de versão, testes automatizados e integração contínua, que garantem a qualidade e a escalabilidade das aplicações (VIANA, 2024).

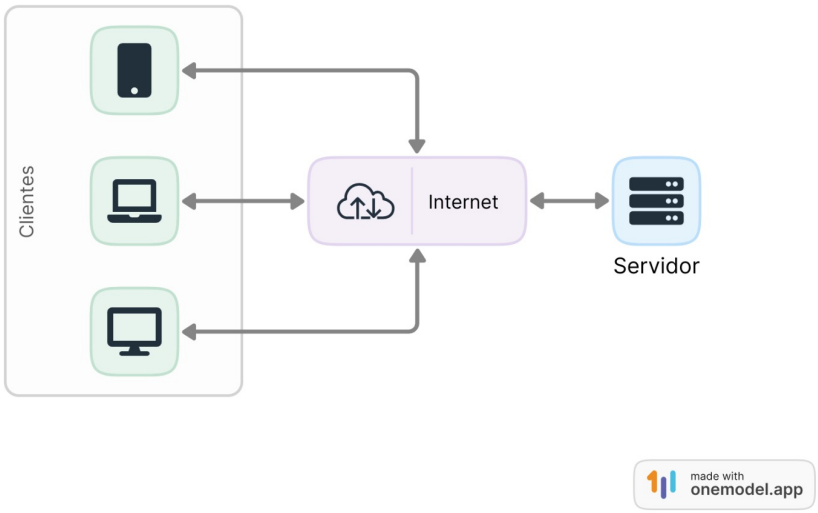
2.3.1 Arquitetura Cliente-Servidor

A arquitetura cliente-servidor é um modelo amplamente adotado no desenvolvimento de aplicações web, caracterizado pela separação entre dois componentes principais: o **cliente**, responsável pela interface com o usuário, e o **servidor**, que processa solicitações e fornece os recursos necessários (ControleNet, 2025).

Nesse modelo, os clientes — como navegadores em diferentes dispositivos — enviam requisições através da internet, enquanto os servidores respondem disponibilizando dados, arquivos e serviços. Essa divisão de responsabilidades favorece a escalabilidade, facilita a manutenção e permite que cliente e servidor operem em plataformas distintas (VIANA, 2024).

A Figura 4 ilustra, de forma simplificada, esse fluxo de comunicação entre cliente e servidor.

Figura 4 – Comunicação entre cliente e servidor.



Fonte: (VIANA, 2024)

A arquitetura cliente-servidor apresenta características que contribuem para sua ampla adoção em aplicações web. Entre elas, destaca-se a **distribuição de responsabilidades**, onde o servidor gerencia dados e processos mais complexos, enquanto o cliente lida com a interface e a interação com o usuário.

Outro ponto relevante é a **independência entre plataformas**, possibilitada pelo uso de protocolos padronizados, o que permite a comunicação entre diferentes dispositivos e sistemas operacionais. Além disso, esse modelo favorece a **facilidade de manutenção**, já que atualizações podem ser feitas no servidor sem necessidade de intervenção nos dispositivos dos usuários.

Na web, essa arquitetura é implementada por padrão: navegadores atuam como clientes, enviando requisições HTTP que são processadas por servidores, os quais respondem com páginas e recursos solicitados (VIANA, 2024).

2.3.2 Protocolo HTTP

O **Protocolo de Transferência de Hipertexto** (HTTP) é a base da comunicação na World Wide Web, definindo como clientes (navegadores) e servidores trocam informações. Ele especifica a estrutura das requisições e respostas, permitindo a recuperação de recursos como documentos html), imagens e vídeos (MDN Web Docs, 2024).

O **Funcionamento do HTTP**opera no modelo cliente-servidor, onde o cliente

inicia uma requisição e o servidor responde com os recursos solicitados ou mensagens de erro, se aplicável. Cada interação consiste em uma mensagem de requisição do cliente e uma mensagem de resposta do servidor. As mensagens HTTP são compostas por:

- **Linha de início:** Indica o método HTTP (como `GET` ou `POST`) e o caminho do recurso.
- **Cabeçalhos:** Fornecem informações adicionais sobre a requisição ou resposta, como tipo de conteúdo e codificação.
- **Corpo:** Contém os dados enviados ou recebidos, sendo opcional dependendo do método utilizado.

(MDN Web Docs, 2024)

Métodos HTTP são operações definidas pelo protocolo que especificam a ação a ser realizada em um recurso. Os métodos mais comuns incluem:

- **GET:** Solicita a representação de um recurso específico. Requisições GET devem ser utilizadas apenas para recuperar dados.
- **POST:** Envia dados ao servidor para processamento, como o envio de formulários.
- **PUT:** Atualiza um recurso existente ou cria um novo se não existir.
- **DELETE:** Remove um recurso específico.
- **HEAD:** Similar ao GET, mas solicita apenas os cabeçalhos da resposta, sem o corpo.

Cada método possui uma finalidade específica e deve ser utilizado conforme a necessidade da aplicação (Wikipedia, 2024).

Códigos de Status HTTP são códigos de três dígitos que indicam o resultado de uma requisição feita pelo cliente ao servidor. Eles são agrupados em cinco classes principais:

- **1xx (Informativo):** Indica que a requisição foi recebida e o processo continua.
- **2xx (Sucesso):** Indica que a requisição foi bem-sucedida. Exemplo: 200 OK.
- **3xx (Redirecionamento):** Indica que é necessário tomar medidas adicionais para completar a requisição. Exemplo: 301 Moved Permanently.

- **4xx (Erro do Cliente):** Indica que houve um erro na requisição do cliente. Exemplo: 404 Not Found.
- **5xx (Erro do Servidor):** Indica que o servidor falhou ao processar uma requisição válida. Exemplo: 500 Internal Server Error.

Esses códigos auxiliam na identificação e resolução de problemas durante a comunicação HTTP (MDN Web Docs, 2024).

Evolução do HTTP refere-se às revisões progressivas do protocolo com o objetivo de aprimorar sua eficiência, segurança e desempenho ao longo do tempo. As principais versões são:

- **HTTP/1.0:** Primeira versão oficial do protocolo, em que cada requisição exigia uma nova conexão com o servidor.
- **HTTP/1.1:** Introduziu conexões persistentes, permitindo múltiplas requisições por conexão. Trouxe também melhorias no controle de cache e suporte a novos métodos.
- **HTTP/2:** Implementou multiplexação, compressão de cabeçalhos e priorização de fluxos, resultando em uma transferência de dados mais rápida e eficiente.
- **HTTP/3:** Baseado no protocolo QUIC, substitui o TCP pelo UDP, oferecendo conexões mais rápidas e seguras, com menor latência e melhor desempenho em redes instáveis.

Essas atualizações refletem a evolução das necessidades da web e a busca por protocolos mais robustos e otimizados (Cloudflare, 2025).

HTTP e HTTPS representam protocolos utilizados para comunicação na web, com a principal distinção centrada na segurança da transmissão dos dados.

O ***Hypertext Transfer Protocol Secure (HTTPS)*** é uma extensão do HTTP que adiciona uma camada de proteção por meio do protocolo *Transport Layer Security (TLS)* ou, anteriormente, *Secure Sockets Layer (SSL)*. Essa camada de segurança garante a confidencialidade, integridade e autenticidade dos dados trafegados entre cliente e servidor.

A criptografia utilizada impede que terceiros acessem ou modifiquem as informações transmitidas, o que é fundamental em transações sensíveis, como cadastros, pagamentos e autenticações. Além disso, o uso de *certificados digitais* garante que o site visitado é

realmente aquele que afirma ser, protegendo os usuários contra ataques como o *man-in-the-middle*[17].

Enquanto o HTTP tradicional opera normalmente na porta TCP 80, o HTTPS utiliza, por convenção, a porta 443. Atualmente, o uso do HTTPS é fortemente recomendado — e até exigido por navegadores modernos — como padrão de segurança para qualquer aplicação web, contribuindo para a privacidade e confiança dos usuários (Wikipedia, 2024).

2.3.3 Modelos de Arquitetura Web

Os modelos arquiteturais variam conforme os requisitos de escalabilidade, manutenção e desempenho:

- **Arquitetura Monolítica**: Um único projeto concentra *frontend* e *backend*, frequentemente usando SSR. Possui inicialização simples, mas pode tornar-se complexo de manter e escalar.
- **Microserviços**: Divide a aplicação em múltiplos serviços independentes. Cada serviço pode escolher a melhor abordagem de renderização (SSR ou CSR), facilitando a escalabilidade seletiva.
- **Serverless**: As funções são executadas sob demanda em plataformas de nuvem, onde a renderização pode ocorrer tanto no servidor (funções que retornam HTML) quanto no cliente (ao entregar apenas APIs).

2.3.4 Ferramentas e *Frameworks*

O ecossistema de desenvolvimento web oferece diversas ferramentas que simplificam SSR e CSR:

- **SSR**: *Next.js* (React), *Nuxt.js* (Vue), *SvelteKit* (Svelte), entre outros.
- **CSR**: React, Vue.js, Angular e muitas bibliotecas voltadas para *Single Page Applications* (SPA).

2.4 Infraestrutura de Serviço Web

A decisão por SSR ou CSR influencia diretamente a infraestrutura necessária:

[17] Um ataque *man-in-the-middle* ocorre quando um invasor intercepta e possivelmente altera a comunicação entre duas partes que acreditam estar se comunicando diretamente. Isso pode permitir que o invasor capture informações sensíveis ou injete dados maliciosos na comunicação.(Wikipedia, 2025)

- **Servidores e Processamento**: Em SSR, o servidor gera páginas dinamicamente, aumentando a carga de CPU. Já em CSR, o servidor atua mais como um provedor de arquivos estáticos e APIs.
- ***Content Delivery Networks* (CDNs)**: Tanto para SSR quanto para CSR, uma CDN pode melhorar a distribuição de arquivos estáticos (HTML, CSS, JavaScript, imagens) e reduzir a latência.
- **Escalabilidade**: Aplicações com alto número de requisições precisam de estratégias adequadas para lidar com picos de acesso. Em SSR, muitas requisições simultâneas podem sobrecarregar o servidor; em CSR, o foco está em serviços de dados e na entrega eficiente de arquivos iniciais.

Segurança também se faz presente em ambas as abordagens. Boas práticas incluem:

- Uso de **HTTPS** para proteger a comunicação.
- Implementação de **CORS** (Cross-Origin Resource Sharing) quando necessário.
- Tratamento de **tokens de sessão/autenticação** com cuidado para evitar vazamento de dados.

—

2.5 Experiência do Usuário (*User Experience* – UX)

Um dos objetivos principais ao optar por SSR ou CSR é oferecer uma experiência de usuário satisfatória. Alguns aspectos relevantes incluem:

- **Tempo de Carregamento Inicial**: Em SSR, o conteúdo aparece mais rápido para o usuário no primeiro acesso. Em CSR, embora o primeiro carregamento possa ser mais lento (devido ao download e execução de scripts), a navegação interna torna-se mais rápida após a aplicação já estar carregada no navegador.
- **Interatividade e Navegação**: CSR geralmente proporciona transições fluidas entre páginas e atualizações em tempo real sem recarregamento completo. SSR, porém, pode recorrer a estratégias de atualização parcial, como AJAX, para melhorar a interatividade.

- **Acessibilidade**: Independentemente da abordagem, a aplicação deve atender a boas práticas de acessibilidade, garantindo que leitores de tela e outras tecnologias assistivas consigam interpretar adequadamente o conteúdo.
- **Consistência de Interface**: A aplicação deve manter uma experiência coesa, independentemente de páginas estarem sendo renderizadas no servidor ou no cliente.

2.5.1 *Search Engine Optimization* (SEO)

O **SEO** é o conjunto de técnicas para melhorar o ranqueamento de um site nos resultados dos mecanismos de busca. No contexto de CSR e SSR, o SEO é especialmente importante por afetar:

- **Indexação de Conteúdo**: No SSR, o HTML completo chega aos robôs de busca, facilitando a indexação imediata. Já em CSR, se o robô não executar JavaScript, pode ocorrer indexação incompleta ou ausente.
- **Metadados Dinâmicos**: O uso de títulos, descrições e tags *Open Graph* deve ser planejado para gerar conteúdo correto em cada rota, principalmente em aplicações SPA.
- **Velocidade de Carregamento**: O tempo de carregamento é um fator relevante no ranqueamento, exigindo otimizações como *caching*, minificação de scripts, compressão de imagens e *lazy loading*.

Alguns mecanismos de busca modernos suportam *renderização dinâmica* (executando JavaScript para rastrear páginas em CSR), mas configurações incorretas podem prejudicar a visibilidade do site nos resultados de pesquisa.

—

—

Conclui-se, portanto, que SSR e CSR oferecem caminhos distintos para a criação de aplicações web, cada qual com implicações em termos de desempenho, SEO, infraestrutura e experiência do usuário. Nas próximas seções deste trabalho, será apresentado um estudo de caso prático que visa comparar e avaliar as duas abordagens em diferentes cenários de uso.

3 Trabalhos Relacionados

4 Estudo de Caso: Requisitos, Organização e Métodos

Este capítulo apresenta o contexto, os requisitos, a organização e os métodos utilizados no desenvolvimento do estudo de caso.

4.1 Contexto

O estudo de caso é realizado em uma empresa fictícia.

5 Estudo de Caso: Design, Implementação e Testes

Este capítulo apresenta o desenvolvimento do estudo de caso.

5.1 Ciclo de Vida do Desenvolvimento de Software

6 Resultados e Discussões

Este capítulo apresenta os resultados obtidos com a execução dos testes de carga descritas.

6.1 Resultados

Essa seção apresenta uma análise gráfica dos resultados obtidos com a execução dos testes de carga.

7 Conclusão

Este trabalho apresentou um estudo de caso.

Referências

ANGULAR. 2025. Acessado em: 11 de abril de 2025. Disponível em: <https://angular.io/guide/architecture>. Citado na página 18.

ANGULAR Universal. 2024. Acessado em: 10 de abril de 2025. Disponível em: <https://angular.io/guide/universal>. Citado na página 21.

ASTRO. 2024. Acessado em: 10 de abril de 2025. Disponível em: <https://astro.build/>. Citado na página 21.

Cloudflare. *O que é HTTP?* 2025. Acessado em 12 de abril de 2025. Disponível em: <https://www.cloudflare.com/pt-br/learning/ddos/glossary/hypertext-transfer-protocol-http/>. Citado na página 26.

CONNER, G. Tornando o instagram.com mais rápido: Parte 2. *Engenharia do Instagram*, 2019. Disponível em: <https://instagram-engineering.com/making-instagram-com-faster-part-2-f350c8fba0d4>. Citado na página 13.

ControleNet. *Cliente-Servidor, uma estrutura lógica para a computação centralizada*. 2025. Disponível em: <https://www.controle.net/faq/cliente-servidor-uma-estrutura-para-a-computacao-centralizada>. Citado na página 23.

EMADAMERHO-ATORI, N. Client-side rendering (csr) vs. server-side rendering (ssr). *Prismic Blog*, 2024. Disponível em: <https://prismic.io/blog/client-side-vs-server-side-rendering>. Citado 5 vezes nas páginas 13, 17, 18, 19 e 20.

GODBOLT, M. *Frontend Architecture for Design Systems*. O'Reilly Media, Inc., 2016. ISBN 9781491926734. Disponível em: <https://www.oreilly.com/library/view/frontend-architecture-for/9781491926772/>. Citado na página 13.

GOOGLE. 2010. Acessado em: 14 de fevereiro de 2025. Disponível em: <https://developers.google.com/search/blog/2010/04/using-site-speed-in-web-search-ranking>. Citado na página 12.

GüNAçAR, O. Time to first byte (ttfb) – easy to understand, difficult to improve. *OnCrawl*, 2025. Acessado em: 10 de abril de 2025. Disponível em: <https://www.oncrawl.com/technical-seo/time-to-first-byte-what-and-why-important-part-1/>. Citado na página 22.

HOLMES, B. Understanding the javascript meta-framework ecosystem. *Prismic Blog*, 2022. Disponível em: <https://prismic.io/blog/javascript-meta-frameworks-ecosystem>. Citado na página 21.

JAVASCRIPT. 2025. Acessado em: 11 de abril de 2025. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Learn_web_development/Core/Scripting/What_is_JavaScript>. Citado na página 17.

MDN Web Docs. *HTTP - Visão Geral*. 2024. Acessado em 12 de abril de 2025. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP>. Citado 3 vezes nas páginas 24, 25 e 26.

NEARY, A. Rearchitecting airbnb's frontend. *Medium - Airbnb Engineering*, 2017. Disponível em: <https://medium.com/airbnb-engineering/rearchitecting-airbnbs-frontend-5e213efc24d2>. Citado na página 13.

NEXT.JS. 2024. Acessado em: 10 de abril de 2025. Disponível em: <https://nextjs.org/>. Citado na página 21.

NODE.JS. 2025. Acessado em: 11 de abril de 2025. Disponível em: <https://nodejs.org/en/docs/>. Citado na página 17.

NUXT.JS. 2024. Acessado em: 10 de abril de 2025. Disponível em: <https://nuxtjs.org/>. Citado na página 21.

QWIK. 2024. Acessado em: 10 de abril de 2025. Disponível em: <https://qwik.builder.io/>. Citado na página 21.

REACT. 2025. Acessado em: 11 de abril de 2025. Disponível em: <https://react.dev/learn>. Citado na página 18.

REMIX. 2024. Acessado em: 10 de abril de 2025. Disponível em: <https://remix.run/>. Citado na página 21.

SITEEFY. *How many websites are there?* 2021. Disponível em: <https://siteefy.com/how-many-websites-are-there/>. Citado na página 12.

SMITH, C. Time to first byte (ttfb) – easy to understand, difficult to improve. *OuterBox Design*, 2025. Acessado em: 10 de abril de 2025. Disponível em: <https://www.outerboxdesign.com/uncategorized/time-to-first-byte-ttfb>. Citado na página 22.

SVELTE. 2025. Acessado em: 11 de abril de 2025. Disponível em: <https://svelte.dev/docs>. Citado na página 18.

SVELTEKIT. 2024. Acessado em: 10 de abril de 2025. Disponível em: <https://kit.svelte.dev/>. Citado na página 21.

TRADEOFFS in Server Side and Client Side Rendering. 2015. Acessado em: 14 de fevereiro de 2025. Disponível em: <https://www.industrialempathy.com/posts/tradeoffs-in-server-side-and-client-side-rendering/>. Citado na página 12.

VIANA, J. Fundamentos da web. *Guia Web*, 2024. Disponível em: <https://jesielviana.gitbook.io/guiaweb/2.-fundamentos-da-web>. Citado 2 vezes nas páginas 23 e 24.

VUE.JS. 2025. Acessado em: 11 de abril de 2025. Disponível em: <https://vuejs.org/guide/introduction.html>. Citado na página 18.

WAGNER, J. L. *Web Performance in Action*. Manning Publications, 2016. ISBN 9781617293771. Disponível em: <https://www.manning.com/books/web-performance-in-action?a_aid=webopt&a_bid=63c31090>. Citado 2 vezes nas páginas 12 e 13.

Wikipedia. *Protocolo de Transferência de Hipertexto*. 2024. Acessado em 12 de abril de 2025. Disponível em: <https://pt.wikipedia.org/wiki/Hypertext_Transfer_Protocol>. Citado 2 vezes nas páginas 25 e 27.

Wikipedia. *Man-in-the-middle attack*. 2025. Acessado em 13 de abril de 2025. Disponível em: <https://en.wikipedia.org/wiki/Man-in-the-middle_attack>. Citado na página 27.